# Konzeptionelle Modellierung von Plugin-Systemen mit Petrinetzen

Dissertation
zur Erlangung
des Doktorgrades

vorgelegt von

Michael Duvigneau
aus Hamburg

Universität Hamburg
Fakultät für Mathematik, Informatik und
Naturwissenschaften
Department Informatik

Hamburg 2009

Genehmigt von der MIN-Fakultät, Department Informatik, der Universität Hamburg
auf Antrag von
Dr. D. Moldt,
Prof. Dr. R. Valk und
Prof. Dr. K. van Hee

Hamburg, den 25. November 2009

Prof. Dr. H. Oberquelle
Leiter Department Informatik

*Für Silke, Marthe und Thomas*

**Zusammenfassung**

Die Komplexität moderner Softwaresysteme wächst nicht zuletzt aufgrund des Wunsches, sowohl aktuelle als auch zukünftige Bedürfnisse vieler verschiedener Anwender mit einem einzigen System befriedigen zu wollen. Eine in der Praxis oft eingesetzte Lösung zur Bewältigung der sich ergebenden Komplexität besteht darin, ein *Plugin-System* in die Software zu integrieren. Ein Plugin-System erlaubt es Anwendern, ein relativ schlankes Kernsystem während seines Einsatzes an individuelle Bedürfnisse anzupassen, indem vorgefertigte Plugins auf einfachste Weise zum Softwaresystem hinzugefügt oder wieder daraus entfernt werden. Um den Konfigurationskomfort für Anwender weiter zu erhöhen, werden zunehmend *dynamisch*, also während der Laufzeit rekonfigurierbare Plugin-Systeme implementiert.

Dem häufigen praktischen Einsatz des Plugin-Konzepts steht bisher eine relativ geringe Aufmerksamkeit in der Forschung gegenüber. Ziel dieser Arbeit ist daher die Definition, Modellierung und Implementierung zentraler Konzepte von Plugin-Systemen. Eine weitere Herausforderung ergibt sich aus der Tatsache, dass moderne Softwaresysteme – also auch Plugin-Systeme – inhärent *nebenläufig* sind. Die sich daraus ergebenden Möglichkeiten und Risiken werden im Rahmen der Modellierung behandelt.

Einige Paradigmen und Techniken aus dem Bereich der Softwareentwicklung werden bereits zur Beschreibung von Plugin-Systemen eingesetzt. Hierzu zählen das Paradigma der komponentenbasierten Softwareentwicklung sowie die Techniken zur Beschreibung von Softwarearchitekturen und der Entwurfsmuster. Die im Rahmen dieser Ansätze formulierten Konzepte zeigen jedoch Schwächen in der Modellierung von dynamischen Änderungen und von möglicherweise vorhandener Nebenläufigkeit. Daher werden diese Ansätze durch das Paradigma der agentenorientierten Softwareentwicklung und die Technik der Modellierung mit Referenznetzen ergänzt. Beide Ansätze sind auf die Modellierung nebenläufiger, sich dynamisch verändernder Systeme ausgerichtet, werden aber noch nicht im Kontext von Plugin-Systemen eingesetzt.

Das breite Spektrum praktisch eingesetzter Plugin-Systeme wird anhand der komplexen Softwarearchitekturen der Mozilla-Anwendungssuite und der Eclipse-Plattform sowie einiger weiterer Systeme vorgestellt. Aus diesem Spektrum werden die einfache (dynamische) Rekonfigurierbarkeit, eine starke Kapselung und das Konzept der intendierten Erweiterbarkeit als zentrale Eigenschaften des Plugin-Konzepts herausgearbeitet. Weitere Anforderungen an das Modell sind die Festlegung von Plugin-Lebenszyklen, die Existenz mehrerer Verwaltungsebenen, das Berücksichtigen von Nebenläufigkeit sowie eine intuitive Modellierung mit präziser Semantik. Der auf Basis dieser Eigenschaften definierte Plugin-Begriff wird zu den Begriffen des Objekts, des Moduls, der Komponente und des Agenten in Beziehung gesetzt.

Zum Schärfen der Begriffe werden zentrale Konzepte von Plugin-Systemen mittels Referenznetzen nach dem Prinzip der schrittweisen Verfeinerung modelliert („Implementierung durch Spezifikation“). Das Modell deckt sämtliche eben genannten Anforderungen ab, schränkt die Nebenläufigkeit nicht unnötig ein und ist mit einer präzisen operationalen Semantik ausgestattet. Die Tauglichkeit des modellierten Konzepts wird zum einen durch einen Abgleich mit den vielfältigen Eigenschaften existierender Plugin-Systeme belegt. Zum anderen werden die konzeptionellen Ergebnisse in Form mehrerer Prototypen auf das Petrinetzwerkzeug Renew übertragen.

**Abstract**

The complexity of modern software systems increases. Among other reasons this is due to the goal of software engineers to satisfy actual and future needs of many different users with one single system. In order to cope with the complexity of such systems it is a common solution in real-world software systems to integrate a *plug-in system*. A plug-in system enables users to adapt a slim core system to their individual needs after deployment. A user just has to add or remove pre-fabricated plug-ins in a simple configuration process. To further increase the comfort of the configuration process it becomes more and more common to implement *dynamically* reconfigurable plug-in systems.

While plug-in systems are rather common to encounter in the real world, researchers brought rather little attention to the topic until now. Therefore, the goal of this thesis is to define, model, and implement essential concepts of plug-in systems. An additional challenge comes from the observation that many modern software systems – and thus, plug-in systems as well – are inherently *concurrent* systems. The possibilities and risks that may be deduced from this observation are caught within the modelling process.

Some paradigms and techniques within the area of software engineering are already used to describe plug-in systems. These include the paradigm of component-based software engineering and techniques to describe software architectures or design patterns. However, the concepts formulated within these approaches show some weaknesses when it comes to the modelling of dynamic changes or concurrency. Therefore, they are extended by the paradigm of agent-oriented software engineering and the technique of modelling with reference nets. Both approaches have not been used within the context of plug-in systems up to now.

The broad spectrum of real-world plug-in systems is outlined along some complex example systems, namely the Mozilla application suite and the Eclipse platform. These two large examples are complemented by some more systems. From this spectrum, essential properties of plug-in systems are drawn. These are simple (dynamic) reconfigurability, strong encapsulation, and the concept of intended extensibility. The conceptual model should satisfy further requirements. It should be able to define plug-in life-cycles, cope with multiple layers of management, and allow concurrency within the system. Furthermore, the conceptual model needs an intuitive visual representation with precise semantics. The concept of a plug-in is defined based on these characteristics and related to the concepts of objects, modules, components and agents.

To enhance the definitions, the plug-in concepts are modelled based on reference nets along the principle of consecutive refinement ("implementation by specification"). The model covers all requirements mentioned above, does not restrict concurrency more than necessary and is supported by precise operational semantics. A comparison with the manifold characteristics of real-world plug-in systems illustrates the suitability of the model. Furthermore, the conceptual results are transferred to the Petri net tool Renew by implementing several prototypes.

# Inhaltsverzeichnis


**1 Einleitung** **1**
- 1.1 Problemstellung . . . 4
- 1.2 Zielsetzung . . . 7
- 1.3 Aufbau der Arbeit . . . 9

**2 Grundlagen** **13**
- 2.1 Referenznetze . . . 15
  - 2.1.1 Petrinetze . . . 16
  - 2.1.2 Netze in Netzen . . . 21
  - 2.1.3 Referenznetze . . . 28
  - 2.1.4 Renew . . . 35
- 2.2 Komponentenbasierte Softwareentwicklung . . . 43
  - 2.2.1 Der Komponentenbegriff . . . 44
  - 2.2.2 Komponentenmodelle . . . 47
  - 2.2.3 Komponenten und Dienste . . . 49
  - 2.2.4 Dynamische Veränderungen in komponentenbasierten Systemen 51
  - 2.2.5 CBSE und Plugin-Systeme . . . 53
- 2.3 Beschreibung von Softwarearchitekturen . . . 58
  - 2.3.1 Grundbegriffe . . . 59
  - 2.3.2 Eigenschaften von Architekturbeschreibungssprachen . . . 61
  - 2.3.3 UML als Architekturbeschreibungssprache . . . 64
  - 2.3.4 Dynamische Architekturbeschreibungssprachen . . . 69
  - 2.3.5 Architekturbeschreibungssprachen und Petrinetze . . . 72
- 2.4 Entwurfsmuster . . . 75
  - 2.4.1 Muster zur Entkopplung von Objekten . . . 76
  - 2.4.2 Muster zum Herstellen dynamischer Beziehungen . . . 77
  - 2.4.3 Das Plugin-Konzept als Muster . . . 81
  - 2.4.4 Muster zur Dekomposition von Plugins . . . 83
  - 2.4.5 Diskussion . . . 85
- 2.5 Agentenorientierte Softwareentwicklung . . . 87
  - 2.5.1 Grundbegriffe . . . 87
  - 2.5.2 Agenten und Petrinetze . . . 97
  - 2.5.3 AOSE und CBSE . . . 101
- 2.6 Diskussion . . . 103

**3 Plugin-Systeme** **109**
- 3.1 Netscape/Mozilla . . . 111
  - 3.1.1 Plugins . . . 112

3.1.2 Extensions . . . 121
3.1.3 Zusammenfassung . . . 134
3.2 Eclipse . . . 135
3.2.1 Plugins . . . 137
3.2.2 Zusammenfassung . . . 155
3.3 Weitere Arbeiten . . . 156
3.3.1 Java Plugin Framework (JPF) . . . 156
3.3.2 NetBeans . . . 157
3.3.3 Client Application Platform .NET (CAP.NET) . . . 161
3.3.4 MagicBeans . . . 162
3.3.5 Plugins und Multiagentensysteme . . . 165
3.3.6 Integrieren von Modulsystemen . . . 167
3.3.7 Modularisierung und Erweiterbarkeit in Java . . . 168
3.3.8 Verteilte Entwicklung . . . 169
3.3.9 Betriebssysteme . . . 172
3.4 Diskussion . . . 174
3.4.1 Gegenüberstellung der Plugin-Systeme . . . 174
3.4.2 Beziehungsarten . . . 181
3.4.3 Plugin-Systeme und Softwareentwicklungstechniken . . . 186

**4 Konzeptionelle Modellierung 191**
4.1 Anforderungen und Definitionen . . . 191
4.1.1 Dynamische Änderbarkeit . . . 192
4.1.2 Einfache Rekonfiguration . . . 195
4.1.3 Erweiterbarkeit an vorgesehener Stelle . . . 196
4.1.4 Kapselung . . . 203
4.1.5 Lebenszyklen . . . 205
4.1.6 Mehrere Verwaltungsebenen . . . 206
4.1.7 Nebenläufigkeit und Threadsicherheit . . . 207
4.1.8 Visuelle Modellierung mit präziser Semantik . . . 209
4.1.9 Definitionen . . . 210
4.2 Modellierungstechnik . . . 219
4.2.1 Wahl der Modellierungssprache . . . 220
4.2.2 Modellierung durch fortschreitende Verfeinerung . . . 221
4.2.3 Implizite Nebenläufigkeit . . . 222
4.2.4 Wert- oder Referenzsemantik . . . 224
4.2.5 Schnittstellendefinition und Aufrufrichtung . . . 225
4.2.6 Grundlegende Abstraktionen . . . 233
4.2.7 Verzahnung von Systemstruktur und Lebenszyklen . . . 236
4.3 Schrittweise Modellierung des Plugin-Konzepts . . . 239
4.3.1 Abstraktes Grundmodell . . . 240
4.3.2 Präzisiertes Modell der Erweiterbarkeit . . . 241
4.3.3 Unabhängige Erweiterbarkeit . . . 244
4.3.4 Rekursion in der Erweiterungsbeziehung . . . 245
4.3.5 Nicht-hierarchische Strukturen . . . 250
4.3.6 Zentralisierung der Erweiterungsverwaltung . . . 252

4.3.7 Konfiguration in einem Schritt . . . 254
4.3.8 Kommunikation und Koordination . . . 257
4.4 Anwendbarkeit des Modells auf existierende Plugin-Systeme . . . 268
4.4.1 Beziehungsarten . . . 268
4.4.2 Lebenszyklus und Konfiguration . . . 278
4.4.3 Abgleich mit den untersuchten Systemen . . . 289
4.5 Diskussion . . . 293
4.5.1 Eigenschaften des konzeptionellen Modells . . . 293
4.5.2 Eigenschaften der untersuchten Plugin-Systeme . . . 294
4.5.3 Einordnung als Softwareentwicklungsansatz . . . 297
4.5.4 Einordnung als Architekturbeschreibungssprache . . . 300

**5 Prototypische Umsetzung 303**
5.1 Fallstudie Renew . . . 303
5.1.1 Motivation . . . 304
5.1.2 Ausgangssituation . . . 305
5.1.3 Vorhandene Erweiterungsmöglichkeiten . . . 307
5.1.4 Dekomposition . . . 309
5.1.5 Neue Flexibilität . . . 313
5.2 Erster Prototyp: Java . . . 315
5.2.1 Zielsetzung . . . 315
5.2.2 Erweiterbarkeit durch Entwurfsmuster . . . 316
5.2.3 Laden von Klassen . . . 320
5.2.4 Laden von Netzen . . . 323
5.2.5 Plugin-Manager . . . 323
5.3 Zweiter Prototyp: Netze . . . 327
5.3.1 Zielsetzung . . . 327
5.3.2 Netze . . . 328
5.3.3 Integration mit dem Java-Prototypen . . . 333
5.4 Dritter Prototyp: Java-OSGi . . . 338
5.4.1 Zielsetzung . . . 339
5.4.2 Erweiterungsbeziehungen in OSGi . . . 340
5.4.3 Simulations-Bundles . . . 342
5.4.4 Unterschiede zum Renew-PMS . . . 346
5.4.5 Eclipse-Integration . . . 349
5.5 Diskussion . . . 351

**6 Schluss 357**
6.1 Zusammenfassung . . . 357
6.2 Ergebnisse . . . 363
6.3 Ausblick . . . 366

**Abkürzungsverzeichnis 371**

**Literaturverzeichnis 375**

# Abbildungsverzeichnis


2.1 Das Nikolausbeispiel als S/T-Netz . . . . . 19
2.2 Das Nikolausbeispiel ohne expliziten Kontrollfluss . . . . . 20
2.3 Das Nikolausbeispiel als gefärbtes Netz . . . . . 21
2.4 Eine mögliche Dekomposition des Nikolausbeispiels . . . . . 26
2.5 Das Nikolausbeispiel im *Netze-in-Netzen*-Paradigma . . . . . 27
2.6 Erweiterte Kantenarten . . . . . 30
2.7 Das Nikolausbeispiel als System aus Referenznetzen . . . . . 32
2.8 Der Erreichbarkeitsgraph zum Netzsystem aus Abbildung 2.7 . . . . . 33
2.9 Die Netze des Nikolausbeispiels im Werkzeug Renew . . . . . 40
2.10 Simulation des Nikolausbeispiels im Werkzeug Renew . . . . . 42
2.11 Beispiel für ein UML-Komponentendiagramm . . . . . 66
2.12 Thematische Einteilung der FIPA-Spezifikationen . . . . . 92
2.13 FIPA-Agenten-Lebenszyklus als Petrinetz . . . . . 94
2.14 Überblick über die Mulan-Architektur . . . . . 98

3.1 Der Lebenszyklus von Netscape-Plugins als Petrinetz . . . . . 118
3.2 Der vervollständigte Lebenszyklus von Netscape-Plugins . . . . . 119
3.3 Das Lebenszyklus-Netz von XPCOM-Komponenten . . . . . 125
3.4 Das Lebenszyklus-Netz von Chrome-Paketen . . . . . 128
3.5 Das Lebenszyklus-Netz der XPInstall/Toolkit-Softwareverwaltung . . 132
3.6 Das XPInstall/Toolkit-Netz mit Steuerung der anderen Lebenszyklen . 133
3.7 Der Lebenszyklus eines klassischen Eclipse-Plugins als Petrinetz . . . . 141
3.8 Der Lebenszyklus eines OSGi-Bundles . . . . . 142
3.9 Der Lebenszyklus eines OSGi-Bundles als Petrinetz . . . . . 144
3.10 Vervollständigte Version des OSGi-Lebenszyklus als Petrinetz . . . . . 145
3.11 Lebenszyklus von OSGi-Diensten . . . . . 147
3.12 Lebenszyklen von deklarativen OSGi-Diensten . . . . . 149
3.13 Vervollständigter Lebenszyklus einer Dienstkomponente . . . . . 150
3.14 Der Lebenszyklus von NetBeans-Modulen als Petrinetz . . . . . 159
3.15 Der vervollständigte Lebenszyklus von NetBeans-Modulen . . . . . 160

4.1 Einordnung von Erweiterungs- und Dienstbeziehung . . . . . 202
4.2 Vermeidung von Inkonsistenzen bei Reentranz bzw. Nebenläufigkeit . 223
4.3 Innere und äußere Schnittstellen eines Netzes . . . . . 228
4.4 Verzahnung von Zustands- und Strukturmodellierung (Beispiel) . . . . 238
4.5 Grundmodell eines dynamisch erweiterbaren Systems . . . . . 240
4.6 Modell der Erweiterungsbeziehung . . . . . 242
4.7 Mehrere Dimensionen unabhängiger Erweiterbarkeit . . . . . 244

4.8 Erstes Modell der rekursiven Erweiterungsbeziehung . . . . . 246
4.9 Kanalkette zur Konfiguration innerer Erweiterungen . . . . . 247
4.10 Rekursive Konfiguration über drei Ebenen . . . . . 248
4.11 Vereinheitlichte rekursive Konfiguration . . . . . 249
4.12 Gegenüberstellung der geschachtelten und der Referenzdarstellung . . 251
4.13 Beispiel für eine nicht-hierarchische rekursive Erweiterungsbeziehung . 252
4.14 Plugin-Management-System (PMS) mit einstufiger Kanalkette . . . . 253
4.15 Atomare Konfiguration durch das PMS . . . . . 255
4.16 PMS mit Dienstverzeichnis . . . . . 258
4.17 PMS mit Dienstverzeichnis und atomarer Konfiguration . . . . . 259
4.18 PMS mit einfachem Lebenszyklus . . . . . 261
4.19 PMS mit komplexerem Lebenszyklus . . . . . 262
4.20 Erweiterbares System mit PMS und selbstkonfigurierender Erweiterung 265
4.21 Einfaches Modell der Modulbeziehung . . . . . 271
4.22 Extern kontrollierte Objektbeziehung (MagicBeans) . . . . . 276
4.23 Lebenszyklusnetz für *eine* Systemkomponente . . . . . 279
4.24 Lebenszyklusnetz für *mehrere* Systemkomponenten . . . . . 280
4.25 Operationale Vervollständigung des Netzes aus Abbildung 4.24 . . . . 281
4.26 Plugin-Lebenszyklusnetze im Überblick . . . . . 284

5.1 Schematischer Überblick der Renew-Architektur in Version 1.6 . . . . 306
5.2 Renew-Package-Abhängigkeiten in Version 1.6 . . . . . 307
5.3 Grundsätzliche Dekompositionsrichtung in Renew 2.0 . . . . . 310
5.4 Renew-Plugin-Dekomposition in Version 2.0 . . . . . 310
5.5 Beziehungen zwischen zentralen Renew-Plugins in Version 2.0 . . . . 312
5.6 Konfigurationsmöglichkeiten in Renew 2.0 . . . . . 313
5.7 Die Delegationshierarchie der Classloader im Renew-Plugin-System . . 321
5.8 Mögliche Weiterentwicklung der Classloader im Renew-Plugin-System 322
5.9 Lebensyklus von Plugins im Java-Prototypen . . . . . 324
5.10 Lebenszyklus unter atomarer Sichtweise . . . . . 325
5.11 Das Plugin-Verwaltungssystem des Netz-Prototypen (`PMS.rnw`) . . . . 328
5.12 Ein erweiterbarer Nikolausbetrieb (`santa.rnw`) . . . . . 329
5.13 Ein Spielzeughersteller-Plugin (`trainMakerPlugin.rnw`) . . . . . 331
5.14 Die Interaktionsmöglichkeiten des Anwenders (`system.rnw`) . . . . . 332
5.15 Das Plugin-System in Ausführung . . . . . 334
5.16 Integriertes Plugin-Verwaltungsnetz (`PMS.rnw`) . . . . . 335
5.17 Beispiel-Plugin zur Erweiterung der Renew-GUI (`FAPlugin.rnw`) . . 336
5.18 Stub-Definition zum Plugin aus Abbildung 5.17 (`FAPlugin.stub`) . . . 336
5.19 Steuerungs-/Interaktionsnetz zum integrierten Prototypen (`UI.rnw`) . 337
5.20 Der kombinierte Prototyp in Ausführung mit integriertem FAPlugin . 338
5.21 Dienstbeziehungen im OSGi-Prototypen . . . . . 343

# Tabellenverzeichnis

2.1 Klassifizierung von Komponentenmodellen nach Kompositionsmöglichkeiten, ergänzt um Plugin-Systeme . . . . . 56
2.2 Beispiele für das Registry-Muster in den untersuchten Plugin-Systemen 79
2.3 Gegenüberstellung der diskutierten Ansätze zur Softwarearchitektur . 104

3.1 Sämtliche Funktionen der NPAPI . . . . . 116
3.2 Gegenüberstellung der vorgestellten Plugin-Systeme . . . . . 179
3.3 Eigenschaften der Beziehungsarten . . . . . 185

4.1 Anforderungserfüllung durch Systeme und konzeptionelles Modell . . . 297

5.1 Anforderungserfüllung durch Prototypen . . . . . 355

# Danksagung

Die vorliegende Dissertation ist zwar in erster Linie meine Arbeit, also die Arbeit eines einzelnen Autoren. Dennoch wäre die Arbeit in der vorliegenden Form nicht ohne die vielen hilfreichen Menschen um mich herum zustande gekommen, denen ich hiermit danken möchte.

Zunächst einmal gilt mein Dank meinen Betreuern Prof. Dr. Rüdiger Valk und Dr. Daniel Moldt. Prof. Dr. Valk gilt zudem der Dank für meine Aufnahme in einen anregenden Arbeitsbereich, in welchem es mir möglich war, frei zu forschen und nicht nur auf rein theoretischen Bahnen denken zu müssen. Ich danke Dr. Daniel Moldt zudem für die intensiven und ausführlichen Gespräche, welche die von mir in dieser Arbeit eingenommene Sichtweise geprägt haben. Er hat es mir in diesem Gesprächen auch immer wieder ermöglicht, zum Thema eine angemessene Distanz aufzubauen, den Blick auf das Ziel wieder freizuräumen und auch persönliche Klippen zu umschiffen. Ebenfalls geht mein Dank an Prof. Dr. Kees van Hee, welcher als dritter Gutachter dieser Arbeit eine faire, konstruktive und hilfreiche Kritik gegeben hat. Prof. Dr. Heinz Züllighoven sei in seiner Rolle als als Prüfungsausschussvorsitzender dafür gedankt, das Prüfungsverfahren zügig und unkompliziert über die Bühne gebracht zu haben.

Über diese „offiziellen“ Personen hinaus möchte ich auch den Kollegen, Mit-Doktoranden, Freunden und Studenten danken, welche sich mir als Diskussionspartner zur Verfügung gestellt, Ideen prototypisch umgesetzt oder Teile der Arbeit korrekturgelesen und so zu dem Gelingen dieser Arbeit beigetragen haben. Ich danke Jesco von Voss, Jan Ortmann, Joachim Sauer und ganz besonders Christina Theilmann für die hilfreichen und ausführlichen Kommentare zur Rohfassung des Manuskripts dieser Arbeit. Darüber hinaus geht besonderer Dank an Lawrence Cabac für die intensiven Diskussionen zur Modellierung, welche wesentlich für die Entwicklung und Schärfung der in dieser Arbeit präsentierten Konzepte waren, sowie für die Implementierung von Plugins, welche das Renew-Plugin-System ausreizen, und von vielen kleinen Features, welche die Bedienbarkeit und Wartbarkeit des Plugin-Systems enorm gesteigert haben. Außerdem danke ich ihm für seinen unermüdlichen Antrieb zur Veröffentlichung von Ergebnissen auf Workshops und Konferenzen. Außerdem danke ich Christine Reese für klärende Diskussionen zu Grundbegriffen und ihre hilfreichen Tipps zur Verwendung von LaTeX und zugehörigen Werkzeugen.

Den unzähligen Urhebern und Zuträgern zu den von mir beim Erstellen dieses Textes genutzten Werkzeugen wie z. B. CVS, SVN, LaTeX, Ghostscript, (X)Emacs sowie den LaTeX-Paketen Hyperref, Beamer (und anderen), den Emacs-Paketen AucTeX, RefTeX und JDEE (und anderen) und Mozilla-Firefox gebührt ebenfalls großer Dank, da sie mich auf angenehme Weise unterstützt haben. Ebenfalls sei den Programmierern der während der Renew-Entwicklung genutzten Entwicklungsumgebungen und Werkzeuge wie z. B. JDEE, Eclipse, Apache Ant, JavaCC und Jalopy gedankt (eine

vollständige Liste der zur Renew-Entwicklung verwendeten Werkzeuge und Bibliotheken findet sich in der Renew-Quellcode-Distribution).

Olaf Kummer und Frank Wienberg als den Urhebern des Petrinetzeditors Renew möchte ich ganz besonders danken, da mir Renew sowohl als Zeichen- und Modellierungswerkzeug als auch als softwaretechnisches Anschauungsobjekt für die Konzepte dieser Arbeit sehr gute Dienste geleistet hat. Ich danke in gleicher Weise den Studenten Jörn Schumacher, Klaus Mitreiter und Felix Simmendinger für die erste Umsetzung von diskutierten Konzepten in Form von implementierten Renew-Prototypen im Rahmen ihrer Diplomarbeiten. Ohne ihre Beiträge wäre ich mit dem Entwurf und der Implementierung der Renew-Architektur nicht bis zum heutigen Stand gekommen. Dank gilt weiterhin Sven Offermann sowie Benjamin Schleinzer für ihre Arbeiten an grundlegenden Bibliotheken und diversen Renew-Plugins, welche die Flexibilität der Anwendung erweitert haben.

Den studentischen Teilnehmern und Teilnehmerinnen der TGI-Lehreprojekte, welche die Funktionalität von Renew- und Mulan durch kleine Ergänzungen, Fehlermeldungen und -korrekturen oder durch eigene Plugins erweitert und so zum Reifeprozess der Plugin-Architektur beigetragen haben, sei ebenfalls gedankt: Hannes Ahrens, Eugene Brin, Timo Carl, Friedrich Delgado-Friedrichs, Endri Deliu, Till Dörges, Michael Haustermann, Frank Heitmann, Marcin Hewelt, Thomas Jacob, Frank Jodeit, Maxim Klenski, Julian Knocke, Till Kothe, Wojciech Laka, Alexander Lehning, Nikolas Lohmann, Jan Ortmann, Florian Plähn, Dimitri Popov, Tobias Rathjen, Jan Schlüter, Valentin Seegert, Volker Tell, Benjamin Teuber, Anatolij Vilner und Thomas Wagner.

Dank gilt auch den Teilnehmern des jüngsten Lehreprojektes, welches nach Abschluss der schriftlichen Dissertationsleistung an der Migration von Renew auf die OSGi-Architektur mitgewirkt haben: Simon Adameit, Tobias Betz, Mathias Deutz, Dominic Dibbern, Matthias Güttler, Jan Kienecker, David Mosteller, Julian Mosteller, Eva Müller, Daniel Raap, Sebastian Schulz, Frederick-Alexander Thommsen und Tri Tran Duong. Ich danke außerdem Matthias Wester-Ebbinghaus für die partnerschaftliche Organisation des Projektes und die Unterstützung bei der Fortführung der OSGi-Umstellung.

Ich danke auch meinen Eltern, welche mich auf dem Weg zum und durch das Studium unterstützt und bestärkt haben und mir auch während des Schreibens dieser Dissertation Entlastung verschafft haben.

Und ich danke ganz besonders meiner Frau und meinen Kindern, die während der Anfertigung dieser Schrift viel zu viele Abende und Wochenenden auf mich verzichten mussten. Danke, Silke, für die Freiräume, die du mir immer wieder verschafft hast.

Hamburg, im Juni 2010
Michael Duvigneau

# 1 Einleitung

Heutige Software wird zunehmend komplexer. Immer häufiger werden ehemalige Einzellösungen zu gemeinsamen Anwendungen zusammengefasst, um zum einen den Anwendern den Wechsel zwischen verschiedenen Arbeitsumgebungen zu ersparen und zum anderen Informationsverluste beim Übergang an den Systemgrenzen innerhalb von heterogenen Softwareumgebungen zu vermeiden.

Beispielsweise sollte ein Dokumentenanzeigeprogramm, wie z. B. ein Webbrowser, möglichst alle anzutreffenden Datenformate – seien es Fotos, Grafiken, Tabellen, Texte oder Töne – richtig wiedergeben. Auch mit gemischten Daten sollte das Programm idealerweise umzugehen wissen.

Eine „Integrierte Entwicklungsumgebung“ (*Integrated Development Environment, IDE*) soll dem Entwickler sowohl beim Modellieren, beim Programmieren als auch bei der Fehlersuche zur Laufzeit unterstützen. Da aber immer häufiger Programmteile in verschiedenen Programmiersprachen erstellt werden, sollte eine IDE möglichst viele verbreitete Sprachen unterstützen.

Ein Text- oder Bildbearbeitungsprogramm soll auch nach seiner Fertigstellung noch um zusätzliche Funktionalität wie neue grafische Filter und visuelle Effekte, Rechtschreibprüfung in Fremdsprachen, Datenbanken für wiederverwendbare Elemente etc. erweitert werden können. So kann jeder Anwender seinen individuellen Bedarf durch Zusatzmodule erfüllt bekommen, die vielleicht auch von einem anderen Hersteller als dem der eigentlichen Anwendung stammen oder zum Zeitpunkt der Veröffentlichung noch gar nicht existierten.

In Unternehmen soll möglichst ein einziges System alle Funktionen, Daten und Abläufe vorhalten und bewältigen. Dabei müssen für verschiedene Anwendungsfälle wie Finanzbuchhaltung, Lagerverwaltung, Auftragsvergabe, Sachbearbeitung, Marketing etc. grundverschiedene Sichten auf die Daten, Abläufe und Funktionen zur Verfügung gestellt werden.

Bei den genannten Beispielen handelt es sich um Szenarien für den Einsatz komplexer Softwaresysteme. Die Entwickler solcher Systeme bewältigen die Komplexität häufig, indem sie das Gesamtsystem in einzelne *Systemkomponenten*[1] zerlegen. Nachdem die Schnittstellen zwischen den Systemkomponenten abgesteckt sind, können sich dann kleinere Entwicklergruppen jeweils um eine Systemkomponente kümmern und diese idealerweise weitgehend unbeeinflusst vom Entwicklungsprozess anderer Systemkomponenten fertig stellen. Zur Dekomposition des Gesamtsystems in einzelne System-

[1] Die naheliegende Bezeichnung *Komponente* für explizit abgegrenzte Teile eines Systems ist bereits durch die Disziplin der komponentenbasierten Softwareentwicklung (siehe Abschnitt 2.2) mit einer Bedeutung belegt, welche strengere Kriterien definiert, als an dieser Stelle gewünscht sind. Der hier stattdessen eingeführte Begriff der *Systemkomponente* betont eine allgemeinere Sicht auf Teile eines Systems, da er als Oberbegriff für Module, Komponenten, Objekte, Plugins, Agenten oder andere ein Softwaresystem gliedernde Konzepte stehen kann.

komponenten werden zentrale Konzepte identifiziert oder Architekturstile festgelegt, welche als Metaphern dienen, um die resultierende Softwarearchitektur zu prägen.

Längst als architekturprägendes Konzept etabliert ist das *Objekt* bzw. die jedem Objekt zugrunde liegende *Klasse* aus dem Paradigma der objektorientierten Softwareentwicklung. Dieses Konzept impliziert jedoch eine sehr feine Granularität, welche sich nicht zur umfassenden Gestaltung komplexer Softwarearchitekturen eignet. Insbesondere in der letzten Dekade sind daher andere architekturprägende Konzepte in den Fokus der Forschung gerückt, die zum einen eine gröbere Granularität in der Dekomposition erlauben, zum anderen aber auch spezielle Eigenschaften der Systemkomponenten betonen. Unter anderen sind hier die Paradigmen der komponentenbasierten und der agentenorientierten Softwareentwicklung sowie die Techniken der Architekturbeschreibungssprachen oder der Entwurfsmuster zu nennen. Für diese Arbeit soll jedoch der vor allem in der Praxis weit verbreitete Ansatz der Plugin-Systeme betrachtet werden.

Ein Ziel des *komponentenbasierten* Vorgehens ist, durch eine geschickte Abgrenzung der Komponenten deren Wiederverwendung in verschiedenen Strängen einer Produktfamilie oder auch in ganz anderen Softwaresystemen zu ermöglichen. Die Erstellung eines Softwaresystems endet dann mit einem letzten Schritt, in welchem die Komponenten zu einem Gesamtsystem zusammengefügt (montiert) und an den Kunden ausgeliefert werden. *Architekturbeschreibungssprachen* dienen der Überblicksdarstellung der Komposition von Systemen aus Komponenten, welche über verschieden charakterisierbare Arten von Verbindern miteinander kommunizieren. Die *agentenorientierte* Softwareentwicklung betont hingegen die Autonomie und Flexibilität der Agenten, welche dynamisch Beziehungen zu anderen Agenten eingehen und wieder auflösen können. Einen ganz anderen Ansatz verfolgt die Technik der *Entwurfsmuster*, welche in einer Systemarchitektur wiederkehrende Konstellationen aus Objekten identifizieren, so dass ähnlich gelagerte Situationen mit demselben Konstellationsmuster modelliert oder implementiert werden können.

Die Technik der *Plugin-Systeme* ist interessanterweise in der Praxis stark verbreitet, ohne eine entsprechende Beachtung in der Forschung zu finden.[2] Die Metapher *Plugin* überträgt die in der physischen Welt oft ausgenutzte Möglichkeit, Zusatzgeräte an eine Schnittstelle des Basisgeräts anstöpseln (*to plug in*) zu können, auf Softwaresysteme. In einer Anwendung mit Plugin-Schnittstelle ist die Kernfunktionalität jederzeit vom Nutzer der Anwendung durch Plugins erweiterbar, jedoch nur im Rahmen der Möglichkeiten, welche von der Kernanwendung vorgesehen werden.

Der Bedarf für solche Systeme ist an den eingangs genannten Beispielen gut zu erkennen. Allen einleitend genannten Beispielen ist eben auch gemein, dass eigentlich nie *ein* Anwender die gesamte in der Software integrierte Funktionalität benötigt. Will man aber *alle* Anwender zufriedenstellen, muss die gesamte Palette an Funktionen, Daten und Abläufen in der Software enthalten sein. Dadurch wächst das Softwaresystem auf eine Größe an, die es aus Sicht jedes einzelnen Anwenders im Vergleich zu den spezialisierten Einzellösungen zu unhandlich, zu langsam, zu unübersichtlich oder zu unspezifisch werden lässt.

[2] Zur Behandlung von Plugins in der Forschung siehe nächsten Abschnitt.

In einem Plugin-System werden die verschiedenen Aufgaben und Funktionen der gewünschten Anwendung ebenfalls in Systemkomponenten aufgeteilt. Eine gewisse Kernfunktionalität wird in der eigentlichen Anwendung durch ein oder mehrere Systemkomponenten bereitgestellt. Sämtliche Erweiterungen, Spezialisierungen oder Anpassungen werden in Form von optionalen Systemkomponenten – Plugins genannt – zur Verfügung gestellt. Wichtig hierbei ist, dass die Kernkomponenten einen konkreten technischen und fachlichen Rahmen für mögliche Erweiterungen definieren. So entsteht ein konfigurierbares Gesamtsystem, dessen Leistungsumfang sich den Bedürfnissen anpassen lässt. Indem sich jeder Anwender auf die Installation jener Plugins beschränken kann, die er benötigt, erhält er ein spezialisiertes, relativ schlankes System. Trotzdem steht mit der Gesamtheit aller Plugins das komplexe System zur Verfügung.

Chatley identifiziert in seiner Dissertation [Cha05, Abschnitt 1.1, S. 3ff.] vier Gründe für den Einsatz von Plugin-Systemen:

- den Wunsch, die Funktionalität eines Systems nach seiner Fertigstellung noch erweitern zu können,
- die Dekomposition großer Softwaresysteme, so dass nur die in der jeweiligen Situation benötigten Teile installiert bzw. geladen werden,
- das Aktualisieren von lang laufenden Anwendungen, ohne sie anhalten und neu starten zu müssen, oder
- der Wunsch, Erweiterungen von Drittanbietern ohne direkte Zusammenarbeit mit den Systementwicklern integrieren zu können.

Andere Autoren nennen weitere Gründe oder Varianten dieser Gründe mit unterschiedlichen Schwerpunkten wie z. B.:

- die Reduktion der Komplexität von Softwaresystemen [MMS03, S. 89],
- die Reduktion von Laufzeiten und Hardwareanforderungen von Softwaresystemen [MMS03, S. 89],
- die einfache Anpassbarkeit eines Softwaresystems an unterschiedliche Laufzeitumgebungen [Fow03, S. 499ff.],
- die einfache Integrierbarkeit nutzerspezifischer Wünsche [Bir05, S. 41], oder
- das Verkürzen der Zeit bis zur Auslieferung von Teilen eines Softwaresystems [Mar06, S. 306].

Zusammengenommen soll also ein Plugin-System die Flexibilität eines Softwaresystems bei möglichst gleichzeitiger Reduktion seiner Komplexität und seines Ressourcenbedarfs erhöhen.

Die eingangs genannten Beispiele bringen jeweils einen oder mehrere dieser Gründe mit: Im Webbrowser sollen proprietäre Medienformate von Drittanbietern darstellbar sein, wobei weder der Webbrowser-Hersteller noch der Hersteller der als Plugin integrierten Multimedia-Präsentationssoftware ihren Quellcode gegenseitig offenlegen wollen. Wenn neue Datenformate nach Fertigstellung des Webbrowsers Verbreitung

finden, soll der Browser nachträglich um die Darstellung des Formates erweiterbar sein. In der IDE greift vor allem die Idee der Dekomposition, da kaum ein Softwareentwickler sämtliche Funktionen aller unterstützten Programmiersprachen nutzen wird. Aber auch die nachträgliche Erweiterbarkeit und Plugins von Drittanbietern sind ein Thema, wenn zusätzliche Programmiersprachen oder Werkzeuge integriert werden. In der Text- oder Bildbearbeitungsanwendung steht gerade die Ergänzung von Funktionalität im Zentrum, welche zum Auslieferungszeitpunkt noch nicht existent ist oder von Drittanbietern bereitgestellt wird, um nutzerspezifische Wünsche zu erfüllen. In der allumfassenden Unternehmens-Software sind möglichst kurze bzw. eine völlige Vermeidung von Ausfallzeiten bei Konfigurationsänderungen anzustreben, so dass die Aktualisierung im Betrieb wünschenswert wird. Zudem gilt auch hier, das nicht jedes Unternehmen denselben Funktionsumfang benötigt und sich der Bedarf jedes Unternehmens mit der Zeit ändert.

## 1.1 Problemstellung

In der Praxis existieren viele Anwendungen, die sich mit dem Attribut der Erweiterbarkeit durch Plugins schmücken. Der Grad an Flexibilität, den eine auf Plugins aufbauende Anwendung bietet, kann variieren. Während manche Anwendungen bei Programmstart die in der Installation vorgefundenen Plugins einbinden und dann mit exakt diesem Leistungsumfang bis zum Programmende arbeiten, bieten andere Anwendungen zusätzliche, einfache Konfigurationsmöglichkeiten, um die Funktionalität von Plugins weiter an die Bedürfnisse des Anwenders anzupassen. Mitunter kann der Anwender installierte Plugins kurzzeitig deaktivieren, oder es wird in der Anwendung die Installation weiterer Plugins angeboten. Die Möglichkeit, Plugins während der Laufzeit zur Anwendung hinzuzufügen und wieder daraus zu entfernen, wird selten angeboten. Es lässt sich aber in jüngerer Zeit ein Trend ausmachen, eine solche dynamische Rekonfigurierbarkeit häufiger vorzusehen, um den Anwendern mehr Konfigurationskomfort zu bieten. Eine dynamische Rekonfigurierbarkeit stellt jedoch höhere Anforderungen nicht nur an die zentral koordinierten Entwickler der Kernanwendung, sondern auch an die vielen unabhängigen Entwickler von Plugins. Damit besteht in jedem Plugin ein erhöhtes Fehlerrisiko, welches auch die Stabilität der Kernanwendung beeinträchtigen kann.

Bis auf wenige Ausnahmen sind Plugin-Schnittstellen aller erweiterbaren Anwendungen proprietär. Möchte ein Entwickler seine Funktionalität in verschiedenen Anwendungen zur Verfügung stellen, so muss er für jede Anwendung eine individuelle Plugin-Fassade implementieren. Eine Vereinheitlichung der Plugin-Schnittstellen ist im Prinzip denkbar, erfordert aber die Überwindung zahlreicher technischer Hürden – insbesondere, wenn individuelle Plugin-Schnittstellen von speziellen Fähigkeiten einer Programmiersprache, eines Betriebssystems oder anderweitigen Plattformen Gebrauch machen. Ein möglicher Zwischenschritt auf dem Weg wäre die Integration individueller Plugin-Systeme ineinander, sei es direkt oder durch die Implementierung von Adaptern. Beides erfordert aber zunächst eine klare Definition der zu integrierenden Konzepte.

Die Konzeption der in der Praxis anzutreffenden Plugin-Systeme ist aber – wie in Kapitel 3 deutlich werden wird – von technischen Gegebenheiten geprägt und lässt ein präzises Begriffsgebäude vermissen. Es werden zwar Schnittstellen – im Wesentlichen in Form von Klassen, Metadaten und Dateiformaten – zu einzelnen Plugin-Systemen definiert, aber selten findet sich eine vollständige Beschreibung des Verhaltens der Plugin-Verwaltung. Dies mag bei einer Anwendung wie z. B. einem Webbrowser, der mit genau einer Plugin-Schnittstelle für eine bestimmte Art von Erweiterungen ausgestattet ist, zu vertreten sein, da die zentrale Anwendung immer noch von nur einem Unternehmen bzw. einem Entwicklerteam betreut wird. In solchen Fällen reicht es aus, die für Entwickler von Plugins relevanten Details zu spezifizieren.

Insbesondere im Bereich der IDEs haben sich aber Softwarearchitekturen verbreitet, die mit vielen voneinander unabhängigen Erweiterungsschnittstellen für verschiedene Anwendungsteile ausgestattet sind. Szyperski spricht in diesem Zusammenhang von mehreren Dimensionen an Erweiterbarkeit bzw. zusammenfassend von „unabhängiger Erweiterbarkeit“ (*Independent Extensibility*, siehe [Szy96]). In den Plugin-basierten IDE-Rahmenwerken dürfen Plugins zusätzliche funktionalitätsspezifische Erweiterungsschnittstellen einführen, welche von der Kernanwendung völlig losgelöst sind. Eine zentrale Plugin-Verwaltung übernimmt in solchen Fällen die wiederkehrenden Aufgaben zur Integration der Erweiterungen, die in die diversen Erweiterungsschnittstellen eingeklinkt werden. Eine klare konzeptionelle Aufarbeitung ist hier dringend notwendig.

In der Forschung werden Plugin-Systeme nur von einzelnen Autoren am Rande anderer Disziplinen behandelt. Einige Autoren betrachten das Konzept des Plugins als *Entwurfsmuster* in der Softwareentwicklung (siehe Unterabschnitt 2.4.3). Andere sehen Plugin-Systeme als eine Variante der *komponentenbasierten* Systeme (siehe Unterabschnitt 2.2.5). In beiden Fällen konzentriert sich die Betrachtung auf technische Notwendigkeiten, insbesondere auf den notwendigen Mechanismus zum flexiblen Laden von Code-Modulen einer Anwendung. Diese Betrachtungsweisen werden der Komplexität der durch die Metapher *Plugin* implizierten Strukturen und insbesondere Abläufe in einer Softwarearchitektur nicht gerecht. Zwar definieren Komponentenmodelle grundlegende Konzepte und Mechanismen wie komponentenübergreifende Kommunikationsschnittstellen und einheitliche Distributionsformate, wie sie auch in Plugin-Systemen benötigt werden. Für Komposition und Konfiguration sind aber in komponentenbasierten Systemen in der Regel Software-Entwickler zuständig – anders als in Plugin-Systemen, welche durch Endanwender dynamisch konfiguriert werden. Objektorientierte Entwurfsmuster beschreiben zwar die Strukturierung bzw. Beziehungen von Systemteilen untereinander, wie auch Plugins zueinander in Beziehung stehen. Die Strukturen und Beziehungen eines Entwurfsmusters sind aber in der Regel nur im Entwurf präsent und verschwinden im Gegensatz zu separat auslieferbaren Plugins zur Laufzeit im Inneren der Softwarearchitektur.

Die in komplexen Softwaresystemen – und damit insbesondere auch dynamisch rekonfigurierbaren Plugin-Systemen – innewohnende Nebenläufigkeit wird weder von den praktischen noch von den konzeptionellen Ansätzen ausreichend beachtet. Selbst unter der Annahme, dass nur *ein* Anwender für die Konfiguration des Systems verantwortlich ist, kann durch unabhängig ausgelöste Ereignisse im System die Berücksichtigung nebenläufiger Prozesse angebracht sein. Zudem kann jedes Plugin im System einen ei-

genen, unabhängigen Kontrollfluss (Thread) einrichten, ohne dass das Gesamtsystem dies wirksam verhindern kann oder sollte. Eine klare konzeptionelle Aufarbeitung typischer Strukturen und Abläufe in einem Plugin-System, welche die systeminhärente Nebenläufigkeit respektiert und sogar unterstützt, steht bisher noch aus.

Ein Beispiel für ein komplexes Softwaresystem ist das seit mehr als zehn Jahren am Arbeitsbereich Theoretische Grundlagen der Informatik (TGI) der Universität Hamburg entwickelte Petrinetzwerkzeug Renew [KWD, siehe auch Unterabschnitt 2.1.4]. Renew steht unter einer Open-Source-Lizenz und kann von beliebigen Entwicklern um weitere Funktionalität ergänzt werden. In den Jahren seit seiner Entstehung ist das Werkzeug daher um viele Funktionen ergänzt worden.

Der folgende Überblick über einige durch die Entwicklerversion von Renew abgedeckten Anwendungsbereiche verdeutlicht zugleich die Komplexität und Vielfalt des Werkzeugs, welche die Aufspaltung des Softwaresystems in eine Menge von Plugins notwendig macht.

- Einer der beiden Schwerpunkte von Renew liegt im *Petrinetz-Simulator*, der von Olaf Kummer parallel zu seiner Dissertation [Kum02a] entwickelt wurde. Dabei handelt es sich im Kern um eine Laufzeitumgebung für objektbasierte Petrinetze.

- Der zweite Schwerpunkt von Renew stellt die von Olaf Kummer und Frank Wienberg gemeinsam entwickelte IDE für Petrinetze (insbesondere die in Unterabschnitt 2.1.3 vorgestellten *Java-Referenznetze*) dar. Die IDE umfasst einen grafischen Editor mit Syntaxprüfung bei der Eingabe, die nahtlose Integration des Petrinetz-Simulators und ein grafisch animiertes Markenspiel der ausgeführten Netze mit etlichen Inspektions- und Debugging-Werkzeugen.

- Simulator und IDE unterstützen weitere alternative Petrinetzformalismen, unter anderem die *FS-Netze* von Frank Wienberg (vgl. [Wie01, DRW06]), Netze in der Semantik des Analysewerkzeugs *Maria* [DF06] oder mit *Scheme*-Ausdrücken beschriftete Netze [DF07]. Jeder Formalismus benötigt eine andere Syntax und unterschiedliche Funktionalität zur Auswertung von Ausdrücken.

- Eine Erweiterung des Petrinetz-Simulators durch Thomas Jacob erlaubt die Verwendung desselben als *Workflow-Engine* [Jac02, JKMUN02] gemäß der Referenzarchitektur der Workflow Management Coalition (WfMC). Die Erweiterung umfasst neben der Definition von Workflows auf Basis von Netzen auch Dialoge zur Konfiguration und Definition von Personen, Rollen, Aufgaben und Formularen sowie Dialoge mit Aufgabenlisten für Angestellte. Das Workflow-System ist eine verteilte Anwendung, bei der mehrere Renew-Instanzen auf verschiedenen Rechnern an einen zentralen Petrinetz-Simulator angekoppelt werden und die Simulation sowohl beobachten als auch beeinflussen können.

- Der Autor hat in seiner Diplomarbeit [Duv02] eine anwendungsunabhängige Ebene der Mulan-Architektur – die Plattform – entlang den internationalen Standards der FIPA (Foundation for Intelligent Physical Agents, siehe [FIP])

neu implementiert. Die Implementierung besteht zum Teil aus Java-Referenznetzen und wurde daher in Renew integriert. Das Ergebnis der Arbeit wurde unter dem Namen Capa auf einem internationalen Workshop veröffentlicht (siehe [DMR03, DMR02]). Capa erlaubt den innerhalb der Mulan-Architektur implementierten Agenten die Kommunikation über Rechnergrenzen hinweg, auch heterogene Umgebungen mit anderen FIPA-konformen Agentenplattformen werden unterstützt. Darüber hinaus definiert Capa weitere Schnittstellen zur Kodierung von Nachrichten und zur Repräsentierung einer Ontologie, an denen Anwendungsentwickler anschließen können.

- Zur Unterstützung der Entwicklung von Multi-Agenten-Anwendungen innerhalb der Mulan-Architektur entstanden weitere Werkzeuge wie der Mulan Viewer zur Beobachtung von Capa-Plattformen [Car03, CD07], die Netzkomponenten zur Strukturierung von Protokollnetzen [Cab02, CMR03, CDR06b], der Mulan Sniffer zur Beobachtung und Auswertung von Agentenkonversationen [CKMR06, CD07, CK07], eine prototypische Unterstützung für Roundtrip-Engineering [Dir06, CDR06a] oder das Rollen- und Agentendefinitionswerkzeug KBE (Knowledge Base Editor) [KW07, CDD+07].[3]

Wie dieser (unvollständigen) Liste an Funktionen leicht angesehen werden kann, ist Renew selber ein komplexes Softwaresystem, das Bedürfnisse verschiedener Anwender zufriedenstellt, obwohl kaum ein Anwender alle Funktionen zugleich benötigt. Diese Funktionsvielfalt ist bis einschließlich Version 1.6 [KWD02] nur nutzbar, wenn sämtliche Code-Bestandteile zu einer einzigen, schwergewichtigen Anwendung verbunden und ausgeführt werden. Lediglich ein Parameter zur Konfiguration der grafischen Benutzungsoberfläche beim Start des Programms erlaubt dann eine Selektion eines Satzes an verfügbaren Fähigkeiten zur Laufzeit (vgl. Abschnitt 5.1).

Die Vielfalt der Einsatzmöglichkeiten und Funktionen in der Anwendung Renew machten die Einführung einer flexibleren Software-Architektur notwendig. Das Konzept des Plugin-Systems erscheint aus dem Profil der Anwendung heraus als sinnvoller Ansatz. Auf diese Weise können den Anwendern bisher fehlende Konfigurationsmöglichkeiten zur Komposition von individuellen Installationen geboten werden. Somit wird jedem Anwender die Möglichkeit gegeben, aus der angebotenen Funktionsvielfalt heraus ein schlankes, auf seinen Bedarf abgestimmtes System zu komponieren.

## 1.2 Zielsetzung

Ziel dieser Arbeit soll die begriffliche Definition, Modellierung und Implementierung zentraler Konzepte eines *Plugin-Systems* sein. Diese Konzepte sollen insbesondere die bei Veränderungen der Systemkonfiguration auftretenden Prozesse und das dynamische Verhalten der Plugin-Verwaltung mit berücksichtigen. Ferner soll der Tatsache

[3] Die erwähnten Werkzeuge zur Unterstützung der Entwicklung von Anwendungen mit Mulan sind (mit Ausnahme von Capa und des Mulan Viewers) von vorneherein als Renew-Plugins erstellt worden, da das in dieser Arbeit beschriebene prototypische Plugin-System in Renew bereits zur Verfügung stand. Diese Werkzeuge werden dennoch an dieser Stelle aufgezählt, da sie ohne das Plugin-System a) hätten in das monolithische Renew integriert werden müssen, und b) womöglich aufgrund der technischen Hürden bei der Integration gar nicht entstanden wären.

Rechnung getragen werden, dass in aktuellen Softwaresystemen auch Plugins mit eigenem, von der Kernanwendung unabhängigem Kontrollfluss eingesetzt werden. Ein besonderer Schwerpunkt ist somit auf die Problematik nebenläufiger Änderungen zu legen, da unter diesen Umständen nicht mehr gewährleistet werden kann, dass sämtliche Konfigurationsänderungsanforderungen sequentiell erfolgen.

Um dieses Ziel zu erreichen, müssen die zu erarbeitenden Konzepte wesentliche Fragen, welche sich im Kontext der Software-Architektur eines Plugin-Systems stellen, beantworten können. Dem System muss durch die Konzepte eine zu den Anforderungen eines Plugin-Systems geeignete *Struktur* gegeben werden, in welcher Aufgaben und *Zuständigkeiten* der Elemente des Systems identifiziert und die wichtigen *Abläufe* innerhalb des Systems klar beschrieben werden.

Die Konzepte und ihre Beziehungen sollen *grafisch modelliert* werden, um dem Betrachter ein *intuitives Verständnis* von Struktur und Abläufen zu erzeugen. Dennoch sollte die grafische Modellierung mit einer *präzisen Semantik* hinterlegt sein, welche die klare Beantwortung von Fragen zu Eigenschaften der beschriebenen Software-Architektur zulässt. Idealerweise sollten sich die Modelle auf Basis dieser Semantik bruchlos in eine gleichwertige Implementierung überführen lassen. Eine restlose Formalisierung des Konzepts wird jedoch nicht angestrebt. Im Interesse einer einfachen, klaren Formalisierung müsste das Konzept zu stark eingeschränkt werden – unter Berücksichtigung aller Details und Eventualitäten geriete die Formalisierung zu komplex und hätte kaum den praxisnahen, nachvollziehbaren und Konzept-klärenden Charakter, wie er für das Ziel dieser Arbeit gewünscht ist.

Die Modelle der Konzepte sollen eine geeignete Balance zwischen notwendiger *Konkretheit*, um eben solche Fragen beantworten zu können, und wünschenswerter *Abstraktheit*, um auf möglichst viele Plugin-Systemen anwendbar zu sein, wahren. Insbesondere müssen die modellierten Konzepte in der Lage sein, das einleitend erwähnte Spektrum an Plugin-Systemen abzudecken und Ähnlichkeiten zu benachbarten Forschungsdisziplinen aufzuzeigen. Idealerweise sollten die Konzepte zudem auf eine Weise modelliert werden, welche zu einem hohen Wiedererkennungswert der Konzepte in den modellierten Systemen führt.

Die soeben genannten Anforderungen an die zu erarbeitenden Konzepte weisen eine Überlappung mit den Anforderungen an Entwurfsmuster auf (siehe Abschnitt 2.4). Die Anforderungen gehen jedoch darüber hinaus, da jedes Entwurfsmuster üblicherweise nur einen kleinen Teil der Softwarearchitektur erfasst. Hier wird dagegen die konzeptionelle Aufarbeitung eines Konzepts verlangt, welches die Softwarearchitektur in großen Teilen beeinflusst.

Des Weiteren liegt eine Besonderheit der Zielsetzung dieser Arbeit darin, mit einer grafischen Darstellung Struktur *und* Abläufe mit einer präzisen Semantik anschaulich zu visualisieren. Dabei ist insbesondere die in komplexen Softwaresystemen auftretende *Nebenläufigkeit* in der grafischen Modellierung von Struktur und Abläufen zu berücksichtigen. Das Modell sollte weder das Vorhandensein von Nebenläufigkeit ignorieren noch die durch Nebenläufigkeit entstehenden Konsistenzprobleme durch totalen Ausschluss nebenläufiger Handlungen in zentralen Systemkomponenten zu vermeiden suchen. Stattdessen sollte die systeminhärente Nebenläufigkeit so weit als möglich respektiert werden und erhalten bleiben.

Ein Nebenziel der Arbeit stellt die Aufarbeitung des *Standes der Technik* von Plugin-Systemen in Praxis und Forschung dar. Diese Aufarbeitung ist nötig, um eine Vereinheitlichung verwendeter Begriffe vor der Definition und Modellierung der Konzepte zu erreichen. Nur so ist es möglich, die Konzepte überhaupt identifizieren und zur Beschreibung der Systemstruktur und -abläufe einsetzen zu können.

Ebenfalls Ziel der Arbeit ist die Flexibilisierung der Softwarearchitektur der Anwendung Renew. So kann die Anwendung als Fallstudie zur Erprobung von *Prototypen* dienen. Indem *Prototypen* von Plugin-Systemen auf Basis der Konzepte implementiert werden, kann die Tauglichkeit der gefundenen Konzepte nachgewiesen und die konzeptionellen Ergebnisse zurück in die Praxis transferiert werden.

Mindestens einer der Prototypen soll sich zudem als Grundlage der modernisierten Renew-Architektur für den Einsatz in der Praxis eignen, so dass er mit einem Renew-Release auslieferbar ist.

Das Ergebnis der Arbeit umfasst also mehre Ebenen:

- *konzeptionell* werden Begriffe, Strukturen und Abläufe in Plugin-Systemen erfasst und sowohl untereinander als auch mit anderen bekannten Paradigmen und Techniken der Softwareentwicklung in Beziehung gesetzt,
- *praktisch* wird durch Prototypen die Eignung der gefundenen Konzepte überprüft, und
- als *Anwendung* entsteht ein Plugin-System für das Werkzeug Renew.

Die Ergebnisse der konzeptionellen Ebene lassen sich *allgemein* in der Diskussion, dem Entwurf und der Implementierung von Plugin-Systemen verwenden. Auf der praktischen und der angewandten Ebene werden die konzeptionellen Ergebnisse anhand des Einsatzes in Renew veranschaulicht und die Komplexität von Renew handhabbar gestaltet.

## 1.3 Aufbau der Arbeit

Im den folgenden Absätzen werden die Themen in der Reihenfolge des Ganges der Untersuchung vorgestellt. Im Anschluss wird der Zusammenhang einzelner Themen noch einmal im Kontext ihrer Querbezüge durch die Arbeit erläutert.

Zunächst werden im zweiten Kapitel Grundlagen zum Entwurf und zur Modellierung von Softwarearchitekturen für dynamische Systeme erarbeitet. Für diese Arbeit relevant sind im Wesentlichen die bereits in Abschnitt 1.1 aufgezählten Paradigmen und Ansätze zur Softwareentwicklung, welche über die elementaren Techniken der objektorientierten Softwareentwicklung hinausgehen. Hierzu zählen die Paradigmen der komponentenbasierten und der agentenorientierten Softwareentwicklung ebenso wie die Ansätze der Architekturbeschreibungssprachen, der Entwurfsmuster und der Referenznetze. Für die spätere Modellierung der Konzepte von Plugin-Systemen ist insbesondere die Technik der Referenznetze von Interesse. Es werden daher Querverbindungen zu den meisten der anderen vorgestellten Ansätze aufgezeigt. In diesem Kontext wird auch die Anwendung Renew vom Blickwinkel der Verwendung als Werkzeug

heraus vorgestellt (siehe Unterabschnitt 2.1.4). Das Kapitel schließt in Abschnitt 2.6 mit einer vergleichenden Diskussion der vorgestellten Ansätze und Paradigmen.

Das dritte Kapitel widmet sich der Betrachtung einiger existierender Plugin-Systeme, um die Bandbreite der in der Praxis vorkommenden Varianten derartiger Software-Architekturen aufzuzeigen. Im Gang der Untersuchung wird zunächst die in den jeweiligen Systemen vorhandene Begriffsvielfalt beibehalten. Ein besonderes Augenmerk liegt auf den in den Systemen meist nur implizit definierten Lebenszyklen zur Integration und Aktivierung von Plugins. Diese Plugin-Lebenszyklen werden mittels Petrinetzen formalisiert.

In der Untersuchung nehmen mit der Mozilla-Anwendungsfamilie und der Eclipse-Plattform zwei große Open-Source-Softwaresysteme einen breiten Raum ein. Beide beinhalten jeweils aus ihrer Evolution heraus mehrere verschiedene Architekturen zur Integration von Plugins. Die Untersuchung wird durch die Betrachtung weiterer Plugin-Systeme aus Forschung und Praxis abgerundet. Außerdem werden einige im Kontext flexibler Softwarearchitekturen auftretende technische Fragestellungen erwähnt, welche in separaten Forschungsdisziplinen behandelt werden und im Rahmen dieser Arbeit nur am Rande besprochen werden sollen.

Das dritte Kapitel schließt wie das vorige mit einer vergleichenden Diskussion der in den betrachteten Systemen vorkommenden Konzepte. In der Diskussion nehmen die zur Kopplung der Plugins untereinander verwendeten Beziehungsarten einen besonderen Raum ein. Die Diskussion schließt zudem einen Vergleich mit den am Ende des zweiten Kapitels diskutierten Konzepten zur Softwareentwicklung ein. In der Praxis werden viele verschiedene Begriffe zur Bezeichnung von Plugins verwendet, deren Vielfalt bei der Untersuchung und Diskussion zunächst erhalten bleibt. Eine vereinheitlichende Definition des Begriffs erfolgt erst im folgenden Kapitel.

Im vierten Kapitel findet die eigentliche konzeptionelle Aufarbeitung der zuvor betrachteten Plugin-Konzepte statt. Zu diesem Zweck werden in Abschnitt 4.1 zunächst aus den zuvor beobachteten Systemen Anforderungen an das konzeptionelle Modell von Plugin-Systemen abgeleitet und zentrale Konzepte definiert. In diesem Kontext wird auch der Plugin-Begriff gegen andere architekturprägende Konzepte abgegrenzt. Zur Vorbereitung der präzisen grafischen Modellierung der Konzepte wird in Abschnitt 4.2 die auf Referenznetzen basierende Modellierungstechnik erläutert. Hierbei wird insbesondere auf Unterschiede zur herkömmlichen Modellierung von objektorientierten Softwarearchitekturen eingegangen.

Das Modell des Plugin-Konzepts wird in Abschnitt 4.3 in mehreren Schritten entwickelt. In Abschnitt 4.4 schließt eine Untersuchung an, welche die Abdeckung der am Ende des vorigen Kapitels diskutierten Plugin-Systeme und insbesondere der identifizierten Beziehungsarten zum Thema hat. Die Berücksichtigung von Plugin-Lebenszyklen stellt einen weiteren Schwerpunkt der Untersuchung dar. Auch hier findet ein Abgleich mit den Lebenszyklen der zuvor untersuchten, existierenden Plugin-Systeme statt. Das Kapitel wird in Abschnitt 4.5 durch eine Diskussion abgeschlossen. Diese hat zum einen die Erfüllung der zu Beginn des Kapitels aufgestellten Anforderungen zum Thema. Zum anderen werden die Eigenschaften des konzeptionellen Modells in die Untersuchungsergebnisse des zweiten und dritten Kapitels eingeordnet.

Thema des fünften Kapitels ist schließlich die prototypische Umsetzung des zuvor erarbeiteten konzeptionellen Modells. Zu diesem Zweck wird zunächst die als Fall-

studie dienende Anwendung Renew aus dem Blickwinkel der Softwarearchitektur heraus betrachtet. Im Kontext der vorliegenden Arbeit sind drei Prototypen entstanden, welche auf verschiedenen Programmiersprachen und Rahmenwerken aufbauen. Die Vorstellung der Prototypen wird in Abschnitt 5.5 durch eine vergleichende Diskussion abgerundet. Die Eigenschaften der Prototypen werden zudem mit den zuvor ermittelten Eigenschaften des konzeptionellen Modells abgeglichen.

Das sechste Kapitel schließt die Arbeit mit einer Zusammenfassung des Vorgehens und der Ergebnisse sowie mit einem Ausblick auf Anknüpfungspunkte für weitere Arbeiten ab.

Im Folgenden werden drei Kernthemen der Arbeit noch einmal mit ihren Querbezügen durch die Kapitel der Arbeit im Zusammenhang vorgestellt. Bei den Themen handelt es sich um die reine konzeptionelle Modellierung, die Vergleichbarkeit zwischen existierenden Plugin-Systemen und Modellierung sowie die prototypische Anwendung der modellierten Konzepte auf Renew.

*Konzeptionelle Modellierung:* Um direkt in die Definition und Modellierung der zentralen Plugin-Konzepte einzusteigen, die in Kapitel 4 behandelt werden, sind zunächst die zentralen Anforderungen und Definitionen in Abschnitt 4.1 zu erarbeiten. Zu den dabei verwendeten Begriffen und Konzepten finden sich im Text ausreichend Rückverweise auf frühere Abschnitte der Arbeit, welche die entsprechenden Themen behandeln. Anschließend kann die eigentliche Modellierung der Plugin-Konzepte in Abschnitt 4.3 nachvollzogen werden. Da die Modellierung auf den grafischen Formalismus der Referenznetze zurückgreift, kann es notwendig sein, gelegentlich den Rückverweisen auf die Erläuterung der Eigenschaften der Modellierungstechnik in Abschnitt 4.2 oder gar auf die Einführung in die Grundlagen der Modellierung mit Petrinetzen in Abschnitt 2.1 zu folgen.

Das nach der Lektüre der Anforderungen, Definitionen und Modelle vorhandene Wissen reicht aus, um den Teil der bewertenden Diskussion in Unterabschnitt 4.5.1 nachzuvollziehen, der sich mit der Erfüllung der aufgestellten Anforderungen durch das konzeptionelle Modell beschäftigt.

*Vergleichbarkeit:* Um der Diskussion folgen zu können, in wie weit das konzeptionelle Modell auf existierende Plugin-Systeme anwendbar und mit einigen Paradigmen und Techniken aus dem Gebiet der Softwareentwicklung vergleichbar ist, sind grundlegende Kenntnisse der in den vorangegangenen Kapiteln untersuchten Vergleichssysteme, -paradigmen und -ansätze erforderlich. Vor dem Einstieg in Kapitel 4 bieten die abschließenden Diskussionen der vorangehenden Kapitel (also Abschnitt 2.6 und Abschnitt 3.4) einen groben Überblick über die Eigenschaften und prägenden Konzepte der vorgestellten Ansätze und Systeme, unterstützt durch einige in den Diskussionen zusammengetragene tabellarische Aufstellungen. Diese Diskussionsabschnitte und Tabellen bieten Rückverweise auf konkrete Abschnitte des jeweiligen Kapitels, welche einzelne Ansätze oder Systeme behandeln, so dass diese bei weitergehendem Interesse leicht aufgefunden werden können.

Im Anschluss an die Abschnitte zur konzeptionellen Modellierung (siehe oben) ist nun die Diskussion der Modellierbarkeit bestimmter Eigenschaften und Lebenszyklusmodelle konkreter existierender Plugin-Systeme in Abschnitt 4.4 nachvollziehbar. Abschließend werden die konzeptionellen Ergebnisse in Abschnitt 4.5 zu den existieren-

den Plugin-Systemen und vorgestellten Softwareentwicklungsansätzen in Beziehung gesetzt.

*Prototypische Anwendung auf Renew:* Um nachvollziehen zu können, wie sich die Anwendung des modellierten Konzepts auf die in dieser Arbeit als Fallstudie herangezogene Anwendung Renew auswirkt, sollte zunächst die in Abschnitt 2.1 vorgestellte Anwendungsdomäne der Petrinetzmodellierung bekannt sein. In diesem Abschnitt wird ein auch in der prototypischen Implementierung wieder aufgegriffenes Beispiel zur Modellierung mit Petrinetzen eingeführt. Die Vorstellung des Softwaresystems Renew aus Anwendersicht ist ebenfalls Teil des Abschnitts (siehe Unterabschnitt 2.1.4).

Im Anschluss an die Abschnitte zur Definition und Modellierung der Konzepte von Plugin-Systemen (siehe oben) kann dann in Kapitel 5 die prototypische Implementierung der Konzepte behandelt werden. In Abschnitt 5.1 wird das System Renew ergänzend aus einem softwaretechnischen Blickwinkel heraus betrachtet und die notwendigen softwaretechnischen Vorarbeiten zur Einführung eines Plugin-Systems erläutert. Anschließend werden in den aufeinander aufbauenden Abschnitten 5.2–5.4 die drei prototypischen Herangehensweisen vorgestellt. Der letzte der drei Prototypen erfordert allerdings Vorkenntnisse zum OSGi-Rahmenwerk, so dass Rückverweise zur Untersuchung der ebenfalls auf OSGi aufbauenden Eclipse-Plattform in Abschnitt 3.2 nötig sind. Die in Abschnitt 5.5 anschließende Diskussion der Eigenschaften der Prototypen setzt im Wesentlichen auf der Diskussion der Eigenschaften des konzeptionellen Modells in Unterabschnitt 4.5.1 auf.

# 2 Grundlagen

Plugin-Systeme werden mit dem Ziel eingesetzt, die Komplexität von Softwaresystemen mittels einer flexiblen, strukturierenden, dynamischen und erweiterbaren Softwarearchitektur handhabbar zu gestalten. Dieses Ziel verfolgen neben den Plugin-Systemen auch andere Paradigmen und Techniken zur Entwicklung von Software. In diesem Kapitel werden einige dieser Ansätze vorgestellt, die entweder Gemeinsamkeiten mit dem Plugin-Konzept aufweisen oder sich beim Entwurf der Softwarearchitektur von Plugin-Systemen anwenden lassen.

Zunächst einmal sollen drei Paradigmen der Softwareentwicklung vorgestellt werden. Softwareentwicklungsparadigmen verwenden im Grundprinzip dieselben Techniken, um die Komplexität einer zu entwickelnden Anwendung während Spezifikation, Modellierung, Implementierung, Verkauf und Wartung zu reduzieren. Allerdings orientieren sich die Paradigmen an unterschiedlichen Konzepten, um die Softwarearchitektur eines Systems übergreifend zu prägen. Die architekturprägenden Konzepte eines Paradigmas dienen als Metapher, anhand derer Richtlinien zur Anwendung der Techniken in diesem Paradigma abgeleitet werden.

Booch identifiziert in [Boo94] drei grundlegende Techniken, die von Softwareentwicklungsparadigmen zur Komplexitätsbewältigung eingesetzt werden. Weiß und Jakob ergänzen in [WJ04, S. 9] die Liste der Techniken in Anlehnung an McClure [McC97] um einen vierten Punkt *Wiederverwendung*:

**Dekomposition** Zerlegen des Problems in mehrere kleinere, einfacher zu handhabendere Teile, die weitgehend unabhängig von den anderen Teilen entwickelt werden können.

**Abstraktion** Definition eines Modells, das wichtige Aspekte des Systems betont und unwichtige ausblendet.

**Organisation**[1] Identifizieren von Beziehungen und Wechselwirkungen zwischen den Systemkomponenten.

**Wiederverwendung** Bewahrung bzw. Aufbereitung von Ergebnissen (Code, Dokumente, Architekturmuster etc.), um sie später wieder einzusetzen.[2]

[1] Booch spricht in [Boo94] anstelle von *Organisation* von *Hierarchie*. Hier wird aber in Anlehnung an [Jen00, S. 282] der allgemeinere Begriff *Organisation* verwendet, da Booch den Begriff *Hierarchie* über wiederholt auftretende Muster [vgl. Boo94, Abschnitt 1.3] definiert und dabei eine *Klassenhierarchie* mitsamt aller dort üblicherweise modellierten Beziehungen (Vererbung, Aggregation, Nutzung, Instantiierung oder Metaklassen) meint [vgl. Boo94, Kapitel 3]. Weiß und Jakob verwenden in [WJ04, S. 9] stattdessen den Begriff *Strukturierung*. Da der Begriff *Strukturierung* auch als Dekomposition verstanden werden kann, soll er hier vermieden werden.

[2] Die *Wiederverwendung* unterscheidet sich von den anderen drei Techniken insofern, als dass sie zwingend mehr als ein Softwareprojekt betrifft. McClure weist in [McC97, Abschnitt 2.2.3] darauf

Allen vier Techniken ist das Ziel gemein, die Komplexität eines Systems für den Entwickler zu reduzieren – sei es durch Beschränkung des zu bearbeitenden Bereichs, Ausblenden von Details, Kategorisierung von Beziehungen oder Rückgriff auf bekannte Lösungen.

Die im Folgenden vorgestellten Softwareentwicklungsparadigmen bauen auf verschiedene architekturprägende Konzepte auf. Sie leiten daraus verschiedene Vorgaben und Richtlinien ab, nach denen dekomponiert, abstrahiert und organisiert wird. Die Konzepte und Richtlinien stehen teils in Konkurrenz, können sich aber auch ergänzen. Letzteres ist möglich, falls sie zueinander orthogonale Sichtweisen einnehmen oder auf unterschiedlichen Granularitätsebenen eingesetzt werden.

Das Paradigma der *objektorientierten Softwareentwicklung* (*Object-oriented Software Engineering, OOSE*) wird in Forschung und Praxis in den letzten zwei Jahrzehnten immer selbstverständlicher eingesetzt. Als zentrales, architekturprägendes Konzept dient in der OOSE das Objekt. Seine Eigenschaften werden herangezogen, um die Anwendung der Techniken zur Komplexitätsbewältigung zu steuern. Die Dekomposition eines Systems in Objekte erlaubt es, Daten und zugehörige Operationen in einem Objekt zu kapseln und Objekte nach Typen zu klassifizieren. Zudem werden Klassen von Objekten mit einheitlichem Code ausgestattet, der von allen gleich klassifizierten Objekten trotz ihrer individuellen Identitäten wiederverwendet wird. Die OOSE wird in dieser Arbeit nicht explizit vorgestellt, da sie das derzeit marktdominante Softwareentwicklungsparadigma ist und daher allen Lesern hinlänglich bekannt sein sollte.

Stattdessen wird in Abschnitt 2.1 mit den Referenznetzen ein objektbasierter Petrinetzformalismus vorgestellt, der im Rahmen dieser Arbeit zur Modellierung der Konzepte von Plugin-Systemen eingesetzt werden wird. Die Vorstellung schließt eine Diskussion der Fähigkeiten dieser Modellierungs- und Implementierungssprache sowie des zu Grunde liegenden Paradigmas der *Netze-in-Netzen* ein. Da es sich bei Referenznetzen um einen objektbasierten Formalismus handelt, orientiert sich diese Diskussion zwangsläufig am architekturprägenden Konzept des Objekts.

Gegenstand des Abschnitts 2.2 ist das Paradigma der *komponentenbasierten Softwareentwicklung* (*Component-based Software Engineering, CBSE*). Es orientiert sich am Konzept der Komponente, die als wiederverwendbarer Baustein mit wohldefinierten öffentlichen Schnittstellen in ein System integriert werden kann. In der Praxis finden sich etliche Rahmenwerke für komponentenbasierte Systeme, die von Systementwicklern zur Komposition von Software aus Komponenten genutzt werden. Einige Autoren sehen Plugin-Systeme als Anwendungsfall komponentenbasierter Systeme.

Das in Abschnitt 2.5 vorgestellte Paradigma der *agentenorientierten Softwareentwicklung* (*Agent-oriented Software Engineering, AOSE*) orientiert sich am Konzept des autonomen, flexiblen Agenten. Die meisten agentenorientierten Systeme sind noch akademischer Herkunft und kämpfen mit den Schwierigkeiten dynamischer, offener und verteilter Systeme. Versteht man das Agentenkonzept jedoch lockerer als Metapher zur Organisation allgemeiner Softwaresysteme, so lassen sich starke Richtlinien zur Organisation dynamischer Systeme extrahieren, die auch auf Plugin-Systeme anwendbar sind.

hin, dass zwischen *Produzenten* und *Verbrauchern* von wiederverwendbaren Ergebnissen unterschieden werden muss.

Die Vorstellung der Softwareentwicklungsparadigmen wird durch zwei im Rahmen der Softwareentwicklung einsetzbare Techniken ergänzt. Diese Techniken sind relevant, da die im Rahmen dieser Arbeit erfolgende Modellierung von Plugin-Konzepten als eine Anwendung dieser Techniken gesehen werden kann.

Thema des Abschnitts 2.3 ist die Beschreibung von Softwarearchitekturen, welche im Speziellen der Gegenstand von Architekturbeschreibungssprachen (*Architecture Description Language, ADL*) ist. ADLs orientieren sich wie die CBSE am Konzept der Komponente, ergänzt um das Konzept des Verbinders. Sie sind eher im akademischen Umfeld anzutreffen und dienen der Formalisierung, Visualisierung und Analyse von Struktur und Verhalten eines Systems. Einige ADLs bemühen sich zudem, die dynamische Änderbarkeit einer Softwarearchitektur zu beschreiben. Architekturbeschreibungssprachen sollten sich zur Modellierung oder Spezifikation von Plugin-Systemen eignen. In dieser Arbeit wird jedoch aufgrund der besseren Darstellbarkeit dynamischer Veränderungen die Modellierung mit Referenznetzen gewählt.

Im Abschnitt 2.4 wird die Technik der Entwurfsmuster vorgestellt, die im Rahmen der OOSE häufig wiederkehrende Mikroarchitekturen aus wenigen Objekten oder Klassen in charakteristischen Konstellationen beschreiben. Die Anwendung von Entwurfsmustern ist im Rahmen der OOSE gängige Praxis. Ein objektorientiert implementiertes Plugin-System verwendet naturgemäß viele Entwurfsmuster, zudem wird das Plugin-Konzept selber von einigen Autoren als Entwurfsmuster verstanden und beschrieben.

Zum Abschluss des Kapitel erfolgt in Abschnitt 2.6 eine zusammenfassende Diskussion der vorgestellten Paradigmen und Techniken.

## 2.1 Referenznetze

Bei *Referenznetzen* handelt es sich um einen von Olaf Kummer in [Kum02a] definierten objektbasierten Petrinetz-Formalismus, welcher dem Paradigma der *Netze-in-Netzen* folgt. Das von Valk unter anderem in [Val98, Val04] präsentierte Konzept der *Netze-in-Netzen* kann unter verschiedenen Semantiken interpretiert werden, welche grob als Wert- und Referenzsemantiken eingeordnet werden können. Insbesondere die Referenzsemantik weist eine große Ähnlichkeit zum Paradigma der objektorientierten Softwareentwicklung (OOSE) auf, wenn sie wie in den Referenznetzen um die Möglichkeit der Instantiierung mehrerer Exemplare eines Netzmusters erweitert wird. Im Werkzeug Renew [KWD] besteht die Möglichkeit, Referenznetze in nahtloser Integration mit der Java-Laufzeitumgebung auszuführen, so dass mit Java-Referenznetzen objektorientiert Software entwickelt werden kann.

Da die Referenznetze in Kapitel 4 zur Modellierung der Konzepte von Plugin-Systemen verwendet werden, sollen ihre Grundlagen im Folgenden grob skizziert werden. In Unterabschnitt 2.1.1 werden die Grundlagen der Modellierung mit Petrinetzen kurz umrissen, um darauf aufbauend in Unterabschnitt 2.1.2 das Paradigma der *Netze-in-Netzen* einzuführen. Hierbei wird auch auf die Verwandtschaft zur OOSE eingegangen. Der Formalismus der Referenznetze als eine spezielle Variante der Anwendung des *Netze-in-Netzen*-Konzepts wird anschließend in Unterabschnitt 2.1.3 vorgestellt. Es werden im Wesentlichen die Ausdrucksmittel präsentiert, welche später in der Mo-

dellierung des Plugin-Konzepts zum Einsatz kommen. Abschließend wird in Unterabschnitt 2.1.4 aus Anwendersicht auf die Fähigkeiten des Werkzeugs Renew zur Modellierung und Ausführung von Referenznetzen in Verbindung mit Java-Code eingegangen.

### 2.1.1 Petrinetze

Die grundsätzliche Idee der Petrinetze wurde von Petri in [Pet62] entwickelt und seitdem von vielen Autoren in unterschiedlichen Richtungen zu diversen Petrinetzformalismen verfeinert. Eine umfassende Wiedergabe sämtlicher Varianten inklusive formaler Definitionen ist im Rahmen dieser Arbeit nicht möglich. Stattdessen sollen hier grundlegende Begrifflichkeiten so weit erläutert werden, wie sie zum Verständnis der im Rahmen dieser Arbeit verwendeten Referenznetze nötig sind. Eine detaillierte intuitive und formale Einführung findet sich z. B. in [GV03, Kapitel 2–4].

Die statische Struktur jedes Petrinetzes besteht aus *Transitionen*, *Stellen* und der *Flussrelation*. Sie wird als bipartiter Graph mit Stellen und Transitionen als Knoten und der Flussrelation als gerichteten Kanten dargestellt. Während Stellen zur Modellierung von Bedingungen, Ressourcen oder Zuständen herangezogen werden, dienen Transitionen der Modellierung von Handlungen oder Zustandsübergängen. Die Flussrelation legt in Verbindung mit einer in vielen Petrinetzformalismen definierten *Kantengewichtung* die Wirkung jeder Transitionen auf ihre benachbarten Stellen fest. Alle Stellen, von denen eine Kante zu einer Transition führt, heißen *Eingangsstellen* der Transition, zusammen bilden sie den *Vorbereich* der Transition. Die Stellen, zu denen eine Kante von der Transition aus führt, heißen *Ausgangsstellen*, sie bilden zusammen den *Nachbereich* der Transition.

Das dynamische Verhalten von Petrinetzen wird durch eine *Schaltregel* definiert. Diese nutzt die Informationen der statischen Struktur eines Netzes, um mögliche Übergänge zwischen *Markierungen* zu definieren. Eine Markierung beschreibt einen Zustand des Petrinetzes, indem allen Stellen Mengen von Marken zugeordnet werden. Beim Schalten einer Transition werden entsprechend der Flussrelation und Kantengewichtung Marken aus Eingangsstellen entfernt und/oder in Ausgangsstellen abgelegt. Zudem bestimmt die Schaltregel, welche Bedingungen eine Markierung erfüllen muss, damit eine Transition *aktiviert* ist (schalten kann). Das dynamische Verhalten eines Netzes beschreibt ausgehend von einer anzugebenden *Anfangsmarkierung* alle von dort aus möglichen Transitions-Schaltvorgänge sowie die dadurch erreichbaren Folgemarkierungen.

Die diversen Petrinetzformalismen variieren darin, ob und wie Kantengewichtungen, Markierungen und Schaltregel definiert werden. Darüber hinaus gibt es Formalismen, die die Betrachtung von Systemen aus mehreren Netzen vorsehen. Grundsätzlich verbinden sämtliche Petrinetzformalismen eine intuitive grafische Darstellung mit einer formal definierten Semantik. Das zu erwartende dynamische Verhalten eines Netzes ist vom Betrachter in der Regel schnell zu erahnen.[3] Die durch die Schaltregel gegebene

[3] Die Eigenschaft der intuitiven grafischen Darstellung besteht natürlich nur, solange der Zeichner eines Netzes eine solche bewusst angestrebt hat. Es ist in jeder grafischen Modellierungssprache problemlos möglich, durch ungeeignete Anordnung der Elemente (entsprechend der hier bildlich interpretierbaren Redewendung „Spaghetti-Code“ einer textuellen Programmiersprache) oder trick-

präzise Semantik vermeidet darüber hinaus Unschärfen in der exakten Interpretation einer Zeichnung.

Petrinetze lassen sich gut zur Modellierung bestimmter Aspekte von Systemen einsetzen. Die folgende Liste beschreibt häufige Modellierungstechniken, erhebt aber keinen Anspruch auf Vollständigkeit.

**Abläufe** Ketten aus abwechselnd verbundenen Transitionen und Stellen beschreiben Sequenzen von Aktionen, durch Verzweigungen an Stellen können Alternativen modelliert werden. Unabhängig voneinander markierte Stränge (z. B. durch Verzweigungen an Transitionen) beschreiben nebenläufiges Verhalten. Auch Schleifen sind modellierbar, erfordern aber zusätzlich die Modellierung von Bedingungen oder Ressourcen (siehe unten).

Für diese Art der Modellierung werden häufig *1-sichere* Netze verwendet, in denen auf jeder Stelle zu jedem Zeitpunkt maximal eine Marke liegt. Sobald mehrere Marken auf Stellen liegen, welche Teil einer solchen Kontrollflussmodellierung sind, entsteht wiederum Nebenläufigkeit, da Transitionen nebenläufig zu sich selber aktiviert werden können.

**Ressourcen** Stellen, die mit mehreren Transitionen verbunden sind, welche Marken erzeugen und/oder verbrauchen, eignen sich zur Modellierung von beschränkt verfügbaren Ressourcen. Die benachbarten Transitionen modellieren entsprechend das Erzeugen oder Verbrauchen dieser Ressourcen. Eine Aktivität, welche eine Ressource zwar exklusiv benötigt, aber nicht verbraucht, lässt sich durch eine Paar von verschieden gerichteten, gleich gewichteten Kanten zwischen Stelle und Transition modellieren.

Existieren in der Nachbarschaft einer Ressource mehrere verbrauchende oder reservierende Transitionen, so besteht ein Konflikt zwischen den modellierten Aktivitäten. Sie konkurrieren um die auf der Ressourcen-Stelle vorhandenen Marken. Falls immer nur Marken zur Aktivierung maximal einer der Transitionen zur Verfügung stehen, besteht zwischen diesen konkurrierenden Aktivitäten ein wechselseitiger Ausschluss.

**Bedingungen** Eine Stelle kann zur Modellierung einer Bedingung, also einer alternativ erfüllten (markierten) oder nicht erfüllten (unmarkierten) Aussage eingesetzt werden. Die Modellierung gestaltet sich ähnlich wie die Modellierung von Ressourcen (siehe oben), allerdings muss die Stelle 1-sicher sein. Solche Bedingungen lassen sich z. B. als Kriterium in Ablaufverzweigungen einsetzen. Für eine vollständige Fallunterscheidung müssen dann auch die komplementären Bedingungen modelliert und konsistent mit den bedingungsverändernden Transitionen verbunden werden.

Diese Aufzählung umfasst lediglich einige besonders naheliegende Modellierungstechniken, welche auf sämtliche Petrinetzformalismen übertragbar sind. Moldt präzisiert die Kombinierbarkeit dieser Ausdrucksmöglichkeiten in [Mol96, Kapitel 8] mit den so

reiche Kodierung des modellierten Systems (entsprechend „obfuscated Code“) die Verständlichkeit einer Zeichnung zu torpedieren.

genannten Szenariennetzen. Szenariennetze entstehen durch Kombination einfacher, 1-sicher und stellenverzweigungsfrei modellierter Abläufe, die während der Kombinationsschritte um Alternativen, Ressourcen und Schleifen ergänzt werden. So sind in den kombinierten Modellen beliebig komplexe Systeme modellierbar.

Eine grundsätzliche Eigenschaft der Modellierung mit Petrinetzen ist die regelmäßig vorhandene Nebenläufigkeit. Sie besteht zunächst zwischen sämtlichen Transitionen, ausgenommen jenen, welche über die Flussrelation direkt oder indirekt miteinander verbunden sind. Das bedeutet, dass in der Modellierung durch das Setzen von Kanten explizit Nebenläufigkeit *aus dem Modell herausgenommen* werden muss. Im Gegensatz dazu gilt in vielen anderen Modellierungs- und Implementierungssprachen der sequentielle Ablauf als Standardsituation, welcher explizit (z. B. in Form mehrerer Threads) um nebenläufige Abläufe ergänzt werden muss.

Ein einfacher Petrinetzformalismus ist der der „S/T-Netze“, welcher ununterscheidbare Marken verwendet. Da so nur die Anzahl der Marken relevant ist, kann die Gewichtung jeder Kante und die Markierung jeder Stelle durch natürliche Zahlen beschrieben werden.[4] Bereits in diesem Formalismus lassen sich viele Eigenschaften beobachten, welche für die Modellierung mit Petrinetzen typisch sind.

Eine kompakte Modellierung komplexer Systeme ist jedoch nur mit einer ausdrucksstärkeren Modellierungssprache zu erreichen. Es existieren diverse „höhere“ Petrinetzformalismen, welche unterscheidbare Marken zulassen, auf denen im Zuge einer schaltenden Transition Funktionen berechnet oder Operationen durchgeführt werden können. Prominentester Vertreter dieser Gattung an Petrinetzformalismen sind die von Jensen definierten gefärbten Netze (*Coloured Petri Nets*, CPN, siehe [Jen92]). Marken können hier in bestimmte Werte (Farben) unterschieden werden, welche zum Zwecke der Typisierung von Stellen und Kanten zu Typen (Farbmengen) zusammengefasst werden. Die Färbung der Marken erlaubt es, Stellen und Transitionen zusammenzufalten, welche gleichartige Wirkung auf verschieden gefärbte Marken haben. Zudem können algebraische Datentypen manipuliert und in Funktionen verwendet werden, um beliebige Algorithmen in CPN zu implementieren.

Höhere Netze wie z. B. die gefärbten Netze sind zwar in der Lage, beliebige Systeme vollständig zu beschreiben. Sie erreichen dieses Ziel in der Regel aber nur, indem komplexe Datentypen als Marken verwendet und mittels Operationen manipuliert werden. Den verwendeten Datentypen fehlt weitgehend eine intuitive visuelle und gleichzeitig formal präzisierte Semantik, wie sie die Netze selber bieten. Somit sind Vorteile der Modellierung mit Petrinetzen nur auf *einer* Schicht der Systemarchitektur sichtbar, da sämtliche anderen Schichten weiterhin mit Mitteln konventioneller Programmiersprachen implementiert werden müssen.

Moldt beschreibt in [Mol96] die Verwendung gefärbter Netze zur einheitlichen Modellierung eines Systems. Sowohl die funktionale[5] und die datenorientierte als auch die prozessorientierte Sicht werden als Petrinetz dargestellt und miteinander verbunden. Seine Arbeit bewegt sich sowohl im Kontext der strukturierten Analyse als auch der objektorientierten Analyse. Das Verbinden funktionaler, datenorientierter und prozess-

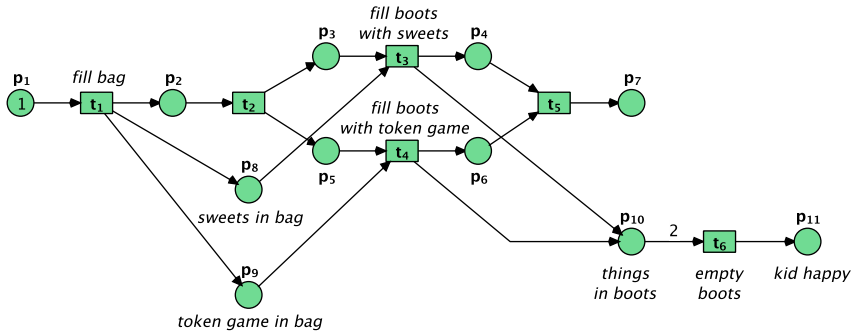
[4] In der grafischen Darstellung werden Kantengewichtungen vom Wert 1 und Stellen mit 0 Marken nicht beschriftet, da diese Situationen als Standardfall betrachtet werden.

[5] Das Adjektiv „*funktional*“ wird in weiten Teilen dieser Arbeit nicht im mathematischen Sinne, sondern in der Bedeutung „*funktionalitätsorientiert*“ verwendet.

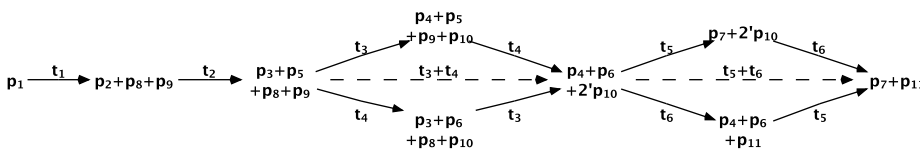
orientierter Modelle in einem Petrinetzmodell ist somit auch in der objektorientierten Softwareentwicklung anwendbar.

#### 2.1.1.1 Beispiel

Abbildung 2.1 zeigt ein S/T-Netz als Beispiel. Es modelliert den Arbeitsablauf des Nikolauses, der zu Beginn seinen Sack mit Geschenken füllt und diese dann in Stiefel verteilt. Im Beispiel ist im oberen Bereich (Transitionen $t_1$ bis $t_5$ und Stellen $p_1$ bis $p_7$) der Arbeitsablauf des Nikolauses als Kontrollfluss modelliert. Die Stellen $p_8$, $p_9$ und $p_{10}$ modellieren Ressourcen, also die im Sack verfügbaren Geschenke oder den Füllgrad der Stiefel. Die Stelle $p_{11}$ modelliert eine erfüllbare Bedingung – im Beispiel den Gemütszustand des beschenkten Kindes.




(a) Ein S/T-Netz

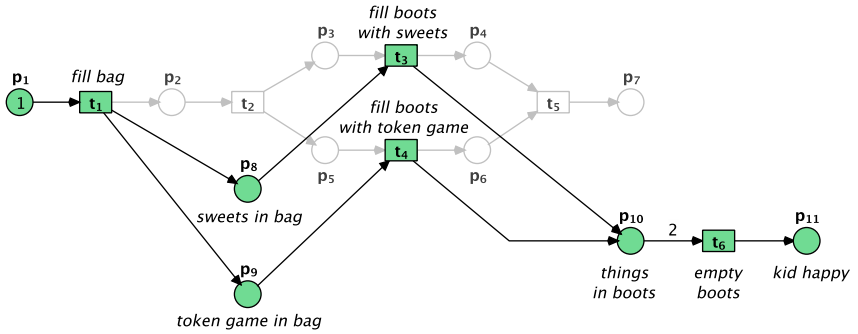



(b) Der zugehörige Erreichbarkeitsgraph

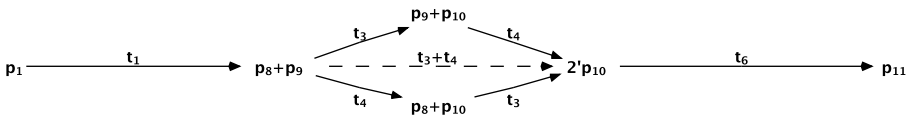
Abbildung 2.1: Das Nikolausbeispiel als S/T-Netz

Im Beispiel ist an nur zwei Orten die Nebenläufigkeit nicht beschränkt worden. Die Anfangsmarkierung umfasst genau eine Marke auf der Stelle $p_1$, so dass in jeder Kontrollflussstelle maximal eine Marke liegen kann. Zudem sind auch die Kantengewichte an den Ressourcen und Bedingungen so gewählt, dass nicht mehr Marken vorkommen können als für den gewünschten Ablauf notwendig sind. Die Transitionen $t_3$ und $t_4$ sind zueinander nebenläufig, so dass der Weihnachtsmann Markenspiel oder Süßigkeit in beliebiger Reihenfolge in die Stiefel füllen kann. Es ist aber auch möglich, dass er beide Dinge gleichzeitig aus dem Sack greift. Ein weiteres Vorkommen von Nebenläufigkeit betrifft die Transitionen $t_5$ und $t_6$. Diese Transitionen beziehen sich auf verschiedene Akteure (Nikolaus und Kind) und haben keinen kausalen Zusammenhang.

Der Erreichbarkeitsgraph des S/T-Netzes beschreibt dessen präzise Semantik. Die beiden Vorkommen von Nebenläufigkeit sind in diesem einfachen Fall an den rautenförmigen Strukturen zu erkennen. Die gestrichelten Kanten stellen die Möglichkeit dar, jeweils beide nebenläufigen Transitionen gleichzeitig zu schalten.




(a) Vereinfachtes Netz




(b) Zugehöriger Erreichbarkeitsgraph

Abbildung 2.2: Das Nikolausbeispiel ohne expliziten Kontrollfluss

Um weniger grafische Elemente im Netz haben zu müssen, kann oftmals auf eine separate Modellierung des Kontrollflusses verzichtet werden. Dies ist möglich, falls der Kontrollfluss über die Stellen, welche Bedingungen und Ressourcen modellieren, ebenfalls gewährleistet wird. Im Beispiel ist dies möglich. Das resultierende Netz ist in Abbildung 2.2 dargestellt. Wie am Erreichbarkeitsgraphen zu erkennen ist, hat das Netz für die verbliebenen Transitionen und Stellen dasselbe Verhalten.

Bei abweichender Anfangsmarkierung oder großzügigerer Ausstattung mit Ressourcen (mehrere Spielzeuge und Süßigkeiten im Sack) sind die Ressourcenstellen jedoch nicht mehr in der Lage, einen sequentiellen Kontrollfluss sicherzustellen. Im Beispiel wäre es etwa möglich, dass sich ein Kind über einen Stiefel freut, der nur Süßigkeiten oder nur Markenspiele enthielte oder gar überfüllt wäre.

In Abbildung 2.3 ist das vereinfachte Beispiel als gefärbtes Netz dargestellt. Die zuvor doppelt enthaltenen Stellen und Transitionen zur Unterscheidung der verschiedenen Geschenke sind nun nur noch einmal im Netz vorhanden. Die Unterscheidung findet durch gefärbte Marken auf den Stellen $p_8$ und $p_{10}$ statt. Die Variable `th` wird im Kontext der Transition $t_3$ wahlweise an die Süßigkeiten oder das Markenspiel gebunden, so dass zwei verschiedene Schaltvorgänge möglich sind. Diesen beiden Schaltmodi können wie im S/T-Netz nebenläufig angewendet werden, da genug unabhängige Marken auf den Eingangsstellen vorliegen. An der Transition $t_6$ ist es durch die Aus-

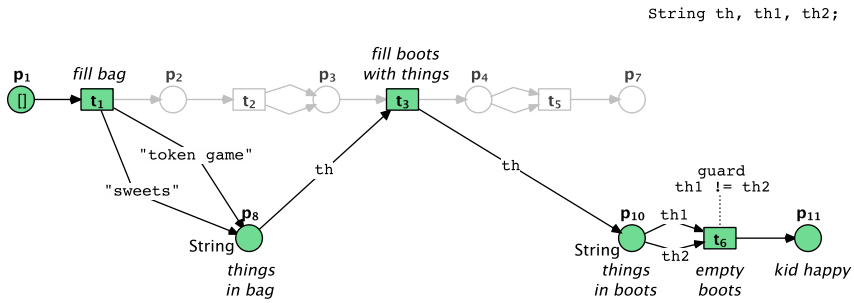



Abbildung 2.3: Das Nikolausbeispiel als gefärbtes Netz

drucksmächtigkeit gefärbter Netze nun möglich, auszudrücken, dass das beschenkte Kind sich nur freut, falls zwei verschiedene Geschenke im Stiefel waren.

### 2.1.2 Netze in Netzen

Valk hat in [Val98, Val01, Val04] das Paradigma der *Netze-in-Netzen* vorgestellt, welches Petrinetze als Marken in Petrinetzen vorsieht. Die von Valk gegebene formale Definition der elementaren Objektnetzsysteme (EOS) beschreibt eine Systemarchitektur mit genau zwei Ebenen: Auf der ersten Ebene befindet sich ein so genanntes Systemnetz und auf der zweiten Ebene alle darin enthaltenen Objektnetze. Die zu Grunde liegende konzeptionelle Idee ist jedoch auf beliebig tief verschachtelte Objektnetze anwendbar. Michael Köhler hat in [Köh04] eine weiter gehende Formalisierung von Objektnetzen betrieben.

Die von Valk und Köhler definierten Petrinetzformalismen setzen auf S/T-Netzen auf, sehen also abseits der Objektnetzmarken keine komplex strukturierten oder algebraisch definierten Marken vor. Es können beliebig viele Objektnetzmarken mit unterschiedlichen Markierungen in einem Netzsystem vorkommen. In den einfacheren Formalismen wie z. B. den EOS weisen alle Objektnetzmarken dieselbe Netzstruktur (Transitionen, Stellen und Kanten) auf, in komplexeren Varianten können auch Objektnetze von unterschiedlicher Struktur auftreten. Eine Erzeugung von Objektnetzkopien mit eindeutiger Identität ist allerdings in keinem der Formalismen als mögliche Wirkung einer Transition vorgesehen. Es können lediglich Kopien der Marken erzeugt werden, die bereits in der Anfangsmarkierung eines Netzsystems existieren.

Valk und Köhler bieten verschiedene Semantiken zur Interpretation dieser Kopien an. Unter den *Wertsemantiken* verfügt jede Kopie über eine individuelle Markierung. Es existieren mehrere verschiedene Wertsemantik-Definitionen. Sie unterscheiden sich darin, wie die Nachfolgemarkierungen von Objektnetzkopien bei der Duplikation oder bei der Zusammenführung mehrerer Kopien desselben Objektnetzes ermittelt werden. Die von Köhler in [Köh04] definierte Wertsemantik verlangt beispielsweise, dass die Summe der Marken auf jeder Stelle eines bei einem Schaltvorgang bewegten Objektnetzes über alle vor bzw. nach dem Schaltvorgang existierenden Objektnetzkopien gleich bleiben muss.

Unter der *Referenzsemantik* werden die Objektnetze während eines Schaltvorganges nicht dupliziert. Stattdessen wird jede Marke als eine Referenz auf ein Objektnetz betrachtet, so dass sich beim Erzeugen oder Konsumieren von Marken lediglich die Zahl der auf ein bestimmtes Objektnetz zeigenden Referenzen ändert. Unter der Referenzsemantik können daher keine echten Kopien eines Objektnetzes mit individueller Markierung während eines Schaltvorganges erzeugt werden. Alle Referenzen auf ein Objektnetz „sehen“ in jeder erreichbaren Markierung dieselbe Markierung dieses Objektnetzes.

Neben Wert- und Referenzsemantiken existiert noch die so genannte Prozesssemantik, in der jede Objektnetzkopie ihre individuelle Markierung um die Information des zu dieser Markierung führenden Netzprozesses anreichert. Das Zusammenführen von Objektnetzkopien im Rahmen eines Schaltvorganges ist dann nur möglich, wenn die Netzprozesse dieser Kopien vereinbar sind. Für die Modellierung im Rahmen dieser Arbeit ist diese Semantik jedoch nicht von Bedeutung und wird daher nicht weiter ausgeführt.

Ohne eine Kommunikationsmöglichkeit würden das Systemnetz und sämtliche Objektnetze eines Netzsystems vollständig unabhängig voneinander schalten. Eine essentielle Eigenschaft des *Netze-in-Netzen*-Paradigmas ist daher die Synchronisation von Transitionen über Objektnetzgrenzen hinweg. In allen in diesem Unterabschnitt beschriebenen Netzformalismen wird eine Synchronisationsrelation $\rho$ definiert. Transitionen, die in dieser Relation enthalten sind, können nicht alleine schalten, sondern nur synchron mit den jeweiligen Partnern. Die Partner müssen zudem in der aktivierenden Markierung des Netzsystems benachbart sein, um schalten zu können.

Als benachbart gelten Transitionen gemäß einer der folgenden zwei Alternativen. Bei der vertikalen Synchronisation ist eine der synchronisierten Transitionen im Systemnetz (bzw. einem „äußeren“ Objektnetz) enthalten, während die Partnertransition sich in einem Objektnetz befindet, welches als Marke an dem Schaltvorgang der äußeren Transition beteiligt ist. Die horizontale Synchronisation verlangt hingegen, dass die Partner-Transitionen zu Objektnetzen gehören, welche zum Zeitpunkt des Schaltens als Marken in derselben Stelle des Systemnetzes (bzw. eines „äußeren“ Objektnetzes) liegen. Grundsätzlich gilt weiterhin, dass jede an einem synchronen Schaltvorgang beteiligte Partner-Transition in der aktuellen Markierung ihres Objektnetzes ebenfalls aktiviert sein muss. Transitionen, die nicht in $\rho$ enthalten sind, können weiterhin nach der klassischen Schaltregel für P/T-Netze innerhalb der Markierung ihres Objektnetzes aktiviert werden und schalten.

Die Referenzsemantik ist einfacher zu definieren, da sie die Objektnetze und ihre Markierungen nicht duplizieren und wieder zusammenführen muss. Köhler zeigt in [KR04], dass sich ein Netzsystem unter Referenzsemantik immer in ein S/T-Netz mit gleichem Verhalten auffalten lässt, während dies für Wertsemantiken nicht der Fall sein muss. Zudem liegt die Referenzsemantik näher an den ebenfalls mit Referenzen arbeitenden modernen objektorientierten Programmiersprachen als die Wertsemantik. Allerdings bedeutet die Existenz mehrerer Referenzen auf ein Objektnetz in verschiedenen Stellen eines Systemnetzes (bzw. äußeren Objektnetzes), dass „Seiteneffekte“ im Systemnetz beobachtbar werden. Steht eine Transition des Systemnetzes in der Synchronisationsrelation, so kann sich ihre Aktiviertheit aufgrund der Wirkung einer Transition mit augenscheinlich disjunkter Lokalität (keine gemeinsame Stelle im

Vor- und Nachbereich) ändern. Das aus verschiedenen Stellen referenzierte Objektnetz überträgt eventuell aktivierungsrelevante Informationen zwischen diesen Stellen, da es seine Markierung ändert. Dieser Effekt stellt eine dynamische Ausweitung der Lokalität von Transitionen dar, die zwar für Petrinetze ungewohnt erscheint, aber dennoch formal wohldefiniertes Verhalten darstellt.

#### 2.1.2.1 Netze-in-Netzen und Objektorientierung

Das Paradigma der *Netze-in-Netzen* ähnelt in seinen elementaren Konzepten zur Strukturierung eines Systems stark dem der objektorientierten Softwareentwicklung. In der „reinen Lehre“ des objektorientierten Paradigmas besteht ein System ausschließlich aus gleichberechtigten Objekten, welche auf der Basis eines einheitlichen Kommunikationsmechanismus Nachrichten austauschen. Zustand und Verhalten der einzelnen Objekte sind gekapselt, also weder nach außen sichtbar noch von außen beeinflussbar. Ein Objekt kann lediglich durch eine Nachricht aufgefordert werden, eine seiner öffentlich angebotenen Operationen („Methoden“ genannt) auszuführen.

Im Paradigma der *Netze-in-Netzen* besteht ein System ausschließlich aus Netzen. Die Netze sind gleichberechtigt in der Hinsicht, dass kein Netz ein anderes Netz kontrollieren oder steuern kann. Die internen Zustandsveränderungen eines jeden Netzes sind durch die Schaltregel definiert, welche für alle Netze gleichermaßen gilt. Die interne Markierung eines jeden Netzes ist von außen nicht wahrnehmbar, also vollständig gekapselt. Als einheitlicher Kommunikationsmechanismus dient die Synchronisationsrelation $\rho$. Objekte können so die Aktiviertheit bestimmter Transitionen kommunizieren, ohne die tatsächliche Aktivierungsbedingung (die Kantengewichte im Vorbereich) offen zu legen.

Die öffentlich angebotenen Methoden eines Objekts formen seine Schnittstelle; alle Objekte mit gleicher Schnittstelle bilden eine Klasse. Klassen sind wiederum in einer Hierarchie angeordnet, welche Generalisierungs- und Spezialisierungsbeziehungen zwischen den Klassen definiert. In vielen existierenden objektorientierten Programmiersprachen (so auch in der später im Rahmen dieser Arbeit zum Einsatz kommenden Sprache Java) wird allerdings zwischen Objekt-Schnittstelle (*Interface*) bzw. -Typ und der Klasse des Objekts unterschieden. So können zu einer Schnittstelle bzw. zu einem Typ mehrere konkrete Klassen existieren, welche das durch die Schnittstelle definierte, nach außen sichtbare Verhalten auf unterschiedliche Weise intern implementieren. Die Unterscheidung von Typ und Klasse ist in der „reinen Lehre“ allerdings nicht notwendig, da sie nur einen Anwendungsfall der Spezialisierungsbeziehung darstellt.

Im *Netze-in-Netzen*-Paradigma wird keine explizite Typhierarchie definiert. Dennoch lässt sich die Gesamtheit der von einem Objektnetz verwendeten Elemente der Synchronisationsrelation $\rho$ als Schnittstellenbeschreibung auffassen. Wie weiter unten im Beispiel sichtbar wird, kann $\rho$ statt durch extensionale Aufzählung der synchronisierten Transitionspaare auch durch die Annotation von Symbolen an den zu synchronisierenden Transitionen definiert werden. Transitionen mit einem Symbol können dann nicht mehr „spontan“ schalten, sondern nur noch, wenn eine andere Transition mit demselben Symbol im Schaltvorgang benachbart ist. Die Menge der an den Transitionen eines Objektnetzes notierten Synchronisationssymbole definiert somit implizit die Schnittstelle des Objektnetzes.

Den hier unter dem *Netze-in-Netzen*-Paradigma zusammengefassten Petrinetzformalismen fehlt eine Möglichkeit zur Erzeugung neuer Instanzen eines Netzes, wie sie in objektorientierten Programmiersprachen üblich ist. Es können aber sehr wohl mehrere Objektnetze die gleiche Struktur aus Transitionen, Stellen und Kanten aufweisen. Betrachtet man nun die Struktur eines Netzes als Definition seiner Klasse, so stellen alle Objektnetzmarken, welche dieselbe Struktur aufweisen, Instanzen dieser Klasse dar. Jede dieser Instanzen verfügt über einen individuellen Zustand in Form der Markierung des jeweiligen Objektnetzes. Alle diskutierten Semantiken lassen die Betrachtung von Netzmarken als Instanzen zu: In der Referenzsemantik stellt jedes Objektnetz eine Instanz dar, auf die mehrere Referenzen zeigen können. Die Zustände der Objektnetzkopien unter den Wertsemantiken können allerdings miteinander verschmolzen werden. Um die Kapselung von Objektzuständen zu erhalten, empfiehlt es sich hier, nur solche Objektnetzkopien als unabhängige Instanzen aufzufassen, die nicht verwandt sind (also weder aus einer gemeinsamen Urkopie hervorgegangen sind noch irgendwann einmal zu einer gemeinsamen Folgekopie zusammengeführt werden können). Diese Betrachtung erfordert allerdings eine sorgfältige Konstruktion der Netzstruktur, um verwandte Kopien ermitteln zu können.

Im Vergleich zur Implementierung von Methoden in vielen verbreiteten objektorientierten Programmiersprachen bestehen bei der Synchronisation von Transitionen allerdings zwei bemerkenswerte Unterschiede. Die meisten objektorientierten Programmiersprachen sind im imperativen Programmierparadigma angesiedelt und sehen zur Implementierung einer Methode daher ein Konzept vor, dass der imperativen Prozedur gleicht. Beim Aufruf der Methode (der Prozedur) können Informationen in Form von Objekten als Methodenparameter an das aufgerufene Objekt übergeben werden. Hat das Objekt die Methode abgearbeitet, gibt es eventuell ein Ergebnis in Form eines Objekts zurück. Ein Objekt wird somit als *reaktives System* und ein Methodenaufruf als Sequenz *zweier unidirektionaler* Kommunikationsvorgänge betrachtet.

Das synchrone Schalten von Transitionen stellt dagegen nur *einen* Kommunikationsvorgang dar. Die vorgestellten Petrinetzformalismen basieren zudem auf den S/T-Netzen, so dass keine unterscheidbaren Informationen wie z. B. in gefärbten Netzen kommuniziert werden können. Die kommunizierte Information beschränkt sich daher in den besprochenen Formalismen auf die Aktiviertheit der an der Synchronisation beteiligten Transitionen. Diese Information wird allerdings *bidirektional* ausgetauscht: Die Synchronisation kann nur zustande kommen, wenn in allen beteiligten Netzen genügend Marken entsprechend der Kantengewichtungen im Vorbereich der beteiligten Transition vorhanden sind. Solange diese Bedingung nicht in allen beteiligten Netzen erfüllt ist, kann die synchronisierte Transition in keinem der Netze schalten. Jede der beteiligten Transitionen ist daher gleichermaßen für das Zustandekommen der Synchronisation (also der Kommunikation) entscheidend. Die synchrone bidirektionale Kommunikation wird in Unterabschnitt 4.2.5.1 noch einmal aufgegriffen, um ihre Auswirkung auf die Modellierung von Systemen zu klären.

Der zweite Unterschied zwischen imperativer objektorientierter Programmierung und der Modellierung im *Netze-in-Netzen*-Paradigma ist die bereits erwähnte Betrachtung von Objekten als reaktive Systeme. Objekte verfügen in der Regel über keinen eigenen Kontrollfluss (*Thread*), sondern werden nur solange aktiv, wie sie einen Methodenaufruf abarbeiten. Bei der Modellierung mit Petrinetzen kann ein solches reaktives

Verhalten eines Netzes durchaus hergestellt werden, es stellt aber keinesfalls die Regel dar. Stattdessen können in allen Objektnetzen eines Systems ebenso wie im Systemnetz aus der Anfangsmarkierung heraus schon Transitionen aktiviert sein und schalten. Objektnetze modellieren daher üblicherweise eher proaktive denn reaktive Systeme. Diese Eigenschaft wird insbesondere bei der Modellierung von Multiagentensystemen mit Petrinetzen ausgenutzt (siehe Unterabschnitt 2.5.2).

Die beiden angesprochenen Aspekte beschreiben Unterschiede zwischen dem Paradigma der *Netze-in-Netzen* und solchen objektorientierten Programmiersprachen, welche im *imperativen* Programmierparadigma verwurzelt sind. Solche Sprachen bilden das Konzept der Methode auf das Konzept des reaktiven Prozeduraufrufs ab. Die „reine Lehre“ der Objektorientierung schreibt diese Abbildung aber nicht vor, sondern lässt beliebige Kommunikationstechniken zu.

Das Paradigma der *Netze-in-Netzen* definiert keine Typhierarchie. Es nicht möglich, ein Netz explizit als Spezialisierung oder Generalisierung eines anderen Netzes zu deklarieren. Ebenfalls nicht vorgesehen ist die Möglichkeit, Teile der Netzstruktur eines Netzes zu vererben, also wiederzuverwenden, ohne die Struktur kopieren zu müssen. Zur besseren Unterscheidung von objekt*orientierten* Programmiersprachen, welche Spezialisierung, Generalisierung und Vererbung unterstützen, empfiehlt es sich, die Formalismen des *Netze-in-Netzen*-Paradigmas als objekt*basiert* zu bezeichnen.

Im Vergleich zu anderen Modellierungstechniken für dynamisches Verhalten objektorientiert entworfener Systeme (wie z. B. die in der Unified Modeling Language (UML) vertretenen Statecharts, Aktivitätsdiagramme oder Interaktionsdiagramme) bieten Petrinetze eine einfache grafische und dennoch formal präzise spezifizierte Darstellung von Abläufen inklusive Nebenläufigkeit und Synchronisation. *Netze-in-Netzen* ergänzen die Fähigkeit der hierarchischen Strukturierung, indem Objekte des Systems abgegrenzt und mit ihren Schnittstellen (Synchronisationspunkten) modelliert werden können. Die Ausdrucksmächtigkeit der *Netze-in-Netzen* im Hinblick auf objektorientierte Analyse, Modellierung und Programmierung übertrifft damit die der dynamischen Diagrammtypen von UML, da sie dynamische und statische Aspekte kombinieren. Es ist allerdings festzuhalten, dass Petrinetze nicht geeignet sind, alle statischen Diagrammtypen der UML (siehe hierzu auch Unterabschnitt 2.3.3) vollständig zu ersetzen, da sie nicht in der Lage sind, das Beziehungsgeflecht zwischen Klassen und Objekten im Überblick darzustellen. Insbesondere fehlt eine direkte Ausdrucksmöglichkeit für Vererbungsbeziehungen.

#### 2.1.2.2 Beispiel

Das in Unterabschnitt 2.1.1.1 vorgestellte Beispielnetz zum Arbeitsalltag des Nikolaus' vermengt viele Aspekte, welche im Interesse einer klaren Modellierung und einfachen Wartbarkeit des Systems getrennt betrachtet werden sollten. Neben dem eigentlichen Arbeitsablauf des Nikolauses modelliert das Netz die Funktionalität zweier Behältnisse (Sack und Stiefel) sowie den Einfluss des beschenkten Kindes, welches eine Aktion durchführt und daraufhin seinen Gemütszustand verändert. Gemäß dem Paradigma der objektorientierten Softwareentwicklung sollten diese Elemente gekapselt modelliert werden.

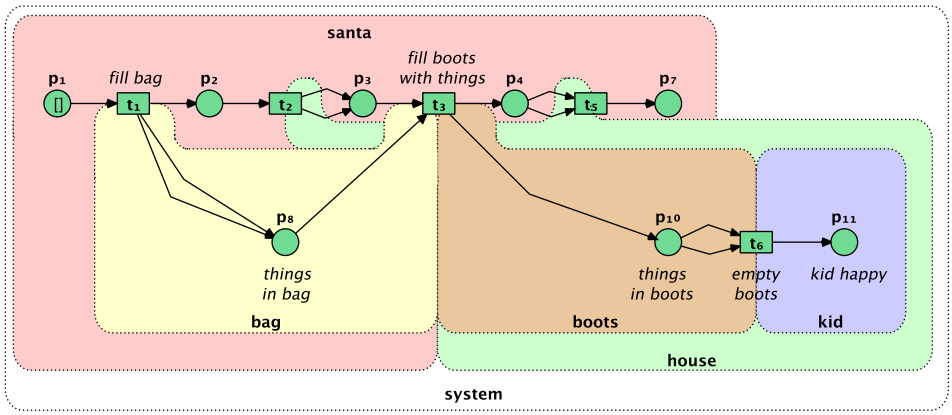



Abbildung 2.4: Eine mögliche Dekomposition des Nikolausbeispiels

In Abbildung 2.4 wird eine entsprechende Dekomposition des Beispiels skizziert. Als Grundlage dient das gefärbte Netz aus Abbildung 2.3, hier jedoch der Einfachheit halber in einer ungefärbten Variante. Die farbigen Flächen stellen die Einflussbereiche der eben genannten Aspekte des Modells dar. Sie berühren alle jene Transitionen, die einen Effekt auf den Zustand des jeweiligen Modellaspekts haben. Die gänzlich durch einen Farbbereich umschlossenen Stellen sind jeweils einem einzigen Einflussbereich zugeordnet.

Einige Einflussbereiche schließen andere vollständig ein, um eine Enthaltensein-Beziehung auszudrücken: Der Sack wird ausschließlich vom Nikolaus getragen, die Stiefel und das Kind befinden sich innerhalb eines Wohnhauses. Die Beziehung zwischen Nikolaus und Wohnhaus ist in dieser Darstellung schwierig zu erfassen, da der Nikolaus das Wohnhaus nur vorübergehend betritt – es handelt sich hierbei um eine dynamische Beziehung, welche besser im agentenorientierten Paradigma modelliert werden sollte (siehe Abschnitt 2.5).

In Abbildung 2.5 ist ein Netzsystem im *Netze-in-Netzen*-Paradigma dargestellt, wie es sich aus der eben skizzierten Dekomposition des Beispiels ergibt. Das Systemnetz modelliert zwei Orte, an denen sich die Netzmarken aufhalten können: innerhalb oder außerhalb des Wohnhauses. In der Anfangsmarkierung befindet sich eine Marke, die den Nikolaus repräsentiert, außerhalb des Wohnhauses auf der Stelle $p_{12}$. Die Marken, welche die Stiefel und das Kind repräsentieren, liegen auf der Stelle $p_{13}$ innerhalb des Hauses. Eine Marke, die den Sack repräsentiert, liegt in der Stelle $p_1$ des Objektnetzes „santa“. Bei allen angesprochenen Marken handelt es sich um Objektnetze. Um eine übersichtliche Darstellung zu erhalten, sind die Objektnetze nicht in die sie als Marken enthaltenden Stellen eingeschachtelt, sondern nebeneinander liegend dargestellt. Die Pfeile weisen vom Symbol der Marke zum jeweiligen Objektnetz.

Die Synchronisationsbeziehung $\rho$ wird in den Netzen durch die in spitze Klammern gesetzten Bezeichner angegeben. Transitionen mit einem solchen Bezeichner (im Beispiel sind alle Transitionen betroffen) können nur gleichzeitig mit den anderen Tran-

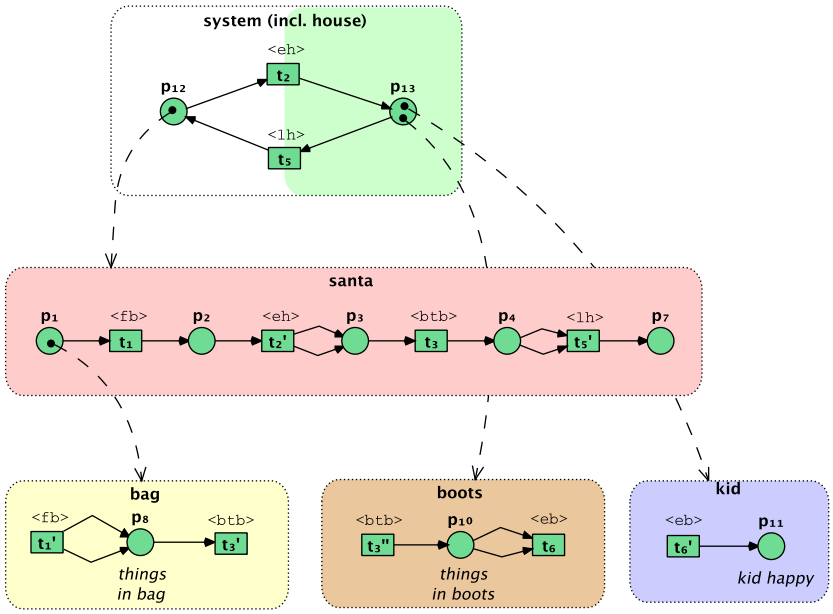



Abbildung 2.5: Das Nikolausbeispiel im *Netze-in-Netzen*-Paradigma

sitionen mit demselben Bezeichner schalten. Zudem müssen die Netze, zu denen die schaltenden Transitionen gehören, entsprechend der bereits beschriebenen Schaltregel im Moment des Schaltvorganges in einer direkten Beziehung stehen. Das anfangs aktivierte Transitionspaar $t_1 + t_1'$ mit dem Bezeichner `<fb>` (für „*fill bag*“) kann schalten, da das Netz „bag“, zu welchem die Transition $t_1'$ gehört, als Marke im Kontext der Transition $t_1$ gebunden wird (vertikale Synchronisation). Die Nachfolgemarkierung enthält dann das Objektnetz „bag“ als Marke in der Stelle $p_2$ des Netzes „santa“. Auf ähnliche Weise kann als nächstes das Transitionspaar $t_2 + t_2'$ mit dem Bezeichner `<eh>` (für „*enter house*“) schalten.

Die Synchronisationsbeziehungen mit den Bezeichnern `<btb>` (für „*bag-to-boots*“) und `<eb>` (für „*empty boots*“) nutzen hingegen die horizontale Synchronisation. Im Falle des Transitionspaares $t_6 + t_6'$ liegen die beteiligten Objektnetze „boots“ und „kid“ gleichzeitig in der Stelle $p_{13}$ des Systemnetzes. Der Bezeichner `<btb>` kombiniert sowohl horizontale als auch vertikale Synchronisation, so dass das Transitionstripel $t_3 + t_3' + t_3''$ nur schalten kann, wenn die Objektnetze „santa“ und „boots“ in der Stelle $p_{13}$ des Systemnetzes benachbart sind und das Objektnetz „bag“ als Marke im Kontext der Transition $t_3$ gebunden wird.

Im Netz „santa“ wird in den Stellen $p_3$ und $p_4$ die Interpretation des Netzsystems unter den verschiedenen Semantiken des *Netze-in-Netzen*-Paradigmas relevant, da die Netzmarke „bag“ durch die Transition $t_2'$ dupliziert wird. Das Beispiel ist allerdings so konstruiert, dass sowohl Wert- als auch Referenzsemantik die gewünschte Termination des Netzsystems erreichen. Im Falle der Referenzsemantik erzeugt $t_2'$ auf der Stelle $p_3$

zwei Marken, welche Referenzen auf dasselbe Objektnetz „bag“ darstellen. Dieses Objektnetz verfügt über zwei Marken auf der Stelle $p_8$, von der bei jedem Schaltvorgang des Transitionstripels $t_3 + t'_3 + t''_3$ eine Marke abgezogen wird. $t'_5$ kann anschließend die beiden identischen Netzreferenzen konsumieren und eine auf $p_7$ ablegen.

Im Falle der Köhlerschen Wertsemantik müssen beim Duplizieren der Netzmarken durch $t'_2$ die Marken auf $p_8$ so verteilt werden, dass jede Kopie des Objektnetzes „bag“ genau eine Marke erhält. Bei jedem Schalten des Transitionstripels mit dem Synchronisationsbezeichner `<btb>` wird dann die Stelle $p_8$ in einer der Kopien geleert. In der Stelle $p_4$ sammeln sich somit nur unmarkierte Kopien des Objektnetzes „bag“, die dann beim Schalten von $t'_5$ wieder zu einer unmarkierten Kopie zusammengeführt werden.

Die objektorientierte Dekomposition des Beispiels erleichtert es nun, einzelne Elemente des Modells zu variieren, ohne das gesamte Netz bzw. alle Objektnetze anpassen zu müssen. So könnte beispielsweise der Gemütszustand des Kindes weiter elaboriert oder die Interaktion zwischen Kind und Stiefel verfeinert werden. Es wäre auch möglich, den Arbeitsablauf des Nikolaus' zu verändern, ohne dass dies Auswirkungen auf die Schnittstellen von Sack, Stiefeln oder Wohnhaus hätte. Auch ein Ausbau des Systems auf mehrere Häuser, Stiefel und Kinder ist problemlos möglich.

### 2.1.3 Referenznetze

Kummer hat in seiner Dissertation [Kum02a] mit den so genannten *Referenznetzen* einen objektbasierten Petrinetzformalismus definiert, der die Referenzsemantik des *Netze-in-Netzen*-Paradigmas mit der Ausdrucksmächtigkeit gefärbter Netze vereint. Zudem integriert der Formalismus das Konzept der Erzeugung von Netzinstanzen auf Basis von Netzmustern während eines Schaltvorganges. Ein bidirektionaler synchroner Kommunikationsmechanismus erlaubt den Austausch beliebiger Daten zwischen Netzinstanzen. Ergänzend werden einige zusätzliche Kanten mit spezieller Semantik eingeführt. Eine intuitive Einführung in den Formalismus der Referenznetze findet sich im Handbuch dies Werkzeugs Renew [KWD06b, Abschnitt 3], eine formale Definition des Formalismus auf Basis von beschrifteten Algebren und Graphersetzungssystemen wird in [Kum02a, Kapitel 10] gegeben. Für die Zwecke dieser Arbeit genügt eine intuitive Einführung in die Variante der Java-Referenznetze, welche im Werkzeug Renew (siehe Unterabschnitt 2.1.4) modelliert und ausgeführt werden können.

Als Marken sind im Formalismus der Referenznetze (wie in den gefärbten Netzen) beliebig komplex strukturierte Daten zulässig, auf denen im Kontext eines Schaltvorganges beliebige Operationen durchgeführt werden können. Die Syntax zur Definition von Markentypen und Operationen orientiert sich an der Syntax der objektorientierten Programmiersprache Java, unterscheidet sich aber in einem wesentlichen Punkt. Während die Programmiersprache Java dem *imperativen* Programmierparadigma folgt, orientiert sich die Beschriftungssprache der Java-Referenznetze am *funktionalen* Programmierparadigma, da dieses der formal verwendeten Definition über beschriftete Algebren näher kommt. Insbesondere sind Seiteneffekte wie z. B. Variablenzuweisungen bei der Definition der Aktiviertheitsbedingung und Nachfolgemarkierung einer Transition hinderlich. Zentrales Element der Auswertung ist stattdessen eine Unifikation

auf den im Vorbereich einer Transition verfügbaren Marken, um Variablenbelegungen für einen Schaltvorgang zu ermitteln.

Grundsätzlich sind an einer Transition drei Arten von Anschriften möglich: (Algebraische) Ausdrücke können den Unifikationsprozess zum Finden möglicher Variablenbindungen beeinflussen. (Boolesche) Bedingungen werden durch das reservierte Wort `guard` eingeleitet und können unerwünschte Variablenbelegungen herausfiltern. Seiteneffekt-behaftete Operationen auf Java-Objekten müssen durch das reservierte Wort `action` geschützt werden, da sie während eines möglicherweise scheiternden Unifikationsprozesses noch nicht ausgewertet werden dürfen. Letztere werden im Rahmen dieser Arbeit während der Modellierung nicht benötigt, sind aber bei der Implementierung von ausführbaren Softwaresystemen mit relevanter Funktionalität unvermeidlich.

Da der Formalismus der Referenznetze auf dem Paradigma der *Netze-in-Netzen* aufbaut, sind auch Netze als Marken zugelassen. Der Formalismus unterscheidet zwischen Netz*mustern* und Netz*instanzen*. Ein Netzmuster definiert eine statische Struktur aus Transitionen, Stellen, Kanten und Anschriften. Zu jedem Netzmuster können entsprechend dem Konzept der Instantiierung in objektorientierten Programmiersprachen beliebig viele Netzinstanzen erzeugt werden. Somit entspricht das Netzmuster einer Klasse und die Netzinstanzen den Objekten, welche auf dieser Klasse basieren. Eine Typhierarchie mit Spezialisierung, Generalisierung oder gar Vererbung kann zwischen Netzmustern nicht definiert werden, so dass auch der Formalismus der Referenznetze als *objektbasiert* zu bezeichnen ist.

Zur Kommunikation zwischen Objekten werden synchrone Kanäle nach Christensen eingesetzt [siehe CD92]. Die synchronen Kanäle erweitern das bereits im *Netze-in-Netzen*-Paradigma vorhandene Konzept des bidirektionalen Abgleichs der Aktiviertheit von Transitionen um den bidirektionalen Austausch beliebig strukturierter Daten oder gar von Netzmarken zwischen Netzinstanzen. Im Referenznetzformalismus besteht nur die Möglichkeit der vertikalen Synchronisation, bei der eine Transition eine ihr zugängliche Netzreferenz nutzt, um sich mit einer Partnertransition in der referenzierten Netzinstanz zu synchronisieren. Die im Rahmen des *Netze-in-Netzen*-Paradigmas ebenfalls denkbare horizontale Synchronisation zwischen Netzmarken, welche in derselben Stelle eines anderen Netzes liegen, ist im Referenznetzformalismus nicht vorgesehen. Sie kann allerdings durch eine Transition im äußeren Netz, welche zwei synchrone Kanäle zu Netzmarken auf derselben Stelle koppelt, simuliert werden (siehe folgendes Beispiel).

Die in den einfacheren Formalismen schon diskutierte Technik, die Synchronisationsrelation durch Symbole an den synchronisierten Transitionen zu definieren, wird auf die synchronen Kanäle übertragen. Passende Synchronisationspartner werden über einen gemeinsamen Kanalnamen identifiziert. Zudem stellt die Stelligkeit des Kanals (also die Zahl der zum bidirektionalen Informationsaustausch eingesetzten Parameter) ein relevantes Kriterium dar.

Jeder synchrone Kanal hat genau zwei Endpunkte: An der Transition, welche die Netzreferenz zur Verfügung hat, wird ein *Downlink* notiert. Die Transition im referenzierten Netz wird mit einem passenden *Uplink* versehen. Verfügen mehrere Transitionen über gleich benannte Kanäle mit gleicher Stelligkeit, so kann nichtdeterministisch jede der möglichen Paarungen aktiviert werden – vorausgesetzt, die beteiligten Transitionen sind aktivierbar.

Eine Netzmarke wird entsprechend der in Unterabschnitt 2.1.2 vorgestellten Referenzsemantik als *Referenz* auf eine Netzinstanz interpretiert, so dass mehrere Marken dieselbe Netzinstanz referenzieren können. Wie schon bei der Referenzsemantik des *Netze-in-Netzen*-Paradigmas besprochen (siehe Unterabschnitt 2.1.2) führt diese Eigenschaft zu für Petrinetze ungewohnten Effekten, da die Aktivierung von Transitionen je nach Betrachtungsweise von Seiteneffekten abhängt bzw. die Lokalität von Transitionen sich über ihr unmittelbares grafisches Umfeld hinaus erstreckt. Da auch Netzmarken durch synchrone Kanäle in beide Richtungen kommuniziert werden können, können Netzinstanzen in einem Referenznetzsystem ein beliebig verbundenes Beziehungsgeflecht aufbauen, das auch Zyklen enthalten kann.

Um kompaktere Netzmuster mit hoher Nebenläufigkeit modellieren zu können, werden einige zusätzliche Kantenarten im Formalismus der Java-Referenznetze unterstützt. Neben den zwei klassischen Arten von Kanten, *Eingangs-* und *Ausgangskanten*, bietet der Formalismus auch *Reservierungskanten* (grafisch mit Pfeilspitzen an beiden Enden dargestellt, siehe Abbildung 2.6), *Testkanten* (grafische Darstellung ohne Pfeilspitzen) und *flexible Kanten* (mit einer doppelten Pfeilspitze notiert).

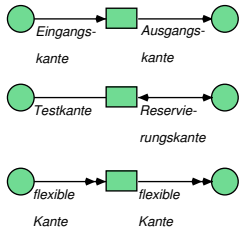



Abbildung 2.6: Erweiterte Kantenarten

Eine Eingangskante zieht eine Marke von der angebundenen Stelle ab, eine Ausgangskante legt eine Marke auf die Stelle. Die Reservierungskante ist schlichtweg eine Kurzschreibweise für die Kombination aus je einer Aus- und Eingangskante mit identischer Anschrift – es wird also eine Marke abgezogen und wieder zurückgelegt. Testkanten lassen ähnlich wie Reservierungskanten die Markierung einer Stelle unverändert, allerdings wird die Marke nicht abgezogen. Daher kann eine Marke von mehreren Testkanten gleichzeitig verwendet werden.

Die flexiblen Kanten gibt es wiederum als Ein- und Ausgangskanten. Eine flexible Kante schiebt eine Multimenge von Marken auf eine Stelle bzw. zieht sie davon ab. Die Menge kann in Form einer Liste, eines Java-Arrays oder eines Java-Collection-Objekts angegeben werden. Da diese Menge dynamisch zur Ausführungszeit berechnet werden kann, erlauben die flexiblen Kanten die Bewegung eine variablen Zahl von Marken in einem Schaltvorgang. Das Berechnen der Menge muss explizit durch andere Kanten oder Transitionsanschriften erfolgen. Flexible Kanten erlauben *nicht* wie z. B. Transferkanten das automatische Einsammeln *aller* Marken einer Stelle, da dieses Verhalten zu einer problematischen Nebenläufigkeitssemantik führen würde.[6]

[6] Das Verhalten von Transferkanten lässt sich mit flexiblen Kanten durch doppelte Buchführung simulieren: Jede Transition, welche die Markierung der Stelle vor einer flexiblen Kante verändert,

Neben den eben besprochenen Kantenarten, welche problemlos in der Modellierung mit Referenznetzen eingesetzt werden können, gibt es weitere Kanten, deren Semantik allerdings nur unter streng sequentieller Simulation eines Netzes wohldefiniert ist. Neben den bereits angesprochenen Transferkanten gehören hierzu insbesondere die Inhibitor-Kanten. Sie werden grafisch mit kleinen schwarzen Kreisen an den Enden dargestellt und erlauben das Aktivieren einer Transition nur, wenn *keine* Marken auf der mit der Inhibitorkante adjazenten Stelle liegen. In S/T-Netzen kann eine Inhibitorkante zusätzlich mit einem Kantengewicht versehen werden. Dann verhindert die Kante das Aktivieren der Transition, sobald *mindestens* so viele Marken auf der Stelle vorhanden sind, wie es dem Kantengewicht entspricht.

Als objektbasierte Java-Referenznetze eignen sich Referenznetze nicht nur zur Modellierung, sondern auch als Implementierungssprache, wie die Betrachtung des Werkzeuges Renew in Unterabschnitt 2.1.4 zeigen wird. Zwar ist für die Netze bisher keine Vererbungsbeziehung oder Typhierarchie definiert, aber die enge Integration in die objektorientierte Programmiersprache Java erlaubt eine Nachbildung solcher Hierarchien. Die ersten Erfahrungen aus der Anwendung der Java-Referenznetze als Programmierwerkzeug sind vom Autor in [Duv01] dokumentiert worden. Referenznetze eignen sich besonders zur Modellierung und Implementierung dynamischer Systeme, wie im Kapitel 4 deutlich werden wird.

Es sollte nicht unerwähnt bleiben, dass noch keine speziell auf den Formalismus der Referenznetze zugeschnittenen Analyseverfahren existieren, wie sie beispielsweise für S/T-Netze oder gefärbte Netze verwendet werden. Grundsätzlich gilt, dass die Erreichbarkeit von Markierungen im Formalismus der Referenznetze wie auch in dem der gefärbten Netze unentscheidbar ist, sofern unendlich große Farbmengen für Marken zugelassen werden. Kummer hat zudem in [Kum02a, Kapitel 4] nachgewiesen, dass schon die in Referenznetzen mögliche Instantiierung beliebig vieler unterscheidbarer Objekte (z. B. Netzinstanzen) ausreicht, um Unentscheidbarkeit zu bewirken. Dieses Ergebnis führt dazu, dass interessante Analysen nur in eingeschränkten Modellen durchgeführt werden können.

#### 2.1.3.1 Beispiel

Das Nikolausbeispiel wurde im Unterabschnitt 2.1.2.2 bereits im Paradigma der *Netze-in-Netzen* modelliert. Da der Referenznetzformalismus sich ebenfalls am Paradigma der *Netze-in-Netzen* orientiert, kann das Modell aus Abbildung 2.5 mit geringfügigen Änderungen übernommen werden. Zum ersten ist eine Anpassung notwendig, weil die horizontale Synchronisation im Referenznetzformalismus nicht enthalten ist. Diese findet im Modell zwischen Nikolaus und Stiefeln sowie zwischen Stiefeln und Kind statt, während sich die beteiligten Objektnetzmarken gleichzeitig in der Haus-Stelle aufhalten. Zum zweiten eröffnen die Konzepte des Informationsaustausches entlang

muss diese Wirkung auch auf einer zweiten Stelle in *einer* Marke nachpflegen. Diese Marke repräsentiert somit als Multimengenobjekt die Multimenge aller Marken der ersten Stelle, so dass dieses Multimengenobjekt zur Bestimmung der Wirkung der flexiblen Kante eingesetzt werden kann. Ein Nebeneffekt dieses Verfahrens ist allerdings die Sequenzialisierung sämtlicher Transitionen im Vor- und Nachbereich der doppelt buchgeführten Stelle. Ein Beispiel für die Implementierung von Transferkanten durch flexible Kanten findet sich in Abbildung 4.25 auf S. 281.

synchroner Kanäle und der Erzeugung von Instanzen eine Schärfung der Objektkapselung im Modell. So lässt sich das Modell näher an der abgebildeten Realität[7] ausrichten. Zum dritten ermöglicht die Unterscheidung von Netzmustern und -instanzen die Wiederverwendung gleichartiger Muster für unabhängige Objekte. Im Beispiel stellen sowohl Sack als auch Stiefel passive Behältnisse mit gleichartiger Funktionalität dar. Zur Steigerung der Modellqualität fallen daher die Veränderungen am Modell nicht ganz so geringfügig aus.

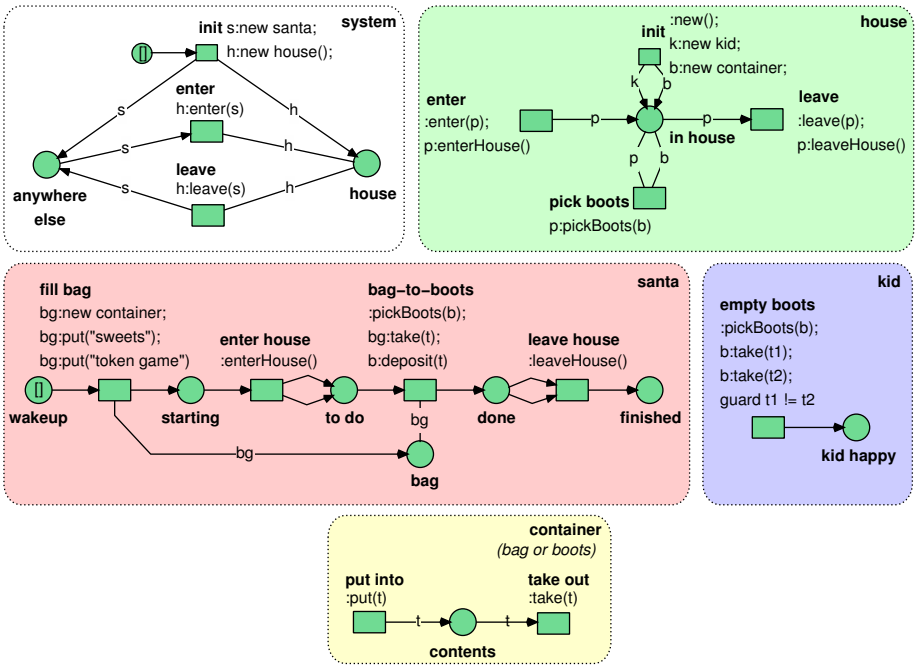



Abbildung 2.7: Das Nikolausbeispiel als System aus Referenznetzen

In Abbildung 2.7 sind alle Netzmuster des Modells dargestellt. Die abstrakten Stellen- und Transitionsbezeichner ($p_1, t_2$ usw.) werden in dieser Darstellung durch sprechende Bezeichner (in fetter Schrift dargestellt) abgelöst. Die Verwendung sprechender Bezeichner ergibt sich aus der Übertragung der aus dem Umgang mit textuellen Programmiersprachen bekannten Empfehlung, sprechende Variablennamen zu verwenden, auf die Implementierung von Softwaresystemen mit Petrinetzen.

Der dynamische Zustand des Netzsystems variiert mit jedem Schaltvorgang und wird daher nicht in der platzintensiven Überblicksdarstellung abgebildet. Ein dynamischer Zustand des Systems ist exemplarisch in Abbildung 2.10 im Rahmen der Simulation dieses Beispiels im Werkzeug Renew zu sehen. Zudem folgt in Abbildung 2.8 eine Kurznotation aller Markierungen des Netzsystems im Erreichbarkeitsgraphen.

[7] Ja, der Nikolaus wird als Teil der Realität betrachtet. Jedes Kind kann diesen Fakt bestätigen.

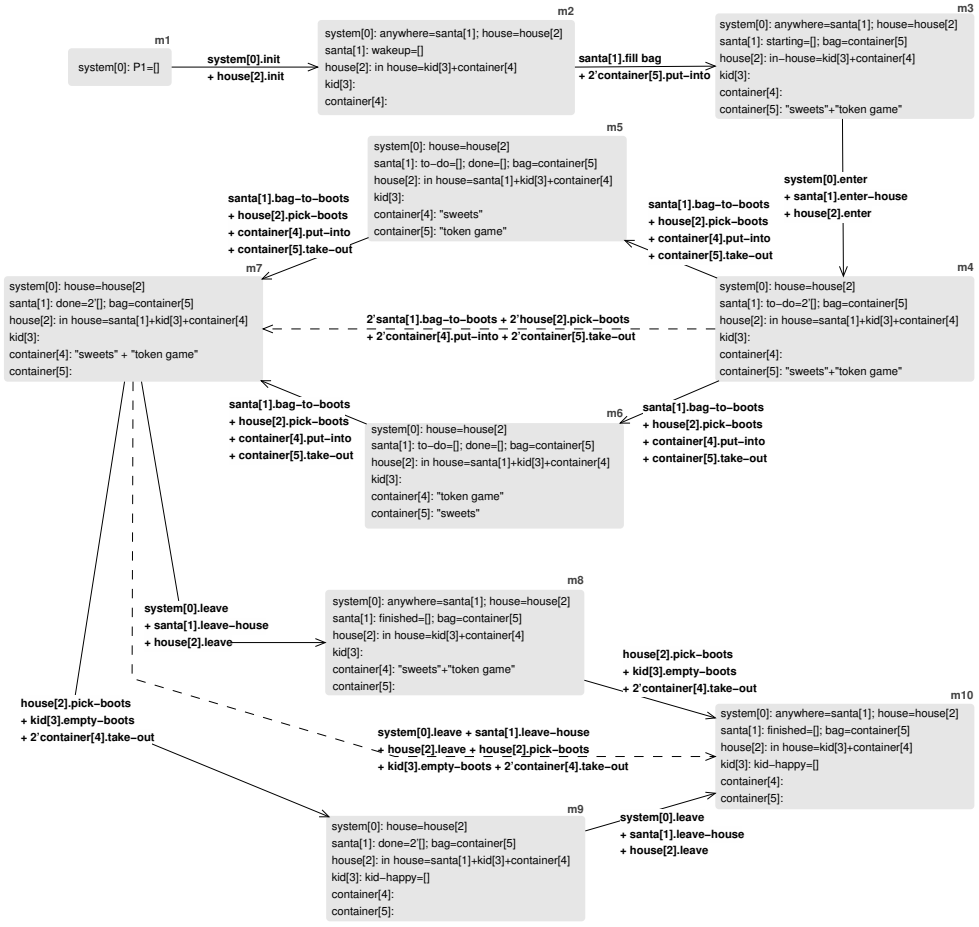



Abbildung 2.8: Der Erreichbarkeitsgraph zum Netzsystem aus Abbildung 2.7

Da im Referenznetzformalismus Netzmarken zwischen Netzinstanzen über synchrone Kanäle übergeben werden können, ist eine getrennte Modellierung des Hauses von der System-Außenwelt möglich. Der Nikolaus kann nun das Haus betreten und verlassen, indem seine Netzmarke vom Systemnetz an das Hausnetz und zurück übergeben wird. Die rechte Stelle des Netzes „system“ enthält nun während des gesamten Ablaufes immer genau eine Marke, nämlich die Netzreferenz auf eine Netzinstanz vom Muster „house“. Die linke Stelle nimmt die Netzmarke der Netzinstanz des Nikolaus' auf, falls dieser sich nicht im Haus befindet.

Die Simulation des Netzsystems startet mit einer Netzinstanz des Musters „system“. Die Anfangsmarkierung besteht aus einer anonymen („schwarzen“) Marke in der kleiner dargestellten Stelle links der Transition „init“. In der Anfangsmarkierung existieren noch keine Netzreferenzen als Marken, da die zu referenzierenden Netzinstanzen erst

durch das Schalten der Transition „init“ erzeugt werden. Dieser separate Schaltvorgang ist nötig, da die Syntax der Java-Referenznetze bisher noch keine Erzeugung von Netzinstanzen in Anschriften zur Definition der Anfangsmarkierung unterstützt. Nach dem Schalten der Transition „init“ existiert dann die eigentlich gewünschte Anfangsmarkierung dieser Netzinstanz mit einer Netzmarke auf Basis des Musters „santa“ und einer auf Basis des Musters „house“.

Die Klammern am Ende der Anschrift `h:new house()` an der Transition „init“ weisen darauf hin, dass das zu instantiierende Netzmuster über einen parameterlosen synchronen Kanal als Konstruktor verfügt. Im Netzmuster „house“ ist die Transition „init“ mit dem Uplink `:new()` versehen. Die beiden „init“-Transitionen schalten beim Instantiieren synchron. In der frisch erzeugten Netzinstanz auf Basis des Musters „house“ werden daher gleichzeitig zwei weitere Netzinstanzen auf Basis der Netzmuster „kid“ und „container“ erzeugt und in die zentrale Stelle des Netzes gelegt. Damit umfasst die Markierung des Referenznetzsystems nach dem ersten Schaltvorgang insgesamt fünf Netzinstanzen.

Wie bereits angedeutet basieren die Objekte Sack und Stiefel auf einem einheitlichen Netzmuster für Behältnisse („container“). Die zugehörigen Netzinstanzen werden zum einen im Netz „house“ durch die Anschrift `b:new container` an der Transition „init“ und zum anderen im Netz „santa“ durch die Anschrift `bg:new container` an der Transition „fill bag“ erzeugt. Letztere füllt zudem gleichzeitig den Sack mit Geschenken, indem der synchrone Kanals zum Uplink `:put(t)` des frisch erzeugten Behältnisses mehrfach etabliert wird. Dieser Effekt wird an der Transition „fill bag“ durch mehrere Downlinks zur selben Netzinstanz `bg:put(...)` erreicht. Jeder Downlink benötigt einen separaten Schaltvorgang der passenden Partnertransition. Es schaltet ein Vorkommen der Transition „fill bag“ synchron mit zwei Vorkommen der Transition „put into“ in derjenigen auf dem Muster „container“ basierenden Netzinstanz, welche als Sack im Netz „santa“ referenziert ist.

Im Erreichbarkeitsgraphen in Abbildung 2.8 ist dieser Schaltvorgang oben rechts als Übergang zwischen den Markierungen m2 und m3 zu sehen. An den Kanten des Erreichbarkeitsgraphen stehen Multimengen der schaltenden Transitionen, die Variablenbindungen sind der Übersichtlichkeit halber nicht mit dargestellt. Die Zahlen in eckigen Klammern sind willkürlich vergeben, sie dienen der eindeutigen Identifikation von Netzinstanzen. Eine solche Identifikation ist notwendig, um mehrere Netzinstanzen unterscheiden zu können, die auf dem gleichen Netzmuster basieren (im Beispiel „container[4]“ und „container[5]“).

Der dritte Schaltvorgang im Netzsystem bewirkt das Eintreten des Nikolaus' in das Wohnhaus. Dieser Effekt wird erreicht, indem in der Netzinstanz „system[0]“ die Transition „enter“ mit der Variablenbindung `s = santa[1]; h = house[2]` schaltet. Zu beachten ist hierbei, dass die Marken der Stelle „house“ durch die Transition nur *getestet* werden, da die Stelle mit einer Testkante an die Transition angebunden ist. Wären mehrere Personenmarken auf der linken Stelle außerhalb des Hauses verfügbar, könnten diese alle nebenläufig das Haus betreten, da die Netzmarke „house[2]“ beliebig oft zeitgleich getestet werden kann.

Die Transition „system.enter“ verfügt über einen Downlink zur an die Variable `h` gebundenen Netzinstanz. Dieser Downlink definiert einen einstelligen synchronen Kanal mit dem Namen `enter`, dem im Netz „house“ der von der Transition „enter“ angebotene

Uplink entspricht. Der Kanalparameter wird an die Netzinstanz „santa[1]“ gebunden, da dies die Belegung der Variablen `s` im Netz „system[0]“ ist. Durch Unifikation erhält die Variable `p` im Netz „house[2]“ dieselbe Belegung, so dass im Ergebnis des Schaltvorganges eine Netzmarke mit der Referenz auf „santa[1]“ in der zentralen Stelle der Netzinstanz `house[2]` liegt.

Zugleich verfügt die Transition „house.enter“ über einen weiteren Downlink zu dem nullstelligen synchronen Kanal `enterHouse`, welcher entsprechend der Variablenbelegung von `p` einen passenden Partner in der Netzinstanz „santa[1]“ benötigt. Dieser wird durch den Uplink an der Transition „santa.enter house“ angeboten, so dass diese Transition ebenfalls synchron zu den anderen beiden schaltet. In der Nachfolgemarkierung wird daher die anonymen Marke von der Stelle „starting“ entfernt und durch zwei anonyme Marken auf der Stelle „to do“ ersetzt.

Die weiteren Schaltvorgänge sollen hier nicht in derselben Detailtiefe besprochen werden, da sie auf den selben Techniken – mehrfache Downlinks, Parameterübergabe, Testkanten – aufsetzen. Zu erwähnen ist noch, dass die Transition „house.pick boots“ genau dem Konstruktionsprinzip entspricht, welches zur Simulation horizontaler Synchronisationen im Formalismus der Referenznetze vorgeschlagen wird. Da sie wiederum die beiden zu synchronisierenden Netzinstanzen mit Testkanten bindet, können theoretisch beliebig viele horizontale Synchronisationen nebenläufig stattfinden. Im Beispiel ist die Zahl jedoch auf zwei beschränkt, da weder der Kontrollfluss des Nikolaus' noch der Inhalt des Sacks mehr Marken bereit halten als für zwei nebenläufige Schaltvorgänge benötigt werden. Wie in der ersten S/T-Netz-Version des Modells entsteht die Rautenstruktur im Erreichbarkeitsgraphen zwischen den Markierungen m4 und m7 durch die Variation der Reihenfolge, in welcher die Geschenke in die Stiefel gesteckt werden.

Der Erreichbarkeitsgraph des Referenznetzmodells entspricht in seiner Struktur genau dem des S/T-Netzes (von der Initialisierungstransition einmal abgesehen). Die Schaltvorgänge und Markierungen sind allerdings deutlich komplexer definiert. In diesem kleinen Beispiel mag daher der Aufwand der Modellierung mit Referenznetzen übertrieben erscheinen. Allerdings sind die Netzmuster des Beispiels – wie schon in der Variante im *Netze-in-Netzen*-Paradigma – nun unabhängig voneinander wiederverwendbar und können individuell verändert werden, solange die Schnittstellen gleich bleiben.

Zudem ist das Modell nun mit der Ausdrucksstärke der gefärbten Netze klarer strukturiert. Während beispielsweise in Abbildung 2.5 letztlich Sack und Stiefel die unterschiedlichen Füll- und Leer-Operationen (zwei Geschenke auf einmal oder nur eines?) modelliert haben, werden sie nun durch ein gemeinsames Netzmuster mit grundlegenden Operationen repräsentiert. Die Kontrolle über das Füllen und Leeren liegt nun bei den tatsächlich handlungsfähigen Akteuren im Modell – dem Nikolaus und dem Kind. Hier können wiederum beliebige Anpassungen unabhängig voneinander vorgenommen werden.

### 2.1.4 Renew

Zum Modellieren und Simulieren von Petrinetzen – insbesondere von Java-Referenznetzen – haben Olaf Kummer und Frank Wienberg das Werkzeug „Reference Net

Workshop“, kurz Renew, entwickelt. Renew wird in der Programmiersprache Java entwickelt, steht unter einer Open-Source-Lizenz, wurde 1999 erstmals veröffentlicht und kann seitdem inklusive Quellcode über das Internet bezogen werden [KWD]. In den letzten Jahren hat der Autor dieser Arbeit die Leitung der Weiterentwicklung übernommen. Die letzte veröffentlichte Version von Renew trägt die Versionsnummer 2.1 [KWD06a] und beinhaltet bereits viele Ergebnisse, die im Rahmen der vorliegenden Arbeit entstanden sind. Die aktuelle Entwicklung von Renew wurde zuletzt im Papier [KWD+04] der Forschungsgemeinde vorgestellt.

Die veröffentlichte Version von Renew beinhaltet im Wesentlichen einen grafischen Editor zum Zeichnen von Petrinetzen sowie einen Simulator zur Ausführung derselben. Diese beiden Funktionalitätsbereiche sind miteinander verzahnt, so dass Renew letztlich als integrierte Entwicklungsumgebung (*Integrated Development Environment, IDE*) genutzt werden kann. Sowohl der Editor als auch der Simulator sind generisch angelegt, so dass sie für verschiedene Petrinetzformalismen verwendbar sind. Als Standard-Formalismus sind die Java-Referenznetze enthalten. Aufgrund seiner Ausdrucksmächtigkeit erlaubt dieser Formalismus auch das Kreieren von S/T- oder gefärbten Netzen, da jene Formalismen lediglich eine freiwillige Beschränkung in der Wahl der Anschriften während des Modellierens erfordern.

Die Funktionalität der ausgelieferten Version von Renew umfasst folgende Aspekte:

**Sequentielle oder nebenläufige Simulation** Der Renew-Simulator kann in verschiedenen Modi betrieben werden. Im sequentiellen Modus entspricht das Verhalten der simulierten Petrinetze der Schaltfolgensemantik, wie sie auch im Erreichbarkeitsgraphen abgebildet wird (ohne Zusammenfassung nebenläufig aktivierter Transitionen zu Schaltschritten). Standardfall ist jedoch der nebenläufige Modus, welcher die Partielle-Ordnungs-Semantik von Petrinetzen abbildet, in der mehrere Transitionen nebenläufig schalten können. Der Renew-Simulator überträgt diese Semantik auf realzeitbehaftete Schaltvorgänge, die in unabhängigen Threads abgearbeitet werden. In diesem Modus kann der Renew-Simulator unmittelbar vom Einsatz auf modernen Mehrprozessormaschinen profitieren.

**Simulation mit und ohne grafische Benutzungsoberfläche** Die Funktionalität des Simulators ist von der Funktionalität des Editors durch eine Architekturschicht getrennt, welche den Betrieb des Simulators ohne grafische Benutzungsoberfläche (*Graphical User Interface, GUI*) erlaubt. Es ist zudem möglich, aus der GUI heraus eine laufende Simulation auf einer anderen, über das Netzwerk verbundenen Maschine zu beobachten und zu steuern.

**Persistente Simulationszustände** Der Zustand einer Simulation kann auf Anforderung des Anwenders in eine Datei geschrieben und später wieder eingelesen werden. Der Renew-Simulator kann zudem so konfiguriert werden, dass er den aktuelle Simulationszustand bei jedem Schaltvorgang in einer Datenbank persistiert. Dies erlaubt das Wiederaufsetzen der Simulation vom letzten persistierten Zustand nach einem unerwarteten Abbruch der Simulation.

**Verschiedene Petrinetzformalismen** Der Renew-Simulator ist im Kern generisch auf die Simulation höherer Petrinetze ausgelegt – er bietet Grundbausteine wie Transitionen und Stellen sowie einen Unifikationsalgorithmus auf Werten, Variablen,

Tupeln und Listen. Darauf aufbauend können verschiedenste Petrinetzformalismen implementiert werden, da die Funktionalität der zentralen Klassen zum Ermitteln aktivierter Transitionen durch Unterklassen und geeignet konstruierte Objekte parametrisierbar ist. Streng genommen lässt sich im Renew-Simulator eine noch allgemeinere Schicht zur Simulation von Graphersetzungssystemen ausmachen, so dass die Simulation von Petrinetzen nur einen Anwendungsfall darstellt. Die Möglichkeit zur Simulation anderer Graphersetzungssysteme ist bisher aber nicht ausgenutzt worden.

Zum Lieferumfang von Renew gehören neben dem Java-Referenznetzformalismus ein Formalismus für S/T-Netze, für Petrinetze mit Wahrheitswerten als Marken, für simulationszeitbehaftete Java-Referenznetze [siehe KWD06b, Abschnitt 3.11] und für Feature-Structure-Netze [siehe Wie01, DRW06].

**Java plus Nebenläufigkeit** Der Formalismus der Java-Referenznetze wird im Renew-Simulator eng mit der darunterliegenden Programmiersprache verzahnt. Zum einen ist (wie bereits in Unterabschnitt 2.1.3 beschrieben) die Manipulation beliebiger Java-Objekte möglich. Zum anderen können Netzmuster sich durch das Implementieren von in Java deklarierten Interfaces als Klassen in die Java-Typhierarchie einfügen, so dass Netzinstanzen wie normale Java-Objekte ansprechbar sind. Beide Fähigkeiten zusammen erlauben die einfache Nutzung der umfangreichen Java-Klassenbibliothek in Kombination mit Petrinetzen. Die Java-Referenznetze lassen sich daher als vollwertige Programmiersprache bzw. als Java-Dialekt auffassen, der die Sprache Java um intuitive und ausdrucksstarke Konstrukte zum Umgang mit Nebenläufigkeit und Synchronisation erweitert.

**Grafischer Editor** Die Zeichenwerkzeuge im Renew-Editor basieren auf einem Rahmenwerk für Zeichnungseditoren namens JHotDraw, welches ebenfalls unter einer Open-Source-Lizenz steht. Neben Transitionen, Stellen und Kanten können daher auch viele verschiedene geometrische Formen und Texte ohne Semantik zu einer Zeichnung hinzugefügt werden.[8] Das JHotDraw-Rahmenwerk ist durch Vererbung um beliebige weitere grafische Elemente erweiterbar. Einige Petrinetz-Formalismen nutzen diese Möglichkeit zur Integration spezieller Transitionen, Kanten oder Markendarstellungen.

**Visualisierung** Der Renew-Editor bietet einige Möglichkeiten, die grafische Darstellung des Markenspiels während einer Simulation um visuelle Effekte anzureichern. So können Netzmarken durch gezeichnete Icons anstelle ihrer textuellen Bezeichung dargestellt werden. Beliebige grafische Elemente können abhängig von der Markierung einer Stelle farblich hervorgehoben oder ausgeblendet werden. Für beliebige Markentypen können spezielle Objekte zur Darstellung dieser Marken in Renew registriert werden.

**Syntaxprüfung im Editor** Beim Zeichnen von Netzen steht eine automatische Syntaxprüfung im Hintergrund bereit, welche viele Fehler schon anmerkt, bevor zu

[8] De facto sind alle Zeichnungen in der vorliegenden Arbeit mithilfe des Renew-Editors entstanden (also auch diejenigen, welche keine Petrinetze darstellen). Die einzigen Abbildungen, welche nicht mit Renew erstellt wurden, sind Bildschirm-Schnappschüsse.

Beginn der Simulation das eigentliche Übersetzen der Netzanschriften stattfindet.

**Netzkomponenten** Lawrence Cabac hat in [Cab02] Vorschläge für wiederverwendbare Bausteine aus Netzelementen erarbeitet und eine Werkzeugleiste zur Integration dieser Bausteine in ein Petrinetz in den Renew-Editor integriert. Inzwischen existieren mehrere verschieden thematisierte Werkzeugleisten mit unterschiedlichen Sätzen an Netzkomponenten. In [CDR06b] wird der Einsatz der Netzkomponenten nach einigen Jahren Praxiserfahrung diskutiert.

**Simulationssteuerung/Debugger** Das Markenspiel einer laufenden Simulation kann im grafischen Editor nicht nur beobachtet, sondern auch gesteuert werden. Es ist möglich, im Einzelschrittmodus gezielt einzelne aktivierte Transitionen und ihre Variablenbelegungen zum Schalten auszuwählen. Im Dauerlaufmodus kann die Simulation jederzeit auf explizite Anforderung oder aufgrund eines Breakpoints angehalten werden. Als Breakpoint-Kriterium bieten sich sowohl Transitions-Schaltvorgänge als auch Veränderungen an der Markierung einer Stelle an.

**Dynamisches Laden von Netzen** Bei größeren Projekten ist es nicht praktikabel, zu Beginn der Simulation sämtliche Netze im grafischen Editor öffnen zu müssen. Der Renew-Simulator verfügt daher über die Möglichkeit, während der Simulation Netzmuster bei Bedarf nachzuladen und in die laufende Simulation zu integrieren. Ein nicht-grafisches Dateiformat für Schattennetzsysteme (*Shadow Net System, SNS*) erlaubt das Einlesen der Netzstruktur ohne Umweg über den grafischen Editor. Der Editor ist wiederum in der Lage, zu den in der Simulation verwendeten Netzmustern die grafische Darstellung nachzuladen, sobald sie z. B. im Rahmen des Debuggens benötigt wird.

**Import/Export** Sowohl die grafische Zeichnung als auch die semantische Struktur von Petrinetzen können in verschiedenen Dateiformaten exportiert werden. Somit sind die Integration von Zeichnungen in Dokumente vieler Arten und der Austausch von Netzen mit anderen Petrinetzwerkzeugen möglich. Dies erlaubt beispielsweise den Einsatz von Analysewerkzeugen.

Die lange Liste der Fähigkeiten ist nicht vollständig, illustriert aber die Vielfältigkeit des Werkzeugs. Darüber hinaus existieren diverse Erweiterungen des Werkzeugs, welche im Rahmen von verschiedenen Forschungsprojekten entstanden sind und nicht mit der veröffentlichten Renew-Version ausgeliefert werden.

**Workflow-Engine** Viele Techniken zur Modellierung und Implementierung von Workflows basieren auf Petrinetzen. Der Renew-Simulator kann daher durch geringfügige Ergänzungen zu einer Workflow-Engine ausgebaut werden, in welcher das Bearbeiten von Aktivitäten in einem Workflow mit dem Schalten von Transitionen zusammenfällt. Die Workflow-Erweiterung von Renew beobachtet den Simulationszustand und reagiert auf aktivierte Transitionen, um den Nutzern des Workflow-Systems anstehende Aktivitäten zu präsentieren. Übernimmt oder erledigt ein Nutzer eine Aktivität, so beginnt oder beendet die korrespondierende Transition ihren Schaltvorgang. Thomas Jacob hat in seiner Diplomarbeit

[Jac02] diese Grundfunktionalität um ein Rahmenwerk zur Definition von Rollen, Rechten und Aktivitäten sowie zur Verteilung der Aktivitätslisten über ein Netzwerk erweitert.

**Multiagentensysteme** Wie in Abschnitt 2.5.2 deutlich werden wird, eignen sich Referenznetze gut zur Modellierung von Multiagentensystemen (MAS). Die von Heiko Rölke in seiner Dissertation definierte Referenzarchitektur Mulan [siehe Röl99, Röl04] wurde vom Autor dieser Arbeit in seiner Diplomarbeit [Duv02] durch eine Agentenplattform komplettiert, die teilweise selber in Java-Referenznetzen implementiert ist. Darauf aufsetzend implementierte MAS benötigen also eine Version von Renew, welche die Agentenplattform beinhaltet. Zudem existieren diverse weitere Werkzeuge zum Debugging und zur Entwicklung von MAS auf der Basis dieser Agentenplattform, welche in den grafischen Editor von Renew integriert sind.

**Modellierung** Insbesondere entstand im Rahmen der auf Mulan aufsetzenden petrinetzbasierten agentenorientierten Softwareentwicklung [Paose, siehe CDD+07] der Bedarf nach diversen grafischen Modellierungswerkzeugen. Aus einigen an die Unified Modeling Language (UML, siehe auch Unterabschnitt 2.3.3) angelehnten Modellen können Petrinetze generiert oder gar ein Roundtrip-Engineering zwischen Modellen und Petrinetzen betrieben werden. Das Erstellen und Verarbeiten der Modelle erfordert speziell entwickelte Modellierungswerkzeuge, welche direkt in die Anwendung Renew (in ihrer Funktion als IDE) integriert sein sollten, um bequem verwendbar zu sein. Die Modellierungswerkzeuge sind daher wiederum als Renew-Erweiterungen zu integrieren, da sie zum einen die grafische Grundfunktionalität von JHotDraw benötigen und zum anderen mit Petrinetzen arbeiten.

Renew stellt somit ein mächtiges grafisches Modellierungs- und Implementierungswerkzeug für Softwaresysteme dar. Erweiterungen wie die eben genannten waren zu Beginn der vorliegenden Arbeit aber nur ausführbar, indem sie in den Renew-Quellcode integriert und mit Renew gemeinsam übersetzt wurden. Die Integration vieler solcher Erweiterungen führt jedoch, wie bereits in der Einleitung angemerkt (siehe Abschnitt 1.1), zu einer komplexen, schwerfälligen und umständlichen Bedienung für den Anwender.

Somit handelt es sich bei Renew um ein komplexes Softwaresystem, welches von vielen der in der Einleitung (siehe Kapitel 1) genannten Probleme zur Einführung eines Plugin-Systems betroffen ist. Ein Plugin-System würde es den Anwendern erlauben, nur die jeweils benötigten Funktionen zu installieren. Auch nachträgliche Erweiterungen der Funktionalität wären einfacher integrierbar. Insbesondere der mögliche Wechsel zwischen grafischem Editor und reinem Simulationsbetrieb lässt auch den Wunsch nach dynamischer Rekonfigurierbarkeit der laufenden Anwendung aufkommen.

Die Erkenntnis, dass Renew von der Einführung eines Plugin-Systems profitieren würde, war mit ein Auslöser für diese Arbeit.[9] Daher wird Renew in dieser Arbeit

[9] Der andere wesentliche Auslöser für die Aufgabenstellung dieser Arbeit war (wie in Abschnitt 1.1 angesprochen) der fehlende konzeptionelle Unterbau existierender Plugin-Systeme (siehe Kapitel 3). Dieser ergab sich bei der Suche nach geeigneten Plugin-Systemen zur Einführung in Renew.

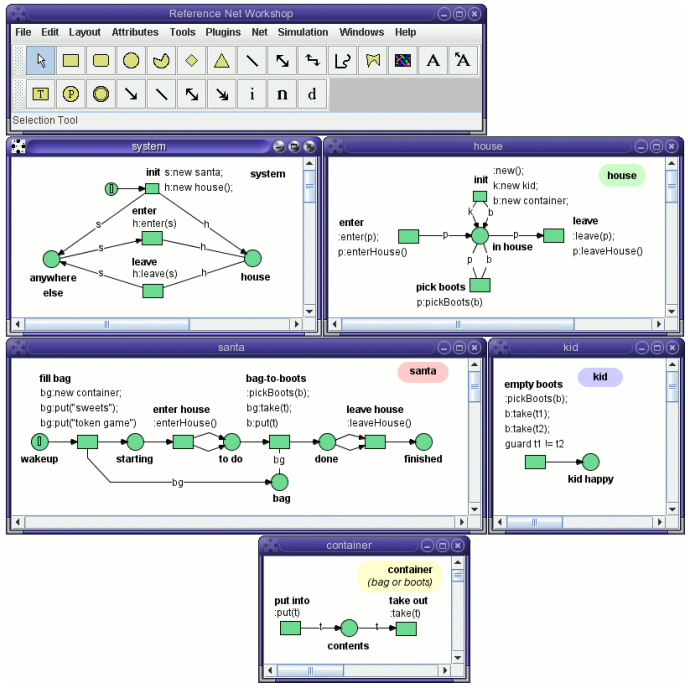



Abbildung 2.9: Die Netze des Nikolausbeispiels im Werkzeug Renew

nicht nur als Werkzeug eingesetzt, um die Denkmodelle darzustellen und zu implementieren, sondern es dient zugleich als Fallstudie zur Tauglichkeit des Plugin-Konzepts, das in dieser Arbeit entworfen wird. Das Werkzeug wird daher in Abschnitt 5.1 noch einmal aus einer softwaretechnischen Sicht heraus betrachtet werden.

#### 2.1.4.1 Beispiel

Die in den vorangegangenen Unterabschnitten präsentierten Versionen des Nikolausbeispiels sind – mit Ausnahme der Variante im reinen *Netze-in-Netzen*-Paradigma – im Werkzeug Renew simulier- bzw. ausführbar. Für die S/T-Netz-Version aus Abbildung 2.1 muss der entsprechende Formalismus (Menü Simulation⇒Formalisms⇒P/T Net Compiler) ausgewählt werden. Die gefärbte Version aus Abbildung 2.3 ist bereits in der Syntax der Java-Referenznetze verfasst und kann daher im Standard-Formalismus („Java Net Compiler“) bearbeitet und simuliert werden. Die in Abbildung 2.7 dargestellte Referenznetzversion besteht aus mehreren separaten Netzen, welche gemeinsam im Editor geöffnet werden sollten. Bei geeigneter Konfiguration des Netzladers ist es zum Ausführen des Beispiels auch ausreichend, lediglich das Netz „system“ zu öffnen, da die anderen Netze beim Simulationsstart automatisch nachgeladen werden.

In Abbildung 2.9 ist ein Bildschirmfoto des grafischen Renew-Editors mit allen Netzmustern der Referenznetzversion des Nikolausbeispiels zu sehen. Oben befindet sich ein Fenster mit Menü und Werkzeugleisten des grafischen Editors. Es handelt sich

bei der dargestellten RENEW-Version um eine aktuelle Entwicklerversion, in der die Ergebnisse dieser Arbeit bereits eingeflossen sind. Der Stand der hier gezeigten Entwicklerversion entspricht grob der geplanten Version 2.2, ihr Umfang ist allerdings um etliche unveröffentlichte Plugins vergrößert. Der vergrößerte Umfang ist andeutungsweise an der Existenz der Menüs „Tools“ und „Plugins“ zu erkennen, in denen viele Erweiterungen des Editors ihre Zusatzfunktionalität verankern.

Die obere Werkzeugleiste erlaubt das Zeichnen geometrischer Figuren ohne Semantik. Die untere Werkzeugleiste dient dem Zeichnen von Petrinetzen. Neben Transitionen, Stellen und den verschiedenen Kantenarten können Anschriften („i“), Knotenbezeichner („n“) und ein Deklarationsknoten („d“) erstellt werden. Letzterer ist zum einen zum Modellieren getypter Netze notwendig, um Variablentypen zu deklarieren. Zum anderen dient der Deklarationsknoten der Aufnahme von Java-`import`-Deklarationen, um auf Java-Klassen im Netzmuster ohne Angabe der voll qualifizierten Klassennamen zugreifen zu können. Im Beispiel werden allerdings keine Deklarationsknoten benötigt, da es ohne Java-Klassen auskommt. Zudem sind alle Stellen und Variablen ungetypt.

Jedes der Netzmuster des Beispiels wird in einem eigenen Zeichnungsfenster dargestellt, da es sich um individuelle Dateien handelt. Die inhaltliche Verbindung zu einem Netzsystem wird erst durch die Transitionsanschriften zum Erzeugen von Netzinstanzen hergestellt. Beim Start der Simulation muss das Fenster aktiv sein, von dessen Netzmuster die erste Netzinstanz erzeugt werden soll, im Beispiel also das Netz „system“. Beim Start der Simulation werden alle geöffneten Netzmuster übersetzt, so dass sie im Simulator verwendet werden können.

In Abbildung 2.10 ist ein Bildschirmfoto einer laufenden Simulation des Nikolausbeispiels zu sehen. Jede Netzinstanz wird in einem eigenen Fenster dargestellt. Diese Fenster weisen zur besseren Unterscheidung von Netzmustern einen blau eingefärbten Hintergrund auf. Marken werden direkt über den Stellen eingeblendet, in denen sie sich momentan befinden. Netzmarken werden durch den Namen des Netzmusters und einen numerischen Identifikator in eckigen Klammern repräsentiert. Ein Klick auf eine Netzmarke öffnet die zugehörige Netzinstanz in einem eigenen Fenster. Um sämtliche Netzinstanzen eines laufenden Systems in eigenen Fenstern zu öffnen, muss der Anwender jede Netzmarke mindestens einmal zu Navigationszwecken anklicken. Alternativ kann er die Netzinstanzen über das im rechten Fenster dargestellte Simulationsprotokoll öffnen.

Das Referenznetzsystem befindet sich in Abbildung 2.10 in dem Zustand, der in Abbildung 2.8 durch die Markierung „m5“ beschrieben wird, allerdings mit abweichenden Instanznummern. Die Netzmarke `santa[293]` ist bereits zwischen den anderen beiden Netzmarken auf der zentralen Stelle der Netzinstanz „house[294]“ einsortiert, also hat der Nikolaus das Haus bereits betreten. Der Fortschritt im Kontrollfluss der Netzinstanz „santa[293]“ wird durch je eine anonyme Marke in den Stellen „to do“ und „done“ angezeigt. Die beiden Netzinstanzen „container[302]“ und „container[295]“, die auf demselben Netzmuster basieren, enthalten jeweils eines der Geschenke. Die Netzinstanz mit der Identifikationsnummer 302 enthält das `"token game"` als Marke und wird von der Netzinstanz „santa[293]“ aus referenziert. Diese Konstellation stellt dar, dass der Nikolaus das Markenspiel noch in seinem Sack hat, die Süßigkeiten aber bereits im Stiefel sind.

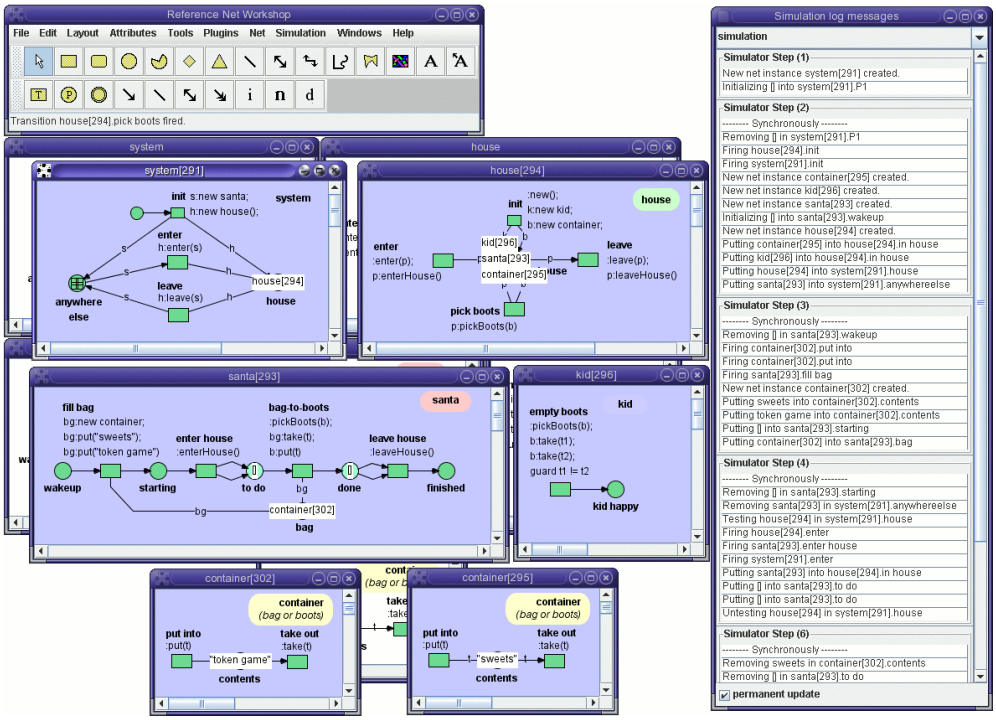



Abbildung 2.10: Simulation des Nikolausbeispiels im Werkzeug Renew

Das Nikolausbeispiel in der hier vorgestellten Version nutzt weder die von Renew gebotenen Visualisierungsmöglichkeiten zur Veranschaulichung der Systemzustände noch die Möglichkeiten, die sich aus der engen Integration in die Java-Laufzeitumgebung ergeben. Das Beispiel wird allerdings im Kontext der prototypischen Implementierung der Plugin-Konzepte in Unterabschnitt 5.3.2 noch einmal aufgegriffen. Die dort vorgestellte Variante basiert zwar auf einer einfacheren Version des Nikolaus-Beispiels, die im Renew-Handbuch zu finden ist [siehe KWD06b, Fig. 3.15], repräsentiert aber mit Hilfe der Visualisierungsmöglichkeiten die verschiedenen Geschenke als grafische Marken.

Die Java-Anbindung im Rahmen der Renew-Ausführungsumgebung könnte im Nikolausbeispiel dazu genutzt werden, das relativ kleine Modell zu einem komplexen Softwaresystem auszubauen. Ein Mischbetrieb von Netzen und Java-Klassen ist auf Basis der Renew-Laufzeitumgebung in beliebiger Kombination möglich. So könnten die bereits modellierten Teile des Systems, in denen die Petrinetze ihre Stärken im Hinblick auf Ablaufmodellierung, nebenläufige Ereignisse und zu synchronisierende Aktivitäten ausspielen, beibehalten werden. Diese ließen sich einfach um Funktionalitäten aus der Java-Klassenbibliothek wie z. B. eine GUI zur Verwaltung des Inhaltes des Sacks oder Netzwerkkommunikation zum Datenaustausch zwischen entfernten Häusern ergänzen. Auch ein Austausch einzelner Netzmuster des Modells durch Java-

Klassen ist denkbar, um z. B. die Effizienz von professionellen Datenbanksystemen in der Verwaltung des Sackinhaltes ausnutzen zu können.

Die Möglichkeiten der engen Java-Integration von Referenznetzen werden in Unterabschnitt 5.3.3 genutzt werden, um eine in Netzen implementierte Plugin-Verwaltung mit einem in Java implementierten Plugin-System zu verbinden.

## 2.2 Komponentenbasierte Softwareentwicklung

In der *komponentenbasierten Softwareentwicklung* (*Component-based Software Engineering, CBSE* oder *Component-based Development, CBD*) wird das Herstellen von Software im Sinne einer Ingenieursdisziplin in den Vordergrund gestellt. Ein Softwaresystem soll durch das Zusammenfügen separat gefertigter *Komponenten* entstehen. Allerdings gibt es bisher für den Begriff *Komponente* keine methodisch und formal gefestigte einheitliche Definition. Stattdessen findet sich eine Vielzahl teils konkreter, teils abstrakter, einander mehr oder weniger ähnlicher Definitionen mit unterschiedlichen Schwerpunkten. So wartet selbst das Standardwerk zu Komponenten-Software [Szy02] von Szyperski mit drei Definitionen auf, die auf unterschiedlichen Abstraktionsebenen liegen.

Zugrunde liegt den Definitionen meist die Idee der Wiederverwendbarkeit von Code, welche durch Komposition eines Systems aus wiederverwendbaren Komponenten ermöglicht werden soll. Bereits 1968 stellte McIlroy in [McI69] fest, dass sich die Software-produzierende Industrie von der individuellen Programmierung von Einzelanwendungen hin zur ingenieurmäßigen Konstruktion von Software aus vorgefertigten, bewährten Bausteinen entwickeln müsse. Der Umfang bzw. die Komplexität der einzelnen Bausteine kann stark variieren, da eine Balance zwischen gegenläufigen Zielsetzungen zu finden ist. Einerseits sind schlanke, einfach wiederzuverwendenden Komponenten oftmals mit vielen Kontextabhängigkeiten versehen, wogegen andererseits große, robustere Komponenten einen reduziertem Wiederverwendbarkeitsgrad aufweisen [vgl. Szy02, Abschnitt 4.1.8, S. 45f.]. Der Grad der Wiederverwendbarkeit ist zudem abhängig vom Einsatzgebiet bzw. dem fachlichen Schwerpunkt der betrachteten Systeme. Somit stehen innerhalb des Fachgebiets der CBSE sehr unterschiedliche konzeptuelle Fragestellungen wie z. B. die Interaktion separater Komponenten in verteilten Systemen, die Herstellung und Verwendung geschäftlich handelbarer Softwarepakete oder die zur Komposition benötigten Verbindungstechniken im Vordergrund. Aus der Vielfalt der Fragestellungen und aus der Existenz vieler programmiersprachen-, betriebssystem- oder herstellerspezifischer Komponententechnologien resultiert die heutige Diffusität des Komponentenbegriffs.

Die Vielfalt der Definitionen des Komponentenbegriffs wird in Unterabschnitt 2.2.1 umrissen, um anschließend in Unterabschnitt 2.2.2 den in mehreren Definitionen herangezogenen Begriff des Komponentenmodells anhand von Beispielen aus der Praxis zu konkretisieren. Eine spezielle Ausrichtung von Komponentenmodellen findet sich im Bereich der WebServices und dienstorientierten Architekturen; sie ist Gegenstand des Unterabschnitts 2.2.3. In Unterabschnitt 2.2.4 werden einige charakteristische Eigenschaften des Blickwinkels erläutert, aus welchem im Paradigma der CBSE ein Softwaresystem betrachtet wird. Diese Eigenschaften dienen in Unterabschnitt 2.2.5 als

Grundlage für eine vorläufige Diskussion von Plugin-Systemen aus dem Blickwinkel der CBSE heraus. Der Komponentenbegriff wird in Unterabschnitt 4.1.9.1 mit dem Begriff des Plugins, wie er in dieser Arbeit definiert wird, verglichen werden.

### 2.2.1 Der Komponentenbegriff

Eine Gegenüberstellung verschiedener Komponentenbegriffe findet sich unter anderem in [Gri01a, S. 21ff.] und in [Szy02, Kapitel 11, S. 195ff.]. Hier sollen lediglich beispielhaft einige Definitionen wiedergegeben werden, die verschiedene Eckpunkte des breiten Spektrums aufzeigen.

Sametinger definiert in [Sam97] eine wiederverwendbare Softwarekomponente folgendermaßen:

> „Reusable software components are self-contained, clearly identifiable artefacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status.“ [Sam97, S. 68]

Die *Eigenständigkeit* (*self-containedness*) stellt sicher, dass eine Komponente ohne Hinzunahme anderer Komponenten verwendet werden kann. Abweichungen von dieser Regel sind in der Praxis kaum vermeidbar, müssen aber klar als Abhängigkeiten (*dependencies*) ausgewiesen werden. Die klare *Identifizierbarkeit* verbietet Komponenten, deren Bestandteile über verschiedene Orte verstreut oder mit anderen Komponenten vermengt sind. Die *Funktionalität* einer Komponente kann von der Komponente entweder direkt erbracht oder auch nur spezifiziert werden. *Schnittstellen* (*interfaces*) kapseln Details, die für das Verwenden einer Komponente nicht benötigt werden. Komponenten können mehrere Schnittstellen haben, die sich in Datenschnittstellen, Benutzerschnittstellen und Programmierschnittstellen einteilen lassen. Ohne *Dokumentation* können Komponenten effektiv nicht eingesetzt oder wiederverwendet werden. Der *Wiederverwendbarkeitsstatus* (*reuse status*) definiert Verantwortlichkeiten, Ansprechpartner und den qualitativen Zustand der Komponente [vgl. Sam97, Abschnitt 6.1.1, S. 68ff.].

Griffel vermeidet in [Gri98] bewusst die enge Verknüpfung des Komponentenbegriffs mit dem Konzept der Wiederverwendbarkeit und zitiert folgende Definition von Jed Harris:

> „Eine Komponente ist ein Stück Software, das klein genug ist, um es in einem Stück erzeugen und pflegen zu können, groß genug ist, um eine sinnvoll einsetzbare Funktionalität zu bieten und eine individuelle Unterstützung zu rechtfertigen sowie mit standardisierten Schnittstellen ausgestattet ist, um mit anderen Komponenten zusammenzuarbeiten.“ [Gri98, S. 31]

Insbesondere dient die damit eröffnete Frage nach der Granularität von Komponenten der Abgrenzung der Komponenten von Objekten, da letztere meist nicht groß genug sind, um ohne Zusammenarbeit mit weiteren Objekten eine Funktionalität erbringen zu können.

Councill und Heineman geben im ersten Kapitel des Sammelbandes „Component-based Software Engineering“ [HC01] eine von allen Autoren gemeinsam erarbeitete Definition von Softwarekomponenten:

> „A *software component* is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.
>
> A *component model* defines specific interaction and composition standards. A *component model implementation* is the dedicated set of executable software elements required to support the execution of components that conform to the model.
>
> A *software component infrastructure* is a set of interaction software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications.“ [CH01, S. 7]

Demnach ist es nicht möglich, eine *Softwarekomponente* zu definieren, ohne sich auf ein bestimmtes Komponentenmodell zu beziehen. Das *Komponentenmodell* erlaubt die Abstraktion von unterliegenden Betriebssystemen oder anderen Komponenten. Es legt stattdessen selber die Regeln zur Herstellung individueller Komponenten sowie der globalen Kommunikation und Interaktion fest, indem ein Standard zur eindeutigen Spezifikation von Schnittstellen definiert wird. Der Interaktionsstandard (*interaction standard*) des Modells legt fest, welche Formen angebotene und benötigte Schnittstellen einer Komponente sowie ihre Abhängigkeiten von der Umgebung (*context dependency*) annehmen dürfen. Daneben bestimmt der Kompositionsstandard (*composition standard*) die zur Komposition einer komplexen Komponente aus Einzelkomponenten benötigten Schritte und Bedingungen. Die *Implementierung* des Komponentenmodells ist dann meist eine dünne Abstraktionsschicht über dem jeweiligen Betriebssystem, welche die Ausführung der unter diesem Modell definierten Komponenten ermöglicht. Interaktionsstandard und Kompositionsstandard finden sich in der Implementierung beispielsweise in Form von Schnittstellenspezifikationssprachen (*Interface Definition Language, IDL*) und grundlegenden Diensten zur Registrierung von Komponenten wieder [vgl. CH01, Abschnitt 1.2, S. 6–14].

Der Begriff der Komponenten-Infrastruktur (*component infrastructure*) wird von Councill und Heinemann nur am Beispiel erklärt. Es handelt sich dabei um ein Erzeugnis der CBSE, welches zunächst als „logischer“ Architektur-Entwurf das Zusammenspiel der Komponenten beschreibt und dann zur Implementierung der gewünschten Anwendungsarchitektur verfeinert wird [vgl. CH01, Abschnitt 1.4, S. 19]. Die Komponenten-Infrastruktur ist also der Teil der konstruierten Anwendung, der ausschließlich aus Komponenten besteht. Daneben kann die Anwendung weitere nicht-komponentenbasierte Teile wie z. B. Code-Bibliotheken oder angebundene Altsysteme enthalten [vgl. HC01, Teil IV, S. 239f.]. Die Komponenten-Infrastruktur kann auch als Basis einer Produktfamilie angesehen werden, deren einzelne Elemente gegebenenfalls durch weitere Komponenten ergänzt werden [vgl. Bas01, Abschnitt 21.1, S. 389f.].

Szyperski gibt in [Szy98, S. xxi, S. 41 u. S. 419f.] drei Definitionen des Komponentenbegriffs auf unterschiedlichen Abstraktionsebenen, die er unverändert in die zweite Auflage [Szy02] übernommen hat. Hier sei zunächst die abstrakteste Definition aus

dem Vorwort wiedergegeben, da sie weniger technisch als die bisher genannten Definitionen ausfällt.

> „Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.“ [Szy02, S. xxi]

Diese Definition betont vor allem den Aspekt der Beteiligung mehrerer unabhängiger Personengruppen an Herstellung, Verkauf und Einsatz von Softwarekomponenten. Dieser Aspekt ist auch Grundlage der bereits wiedergegebenen Definitionen anderer Autoren. Allerdings haben die zuvor genannten Definitionen aus diesem Aspekt bereits weitergehende, konkretere Anforderungen an Komponenten wie z. B. klare Abgrenzung, definierte Schnittstellen, explizite Abhängigkeiten oder Kompositionsstandards abgeleitet, so dass er nicht immer direkt erwähnt wird.

Die anderen beiden Definitionen von Szyperski konkretisieren den Begriff in ähnlicher Richtung wie die bereits genannten Definitionen von Sametinger sowie Councill und Heineman. Der Vollständigkeit halber sei die am häufigsten zitierte Definition hier ebenfalls wiedergegeben:

> „A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“ [Szy02, S. 41]

Diese Definition enthält weiterhin den geschäftlichen Aspekt, welcher sich vor allem in der Eigenschaften der „Komposition durch Dritte“ (*composition by third parties*) widerspiegelt. Die Eigenschaften der Komposition, der klar spezifizierten Schnittstellen, der explizit deklarierten Abhängigkeiten sowie der „unabhängigen Einsetzbarkeit“ decken sich mit den bereits zuvor diskutierten Definitionen, so dass sie hier nicht weiter ausgeführt werden sollen.

Die Abgrenzung zwischen Objekten und Komponenten kann anhand verschiedener Konzepte stattfinden. Bereits genannt wurde die Granularität, die sich z. B. darin äußert, dass Komponenten üblicherweise mehrere Klassen umfassen, deren Ober- und Unterklassen wiederum aus anderen Komponenten stammen dürfen. Oft werden auch Schnittstellen von Komponenten komplexer definiert als Schnittstellen von Objekten, indem z. B. Verhaltensbeschreibungen nebst einzuhaltenden Interaktionsmustern oder nichtfunktionale Eigenschaften in die Komponentenspezifikation mit aufgenommen werden [vgl. Rob04, S. 153ff.]. Wie Szyperski erläutert, müssen zudem gemeinsame Schnittstellen konkurrierender Komponenten in langwierigen Prozessen über Marktmechanismen wie z. B. Standards etabliert werden [vgl. Szy02, Abschnitt 4.1.6, S. 42f.].

Desweiteren besteht nach Szyperski ein wichtiger Unterschied zwischen Objekten und Komponenten in der Identifizierbarkeit und Instantiierbarkeit. Objekte sind Instanzen von Klassen, sie haben eine Identität und kapseln neben ihrem Verhalten auch ihren Zustand. Komponenten hingegen werden bei ihrem Einsatz installiert oder komponiert statt instantiiert. Daher sollte eine Komponente keinen nach außen wahrnehmbaren Zustand in sich kapseln, sondern z. B. lediglich anhand von übergebenen Referenzen auf persistente Daten zugreifen [vgl. Szy02, Abschnitte 4.1.2–4.1.3, S. 36ff. und 8.10, S. 146].

### 2.2.2 Komponentenmodelle

Ein wesentliches Element in der CBSE ist das gewählte, der Software zugrundeliegende Komponentenmodell (siehe die im Unterabschnitt 2.2.1 zitierte Definition aus dem ersten Kapitel des Sammelbandes von Heineman und Councill [HC01]). Als alternative Bezeichnungen für ein Komponentenmodell finden sich in der Literatur auch *Komponenten-Architektur* oder der etwas technischer orientierte Begriff *Komponenten-Standard*. Szyperski nutzt in seiner detaillierten, technischen Definition des Komponentenbegriffs das gleichbedeutende Konzept eines *Komponenten-Rahmenwerkes* [vgl. Szy02, S. 420]. An den Eigenschaften von Komponentenmodellen lässt sich recht gut ablesen, wie der Komponentenbegriff schwerpunktmäßig verstanden wird.

Weinreich und Sametinger konkretisieren die Bestandteile eines Komponentenmodells im dritten Kapitel des genannten Sammelbandes [siehe WS01, Abschnitt 3.2, S. 37–45]. Dazu gehören demnach

- die bereits genannte IDL zur Definition von *Schnittstellen*, welche als Vertrag zur Verwendung der Komponenten betrachtet werden,
- eine Regelung zur Vergabe eindeutiger *Namen*, die z. B. auf hierarchische Namensräume aufbauen kann,
- eine Konvention zur Speicherung von *Metadaten*, welche die Schnittstellen, Komponenten und Beziehungen so beschreiben, dass Werkzeuge und selbstreflektierende Programme darauf zugreifen können,
- ein Mechanismus, der die im Modell benötigte *Interoperabilität* – sei es zwischen verschiedenen Herstellern, Programmiersprachen oder in verteilten Systemen – herstellt,
- eine Schnittstelle, die auf Metadaten und Interoperabilität aufbauend einfache *Anpassbarkeit* (*customization*) des Verhaltens der Komponenten ermöglicht,
- der bereits genannte Kompositionsstandard, der je nach Bedarf durch verschiedenste Techniken wie z. B. Skriptsprachen, visuelle Programmierung oder die Komponenten-Infrastruktur realisiert werden kann,
- eine Unterstützung der *Weiterentwicklung* (*evolution*) des Systems, wenn beispielsweise ältere Komponenten durch neuere abgelöst werden sollen, und
- ein Verfahren zur *Auslieferung, Installation und Konfiguration* (*deployment*) von Softwarekomponenten, so dass andere Komponentenentwickler darauf aufbauen können.

Andresen vertritt in [And03] eine stärker an der Praxis orientierte Definition eines Komponentenmodells:

> „Ein Komponenten-Standard spezifiziert
>
> 1. Syntax und Semantik der Komponenten,
> 2. eine Laufzeitumgebung,

3. spezifische Basis-Dienste (Erzeugung, Aktivierung, Deaktivierung von Komponenten),
4. horizontale Dienste, die verschiedenen Bereichen zur Verfügung stehen (Persistenzmanagement, Transaktionsverhalten, Sicherheit, Verzeichnisdienste etc.) und
5. vertikale Dienste, die einem spezifischen Bereich zur Verfügung stehen (Business-Prozesse, -Entitäten, -Services etc.).

Ein Komponentenstandard nutzt sowohl spezifische Applikations-Server und -Container als auch spezifische Middleware.“ [And03, S. 255]

Es existieren diverse, teils offene, standardisierte oder proprietäre Komponentenmodelle, die durch eine oder mehrere Komponentenmodell-Implementierungen unterstützt werden. Die prominentesten Beispiele sind Microsofts mit aktuellen Windows-Versionen ausgeliefertes *Component Object Model (COM+)*, das auch in .NET Verwendung findet, die von Sun spezifizierten und von diversen Herstellern implementierten *Enterprise Java Beans (EJB)* [DK06] und das von der Object Management Group (OMG) spezifizierte, auf der Common Object Request Broker Architecture (CORBA) aufbauende *CORBA Component Model (CCM)* [OMG06]. Eine Gegenüberstellung und Klassifikation der Modelle bzw. ihrer Vorläufer findet sich in vielen Büchern zum Thema CBSE, beispielsweise in [Gri98, Abschnitt 3.1.4], [Lon01], [Szy02, Kapitel 13–15 u. 21], [And03, Kapitel 10] oder [Zwi05].

Das in der Praxis vorzufindende Verständnis des Komponentenbegriffs wird im wesentlichen durch diese drei Komponentenmodelle geprägt, die aber sehr verschiedene Ursprünge und Zielsetzungen haben. COM+ ist aus einer Architektur für zusammengesetzte Dokumente hervorgegangen, so dass der ursprüngliche Schwerpunkt im Bereich der grafischen Anzeige- und Bearbeitungswerkzeuge der einzelnen Dokumentteile lag. COM+ bietet ein programmiersprachenunabhängiges Typsystem und einen Web-Service-ähnlichen Mechanismus zu Kommunikation zwischen verteilten Komponenten, bleibt aber Windows-betriebssystemspezifisch. Das CCM hingegen ist als Bestandteil der CORBA spezifiziert, so dass es von vorneherein eng mit dem Middleware-Gedanken einer Architektur verbunden ist, welche die Kontaktaufnahme und Kommunikation zwischen verteilten Objekten in heterogenen Netzwerken durch Verzeichnisse von angebotenen Diensten ermöglicht.

Enterprise Java Beans setzen auf der plattformunabhängigen Programmiersprache Java auf. Sie bieten dem Entwickler Abstraktionen für persistente Datenhaltung, Prozesse, Transaktionen, Kommunikation zwischen verteilten Servern und Clients, Plattformunabhängigkeit und einen Konfigurationsmechanismus, so dass darauf aufbauende Komponenten fast nur noch auf der fachlichen Ebene implementiert werden müssen. Als mögliches Einsatzgebiet werden explizit WebServices (s.u.) genannt. [Vgl. DK06, S. 34]

Das CCM und die EJB sind recht ähnlich angelegt, wobei das CCM nicht nur plattform-, sondern auch programmiersprachenunabhängig ist. Beide Komponentenmodelle sehen eine Unterscheidung zwischen *Entity*- und *Session*-Komponenten vor. Erstere stellen Datenzugriffe bereit und abstrahieren von der darunterliegenden Speichertechnik. Letztere implementieren Prozesse, die für die Dauer einer Session an einen Client

gebunden sind und auf die Datenkomponenten zwecks Manipulation zugreifen. Daneben bieten beide Modelle noch ein bis zwei weitere Komponentengattungen an.[10] [Vgl. Zwi05, S. 36f. u. S. 48f.]

Die einfachen *JavaBeans* stellen eine für die Programmierung in Java formulierte Konvention zum Implementieren visuell manipulierbarer Objekte dar, haben aber außer der zugrundeliegenden Programmiersprache nichts mit EJB gemeinsam. [Vgl. Zwi05, S. 33]

Diese industriell eingesetzten Komponentenmodelle sind allesamt *flach*, unterstützen also keine hierarchische oder anderweitige Strukturierung von Komponenten. Hierarchische Komponentenbeziehungen sind beispielsweise dann von Interesse, wenn – wie in Architekturbeschreibungssprachen (siehe Abschnitt 2.3) häufig unterstützt – die Zusammensetzung von Komponenten aus Subkomponenten betrachtet wird. Einige akademische Komponentenmodelle bieten daher eine hierarchische Kapselung von Komponenten an. Die hierarchische Organisation von Komponenten wirft aber weitere Fragestellungen auf, unter anderem, wie Dienstleistungskomponenten in das Modell integriert werden können, die quer zur Komponentenhierarchie von vielen Komponenten benötigt werden [vgl. HP06].

Lau und Wang verstehen in [LW07, S. 710] ein Komponentenmodell als Definition von Syntax, Semantik und Komposition von Komponenten und folgen damit der bereits erwähnten Begriffsdefinition von Councill und Heinemann. Komponentenmodelle lassen sich nach Lau und Wang grob danach einteilen, ob Komponenten mit der Semantik von Klassen, von Objekten oder von Elementen einer Systemarchitektur versehen sind. Letztere Semantik wird vor allem in Architekturbeschreibungssprachen (siehe Abschnitt 2.3) verwendet, während die meisten der in diesem Unterabschnitt vorgestellten Komponentenmodelle auf objektorientierten Programmiersprachen basieren und daher eine Objekt- oder Klassensemantik für Komponenten verwenden. Im Falle der Klassensemantik stellt die Programmiersprache gleichzeitig die Komponentendefinitionssprache (z. B. bei EJB), während bei der Objektsemantik auf Schnittstellendefinitionssprachen (z. B. bei COM oder CCM) zurückgegriffen wird.

### 2.2.3 Komponenten und Dienste

Ein in fast allen Komponentenmodellen vorzufindendes Merkmal ist die Bereitstellung von Diensten durch Komponenten, so dass die Dienste über ein zentrales Diensteverzeichnis komponentenübergreifend angeboten und genutzt werden können. Alle Komponentenmodelle beinhalten Konventionen zur Syntax und Semantik der Metadaten, welche die Schnittstellen und Dienste von Komponenten beschreiben, sowie Implementierungen, die als Laufzeitumgebung fungieren und Standardaufgaben wie das Erzeugen von Komponenteninstanzen oder das Auffinden von Dienstanbietern übernehmen.

Dieses Merkmal der Veröffentlichung von Dienstleistungen ist auch das wesentliche Element der dienstorientierten Softwareentwicklung (*Service-oriented Software Engi-*

[10] Das EJB-Modell bietet noch *Message Driven Beans* an, die auf asynchrone Nachrichten reagierende Prozesse oder Funktionalitäten implementieren, während das CCM mit den *Service*-Komponenten eine zustandslose Variante der Session-Komponenten und mit den *Process*-Komponenten eine nicht-öffentliche Variante der Entity-Komponenten bereitstellt. [Vgl. Zwi05, S. 36f. u. S. 48f.]

*neering, SOSE*). Das prominenteste Einsatzgebiet der SOSE sind die so genannten *WebServices*. WebServices werden vom W3C umrissen als die Schnittstellen, die ein Dienstangebot im World Wide Web, welches üblicherweise interaktiv genutzt wird, für Kommunikation zwischen programmierten Anwendungen nutzbar machen [vgl. W3C]. Obwohl es nicht zwingend erforderlich ist, werden WebServices in der Regel auf Basis von Komponententechnologie und bestehender Middleware implementiert [vgl. ACKM04]. Die neueren Versionen der Spezifikationen der praxisrelevanten Komponentenmodelle betonen durchweg die Eignung des jeweiligen Modells für WebServices.

Unter dem Schlagwort der dienstorientierten Architektur (*Service-oriented Architecture, SOA*) wird die Strukturierung von verteilten Softwaresystemen entlang von Geschäftsprozessen verstanden. Während die grundlegende Struktur einer Architektur ähnlich wie bei WebServices und komponentenbasierten Systemen aus dienstanbietenden Komponenten besteht, steht bei der SOA die Zugehörigkeit der beteiligten Dienste zu verschiedenen Beteiligten (Firmen oder Geschäftsbereichen) im Vordergrund. SOA macht keine Vorgaben zur konkreten Umsetzung dienstorientierter Architekturen, ist also nicht an die Technik der WebServices gekoppelt.

Die Zielsetzung und der Lösungsansatz der CBSE und der SOSE sind sehr ähnlich. So bringen gerade die in der Praxis anzutreffenden Komponentenmodelle eine Infrastruktur zur Implementierung verteilter Systeme mit. Sowohl DCOM als auch EJB und CCM integrieren eine Verteilungsschicht, welche die Komplexität der Kommunikation und Interaktion in einem verteilten System für Softwareentwickler reduziert. Anders herum weist Greenfield in [Gor06, Abschnitt 12.10] darauf hin, dass die wesentliche Leistung der Disziplin der WebServices darin besteht, einheitliche, plattformübergreifende Standards für Dienstbeschreibungen und Kommunikationsmittel zu definieren. Greenfield sieht in WebServices genügend Potential, um die entsprechenden Spezifikationen einzelner Komponentenmodelle und Middleware-Rahmenwerke abzulösen. Die WebService-Standards können daher als Komponentenmodell im Sinne der CBSE aufgefasst werden.

Allerdings setzen die Forschungsgemeinden der CBSE und der SOSE unterschiedliche Schwerpunkte bzw. sind in einzelnen Themengebieten unterschiedlich weit entwickelt. Iribarne stellt in [Iri04] fest, dass die CBSE-Forschung wenig Fortschritte in der Verhaltensbeschreibung von Schnittstellen macht, während die WebServices in der Granularität und Mächtigkeit der Softwarearchitektur und Komposition von Diensten Aufholbedarf haben. Breivold und Larsson stellen in [BL07] die komponentenbasierte und die dienstorientierte Softwareentwicklung einander gegenüber. Sie stellen dabei fest, dass beide Paradigmen kaum in sich gegenseitig ausschließender Reinform eingesetzt werden. Beide Paradigmen würden ähnliche Konzepte verwenden und ähnliche Lösungsstrategien verfolgen, um die Komplexität von Software zu bewältigen. Die SOSE sei aber auf plattform- und unternehmensübergreifende Komposition während der Laufzeit ausgerichtet und betone daher noch stärker als die CBSE Eigenschaften wie Kapselung und explizite Schnittstellendeklarationen.

Viele Forscher bemühen sich derzeit um eine Integration der SOSE und der CBSE, indem dienstorientierte Konzepte in Komponentenmodelle eingeführt oder Begriffe aus dem komponentenorientierten Paradigma auf dienstorientierte Architekturen übertragen werden. Ein naheliegender Ansatz insbesondere im Bereich der WebServices besteht darin, die Begriffe *Komponente* und *Dienst* in dem Sinne gleichzusetzen, dass

jeder Dienstanbieter eine Komponente ist und die Dienstspezifikation der Schnittstellenbeschreibung der Komponente entspricht (vgl. z. B. [Sto05, Abschnitt 2.5, S. 40f.] oder [ABH+07a]). Dieser Ansatz vernachlässigt aber, dass eine Komponente mehrere Schnittstellen besitzen und daher auch mehrere Dienste anbieten kann [vgl. Iri04].

Ein umfangreiches Integrationsprojekt stellt die so genannte *Service Component Architecture (SCA)* [siehe SCA, CSA] dar. In der SCA sollen die Technik der Webdienste in heterogenen Netzwerken, die geschäftlichen Organisationsprinzipien der SOA und die Softwarearchitekturprinzipien der CBSE in einer programmiersprachenunabhängigen Sammlung von Spezifikationen integriert werden. Die durch die SCA-Spezifikationen beschriebene Softwarearchitektur verbindet die Idee der SOA in verteilten Systemen mit der vor allem im Bereich der Architekturbeschreibungssprachen verfolgten Abstraktion der Verbinder (*binding*, vgl. *connector* im Unterabschnitt 2.3.2), um die von den Komponenten angebotenen Dienste über verschiedenartige Kommunikationswege koppeln zu können. Verschiedene Autoren sind dabei, Formalisierungen und Rahmenwerke zur SCA zu entwickeln [siehe z. B. ABH+07a, Wen07].

In der Java-Entwicklergemeinde wird seit seiner Integration in den Unterbau der Eclipse-Plattform (siehe Abschnitt 3.2) das erstmals 2000 veröffentlichte Komponentenmodell der Open Services Gateway Initiative (OSGi, siehe [OSG05], [OSG06a] und [OSG06b]) zunehmend bekannter. Das OSGi-Rahmenwerk war zunächst auf den Langzeitbetrieb in eingebetteten Systemen spezialisiert und bietet daher Techniken zur dynamischen Veränderung oder Aktualisierung der Systemkonfiguration zur Laufzeit. Die *Bundles* genannten Komponenten werden über Dienstnutzungsbeziehungen dynamisch gekoppelt (siehe Unterabschnitt 3.2.1.6).

Im Komponentenmodell des OSGi-Rahmenwerks sind somit die Voraussetzungen vorhanden, um Softwaresysteme dienstorientiert zu implementieren. Der verteilte Betrieb einer OSGi-Laufzeitumgebung über mehrere Rechner ist allerdings nicht vorgesehen; zudem sind Komponentenmodell und Laufzeitumgebung des OSGi-Rahmenwerkes auf die Programmiersprache Java festgelegt. Aufgrund des ursprünglichen Einsatzgebietes für eingebettete Systeme gehören zur Spezifikation des OSGi-Rahmenwerkes auch die Definitionen diverser Standard-Dienste zur entfernten Administration oder zur Unterstützung von Netzwerkkommunikation. Die Spezifikationen der SCA lassen sich prinzipiell in einer OSGi-Umgebung implementieren, so dass die Fähigkeiten beider Architekturen gemeinsam genutzt werden können. [Vgl. Wen07]

### 2.2.4 Dynamische Veränderungen in komponentenbasierten Systemen

Im Forschungsfeld der CBSE wird im Wesentlichen die Sicht der Software-*Hersteller* eingenommen. Sowohl die Implementierung von Komponenten als auch deren Verwendung während der Montage des Systems fallen nicht in den Aufgabenbereich von *Anwendern* der Software. Beispielsweise nennt Griffel in [Gri01a] drei Phasen der komponentenbasierten Softwareentwicklung:

- „In der *Entwicklungsphase* werden benötigte Komponenten bzw. Teilbausteine entwickelt.

- In der *Kompositionsphase* werden Komponenten zu einer Gesamtanwendung zusammengefügt.
- In der *Laufphase* läuft die Anwendung. Veränderungen während der Laufphase werden als *dynamische* Veränderungen bezeichnet."

[Gri01a, S. 19]

Eine derartige Einteilung hat sich in weiten Bereichen der CBSE etabliert, so bauen z. B. Lau und Wang in ihrer in [LW07] eine Taxonomie auf Basis der Kompositionsmöglichkeiten in den drei Phasen *Entwurf* (*Design*), *Einsatz* (*Deployment*) und *Laufzeit* (*Run-Time*) auf (siehe Tabelle 2.1, S. 56). Lau und Wang haben allerdings die untersuchten Komponentenmodelle danach unterschieden, in welcher der ersten beiden Phasen die Komposition von Komponenten durchgeführt werden kann. Die *Kompositionsphase* nach Griffel kann daher wahlweise der *Entwurfsphase* oder der *Einsatzphase* nach Lau und Wang zugeordnet werden. Die *Laufphase* nach Griffel ist dazu passend so zu verstehen, dass sie zusätzlich zur gleichnamigen Phase von Lau und Wang auch die *Einsatzphase* umfassen kann.

Es fällt auf, dass die – für Plugin-Systeme besonders interessante – Installation und Konfiguration der Software durch Anwender nicht explizit thematisiert wird. Es bleibt offen, ob die Installation von Komponenten durch Anwender der Kompositionsphase oder den dynamischen Veränderungen der Laufphase zuzuordnen ist. Mit dem Begriff der *dynamischen* Veränderungen werden meist nur die Wartung und Pflege des konstruierten Systems erfasst (vgl. Charakterisierung von Komponentenmodellen durch Weinreich und Sametinger in Unterabschnitt 2.2.2). Lau und Wang schließen in [LW07, S. 711] die dynamische Komposition von Komponenten aus ihrer Taxonomie aus, da auf Komponenten während der Laufphase kaum sinnvolle Kompositionsoperationen definierbar seien.

Szyperski hat erst in der zweiten Auflage seines Buches [Szy02, Abschnitt 24.6, S. 478f.] einen kurzen Abschnitt zur Montage (*assembly*) von Komponenten durch Endanwender aufgenommen. Er ordnet diese Möglichkeit als halbautomatisches Vorgehen zwischen statischer Montage im Voraus und dynamischer, selbstorganisierter Montage der Komponenten ein und weist darauf hin, dass Endanwender – anders als Softwareentwickler – das System aus einem anwendungsorientierten Blickwinkel heraus betrachten. Szyperski stellt keine Verbindung zu Plugin-Systemen her.

In der EJB-Spezifikation werden sieben Parteien identifiziert, die an der Entwicklung, Komposition und dem Einsatz der Komponenten beteiligt sind. Der Komponentenliefererant (*Enterprise Bean Provider*) erstellt den Code und die Metadaten einer fachlichen Komponente. Der Anwendungskonstrukteur (*Application Assembler*) kombiniert eine oder mehrere Komponenten (die nicht notwendigerweise EJBs sein müssen) mit einer Bauanleitung, so dass der *Deployer* die Anwendung nach dieser Anleitung in eine Laufzeitumgebung installieren kann. Die Laufzeitumgebung besteht aus Server (Betriebssystem, Middleware, etc.) und Container (EJB-spezifische Ergänzung des Servers sowie passende Werkzeuge), die vom *EJB Server Provider* und dem *EJB Container Provider* geliefert werden. Der (*Persistence Provider*) stellt Datenbankfunktionalität zur Nutzung durch die Komponenten bereit. Ein *System-Administrator* ist dafür verantwortlich, das System aus Hardware, Netzwerk, Server, Container, etc. zu

konfigurieren und am Laufen zu halten. *Nutzer* der durch das System bereitgestellten Anwendung werden nicht erwähnt. [Vgl. DK06, Abschnitt 2.2, S. 31–33]

Apperly erläutert in [App01] die Notwendigkeit und Aufgaben einen Komponentenmanagers und meint damit ein Werkzeug, das Software-Entwickler beim Herstellen, Bereithalten, Auswählen und Einsetzen von Komponenten unterstützt.

Griffel thematisiert in [Gri98, Abschnitt 5.3] die *Anpassbarkeit* von Komponenten.

> „Anpassung ist [...] eine (vom Komponentenentwickler) vor(her)gesehene *Konfiguration* der *Eigenschaften* einer Komponente (*Customizing*), im Gegensatz zu einer wirklichen *Modifikation*." [Gri98, S. 481f.]

Es geht bei einer Anpassung also nicht darum, eine Komponente tatsächlich zu verändern, sondern nur darum, sie mit unterschiedlichen Parametern einzusetzen. Solche Anpassungen werden durch Komponenten- und Anwendungsentwickler oder auch durch Kunden bzw. Endanwender durchgeführt. Üblich ist die Parametrisierung der Komponente durch manipulierbare Eigenschaften (*Properties*). Möglich ist aber auch die Verwendung spezifischer, als Bestandteil der Komponente mitgelieferter Konfigurationswerkzeuge. Die Liste verfügbarer Eigenschaften und Werkzeuge kann wahlweise vom Komponentenentwickler explizit durch Metadaten bereitgestellt oder durch Nutzung von Reflexion über dem Code automatisch generiert werden. [Vgl. Gri98, S. 481–498]

Eine Fortführung der automatischen Generierung von Metadaten durch Reflexion wären selbstadaptive Komponenten, die sich der Umgebung automatisch anpassen. Auch könnten Techniken des Soft-Computing wie z. B. Fuzzy-Logik, Neuronale Netze oder Genetische Algorithmen so auf Komponenten übertragen werden, dass die Kopplung von Komponenten leichter fällt. Der Preis dafür ist eine schwerere Nachvollziehbarkeit des Verhaltens des kombinierten Systems. [Vgl. Gri98, S. 499–529]

Eine konsequente Fortführung der eben angedachten Flexibilisierung von Komponenten mündet schnell in die agentenorientierte Softwareentwicklung (AOSE), die Verwandtschaft beider Gebiete ist das Thema des Unterabschnitts 2.5.3.

### 2.2.5 CBSE und Plugin-Systeme

Einige Autoren, deren Arbeiten in späteren Abschnitten dieser Arbeit vorgestellt werden, betrachten Plugin-Systeme als eine Variante komponentenbasierter Systeme. Dabei weisen die Autoren auf erkannte Gemeinsamkeiten und Unterschiede hin.

Chatley stellt in [Cha05, Abschnitt 1.1, S. 2ff.][11] zunächst fest, dass die bei der Definition des Komponentenbegriffs (siehe Unterabschnitt 2.2.1) häufig erwähnte Eigenschaft der „unabhängigen Auslieferbarkeit" oder „Komposition durch Dritte" [vgl. Szy02, S. 41] ebenso gut auf Plugin-Architekturen anwendbar sei. Ein wesentlicher Unterschied sei jedoch, dass eine Plugin-basierte Anwendung auch ohne installierte Plugins arbeitsfähig sei. Zudem zeichne sich ein Plugin-System dadurch aus, dass der Aufwand zur Konfiguration und Administration so gering wie möglich ausfällt. Sowohl das Anpassen eines Systems an die Anforderungen der konkreten Einsatzumgebung als auch das Aktualisieren von Systemkomponenten sollen einfach durchführbar sein.

[11] Zur von Chatley vorgestellten Plugin-Architektur *MagicBeans* siehe Unterabschnitt 3.3.4.

Wolfinger et al. ordnen in [WDPM06, S. 288][12] Plugin-Ansätze als erfolgreichen Schritt auf dem Weg zur Etablierung komponentenbasierter Systeme ein. Damit betrachten sie Plugin-Systeme als Spezialfall komponentenbasierter Systeme und nennen als Beispiel Eclipse als eine technologische Basis zur Entwicklung wiederverwendbarer Komponenten. Für den Erfolg von Plugin-Systemen machen sie drei qualitative Eigenschaften der jeweiligen Softwarearchitektur verantwortlich. Zum ersten seien Plugin-Komponenten wie kleine Anwendungen, d. h. sie würden einen Satz in sich geschlossener Funktionalität mit unmittelbarem Nutzen für den Anwender bieten, der nur wenige Abhängigkeiten zu anderen Komponenten aufweise. Zum zweiten gebe die Softwarearchitektur feste Regeln vor, wie Abhängigkeiten und Interaktionen zwischen Komponenten deklariert werden müssten. Und zum dritten seien Plugin-Systeme Beispiele dafür, dass das nahtlose Zusammenfügen von Komponenten zu einer Arbeitsumgebung möglich sei, so dass Anwender aus der wachsenden Zahl von Komponenten eine individuelle Arbeitsumgebung zusammenstellen könnten. [Vgl. WDPM06, S. 288]

Auch Marquardt spricht in [Mar06, S. 302f.][13] von einer „Plugin-Variation der Komponenten“ (*plug-in flavor of components*) bzw. von Plugin-Komponenten. Beide Konzepte würden zur Lösung ähnlicher Probleme eingesetzt, zudem sei ihnen gemein, dass sie den Bedarf nach Erweiterbarkeit bedienen. Unterschiede sieht Marquardt zum einen darin, dass Plugins anwendungsspezifische Erweiterungen in einem anwendungsspezifischen Kontext darstellten, während Komponenten generischer und für die Wiederverwendung in verschiedenen Kontexten ausgelegt seien. Wie Chatley sieht Marquardt zudem, dass durch Plugins erweiterbare Anwendungen auch ohne Plugin schon nützlich sind, während eine Komponentenmodellimplementierung keine eigene Anwendungsfunktionalität enthalte. Im Gegenzug bedeute dies, dass Plugins stärker in ihren Kontext integriert seien und sie eher in spezielleren als in allgemeineren Aspekten erweitern. Komponenten hingegen stellten abgeschlossene Funktionalität bereit. Die Dekomposition von komponentenbasierten Systemen ist laut Marquardt in erster Linie durch die technischen Grenze zwischen Umgebung und Komponente definiert, während Plugin-Systeme vorrangig Anwendungsfunktionalität voneinander separieren und diese erst nachrangig von technischen Aspekten trennen. Dementsprechend seien Plugins inhaltlich abgeschlossen und enthielten fachlich zusammenhängende Funktionalität über mehrere Schichten der Anwendungsarchitektur hinweg. Anders als Komponenten könnten Plugins separat ausgeliefert werden und sogar unter von der eigentlichen Anwendung abweichender Lizenz vertrieben werden.

Insbesondere der letzte von Marquardt angesprochene Punkt verdeutlicht noch einmal die bereits in Unterabschnitt 2.2.4 diskutierte Eigenschaft von komponentenbasierten Systemen, dass die Komposition eines Systems aus Komponenten durch Softwareentwickler durchgeführt wird, *bevor* das System an den Kunden ausgeliefert wird (bzw. dass die Wartung einer komponentenbasierten Systemkonfiguration wie bei monolithischen Systemen Aufgabe der Entwicklerfirma bleibt).

Die von Wolfinger et al. genannten erfolgsrelevanten Eigenschaften einer Plugin-Architektur stellen tatsächlich eine Konkretisierung allgemeiner Konzepte komponentenbasierter Architekturen dar – die Anwendungsähnlichkeit der Komponenten, die

[12] Zur von Wolfinger et al. vorgestellten Plugin-Architektur *CAP.NET* siehe Unterabschnitt 3.3.3.
[13] Zum von Marquardt definierten Plugin-Entwurfsmuster siehe Unterabschnitt 2.4.3.

festen Integrationsregeln und die nahtlose Zusammenfügbarkeit der Komponenten zu einer Arbeitsumgebung lassen sich als Eigenschaften eines Komponentenmodells (siehe Unterabschnitt 2.2.2) verstehen.

Kornstädt definiert in [Kor02] ein Einschubsystem als eine den Plugin-Systemen ähnliche Softwarearchitektur, welche im Spannungsfeld zwischen Rahmenwerken und Komponentensystemen angesiedelt ist. Dabei erläutert er die Eignung verschiedener gebräuchlicher Metaphern für Einschubsysteme und beschreibt eine Java-basierte Implementierung dieser Softwarearchitektur, welche speziell auf den Einsatz im Gebiet der musikwissenschaftlichen Analyse zugeschnitten ist und auf einfache Weise durch den Anwender konfigurierbar ist.

Keiner der Autoren nennt aber umgekehrt Eigenschaften komponentenbasierter Systeme, die in Plugin-Systemen *nicht* anzutreffen sind. Ein paar Gedanken des Autors dieser Arbeit zu diesem Punkt sind folgende:

- Die Aspekte *verteilter Systeme*, die regelmäßig beim Einsatz von komponentenbasierten Systemen im Bereich der WebServices (siehe Unterabschnitt 2.2.3) zum Tragen kommen, spielen bei den existierenden Plugin-Systemen keine Rolle, da diese Plugin-Systeme überwiegend unter der Kontrolle *eines* Anwenders, und damit nur innerhalb eines Betriebssystem-Prozesses ausgeführt werden. Ebensowenig wird ein hoher Grad an *Interoperabilität* benötigt. Beide Eigenschaften sind für komponentenbasierte Systeme nicht zwingend erforderlich.

- Wie Marquardt bereits andeutet, ist die *Wiederverwendbarkeit* von Plugins auf die als Basis dienende Plugin-Plattform beschränkt. Im Rahmen der CBSE mag dies noch akzeptabel sein, wenn man eine zentrale, generische Plugin-Verwaltung als Komponentenmodellimplementierung ansieht – eine Austauschbarkeit des zugrundeliegenden Komponentenmodells wird auch in der CBSE nicht verlangt. Für die domänenspezifischen Plugin-Schnittstellen vieler Anwendungen scheint diese Sichtweise aber nicht angebracht, da die Plugin-Verwaltung der Anwendungsfunktionalität untergeordnet ist. Somit ist die Wiederverwendbarkeit anwendungsspezifischer Plugins auf die Integration in die jeweilige Anwendung beschränkt (eine Ausnahme stellt die in Unterabschnitt 3.1.1 vorgestellte NPAPI dar, die von mehreren Anwendungen unterstützt wird). Anders als in der CBSE, in welcher ein Handel mit Komponenten zwischen Softwareentwicklern angestrebt wird, ist der Einkauf eines fremden Plugins durch einen Softwareentwickler aufgrund der fehlenden Wiederverwendbarkeit nicht sinnvoll.

- Viele Autoren betonen, dass Komponenten nach Möglichkeit auf statische Abhängigkeiten zu anderen Komponenten verzichten sollten [z. B. Szy02, S. 40]. Die generischen Plugin-Plattformen wie Eclipse (siehe Abschnitt 3.2) oder Mozilla (siehe Abschnitt 3.1) hingegen verdanken ihren Erfolg vor allem der Tatsache, dass sie über die zentrale Plugin-Verwaltung hinaus eine umfassende Softwarearchitektur für grafische Benutzungsoberflächen beinhalten. Viele Plugins sind mit umfangreichen statischen Abhängigkeiten zu den Plugins versehen, welche die Benutzungsoberfläche bereitstellen.

  Korrekterweise müssen diese Plugin-Plattformen daher nicht nur als Komponentenmodellimplementierung, sondern auch als vollständige Komponenten-In-

frastrukturen betrachtet werden. Da die meisten Eclipse-Plugins und Mozilla-Extensions eng in die jeweilige Komponenten-Infrastruktur eingebunden sind, ist deren Wiederverwendbarkeit entsprechend eingeschränkt. Die auch von Wolfinger et al. getroffene Aussage, Plugins hätten nur eine geringe Menge an Abhängigkeiten zu anderen Komponenten, relativiert sich, wenn man alle Komponenten der Infrastruktur als einzelne Elemente im Rahmen eines Komponentenmodells betrachtet.

Insbesondere die eingeschränkte Wiederverwendbarkeit der Plugins lässt die Frage aufkommen, ob Plugin-Systeme noch komponentenbasiert genannt werden dürfen.

Lau und Wang haben in [LW07] eine Taxonomie von Komponentenmodellen basierend auf den Kompositionsmöglichkeiten präsentiert. In Tabelle 2.1 ist zunächst ihre Einordnung maßgeblicher Komponentenmodelle in die vier von ihnen erkannten Kategorien wiedergegeben. Für den Zweck dieser Arbeit hat der Autor die kursiv gesetzten

| Kategorie | untersuchte Modelle | Entwurf | | | | Einsatz |
|---|---|---|---|---|---|---|
| | | Ablage (neu) | Abruf | Komposition | Ablage (kombi) | Komposition |
| Entwurf ohne Repository | ADLs, UML2 u.a. | | | X | | |
| Entwurf mit Nur-Ablege-Repository | EJB, COM, .NET, CCM, WebServices, *XPCOM* | X | | X | | |
| Einsatz mit Repository | JavaBeans | X | | | | X |
| Entwurf mit Repository | SOFA u.a. | X | X | X | X | |
| *anwendungsintegrierte Plugin-Systeme* | *Netscape-Plugins, Mozilla-Chrome* | | | | X (a) | X (b) |
| *generische Plugin-Systeme* | *Eclipse, MagicBeans, NetBeans, XPInstall* | X | | X (a) | X (a) | X (b) |

(a) statische Abhängigkeiten (siehe Text), (b) beschränkt durch Vorgaben zur Entwurfszeit (siehe Text)

Tabelle 2.1: Klassifizierung von Komponentenmodellen nach Kompositionsmöglichkeiten [nach LW07, S. 719], ergänzt um Plugin-Systeme

Zeilen ergänzt, um Plugin-Systeme mit den Komponentenmodellen in Beziehung bringen zu können.

Die vier von Lau und Wang erkannten Kategorien sind: Beim *Entwurf ohne Repository* werden im Entwurf sämtliche Komponenten von Grund auf neu spezifiziert und komponiert, es gibt keine Sammlung bereits existierender Komponenten, zu der daher auch keine komponierten Komponenten hinzugefügt werden können. Beim *Entwurf mit Nur-Ablage-Repository* werden atomare Komponenten in einem Repository abgelegt. Komposition von Komponenten ist möglich, dabei kann aber weder auf Komponenten des Repositories Bezug genommen noch die komponierten Resultate darin abgelegt werden. Beiden eben genannten Kategorien ist gemein, dass die beim Entwurf spezifizierten Komponentenkompositionen zwar in der Einsatzphase realisiert werden, aber keine weiteren Kompositionen in dieser Phase erstellt werden können. Beim *Einsatz mit Repository* können zur Entwurfszeit keine Kompositionen definiert werden, da dieses erst in der Einsatzphase erfolgt. Das Repository kann daher in der Entwurfsphase nur atomare Komponenten aufnehmen und diese in der Einsatzphase zur Verfügung stellen. Beim *Entwurf mit Repository* steht das Repository in der Entwurfsphase voll zur Verfügung, so dass atomare Komponenten abgelegt, zwecks Komposition abgerufen und die komponierten Komponenten wieder abgelegt werden können. Allerdings erlauben auch die Komponentenmodelle dieser Kategorie keine Definition neuer Kompositionen zur Einsatzzeit.

Die Einordnung von Plugin-Systemen in diese Taxonomie gestaltet sich recht schwierig, da Plugins auf eine speziellere Weise als Komponenten komponiert werden. Das Ergebnis einer Plugin-Komposition während der Einsatz- oder Laufphase ist niemals ein neues Plugin, sondern immer ein ganzes Softwaresystem. Daher benötigen Plugin-Systeme kein Repository, welches komponierte Plugins aufnehmen könnte. Stattdessen dienen Repositories in Plugin-Systemen dazu, dem Anwender in der Einsatzphase oder zur Laufzeit die verfügbaren Plugins darzubieten. Diese Argumentation spricht dafür, Plugin-Systeme in die Kategorie *Einsatz mit Repository* einzuordnen.

In den meisten Plugin-Systemen (in Kapitel 3 sind nur die Netscape-Plugins und Mozilla-Extensions als Ausnahmen zu nennen) werden statische Abhängigkeiten zwischen Plugins *explizit* definiert, da sie dem Charakter der Erweiterungsbeziehung geschuldet sind (siehe Unterabschnitt 3.4.2). Solche Abhängigkeiten sollen idealerweise in Komponentensystemen nicht vorkommen [vgl. Szy02, S. 40], sondern sind ein wesentliches Element der modularen Softwareentwicklung, welche Plugin-Architekturen ebenfalls zugrunde liegt. Die Deklaration statischer Abhängigkeiten schränkt die Kompositionsmöglichkeiten von Plugins stark ein, erleichtert aber die automatische Konfiguration des komponierten Gesamtsystems zur Einsatz- bzw. Laufzeit. Eine zur Entwurfszeit vorgenommene Einschränkung mittels statischer Abhängigkeiten auf eindeutig benannte Plugins, die im System zur Einsatz- bzw. Laufzeit vorhanden sein müssen, kommt einer Komposition sehr nahe, da das abhängige Plugin nur in Kombination mit dem bereitstellenden Plugin einsetzbar ist. Das Ergebnis dieser „Komposition“ ist allerdings kein neues „komponiertes“ Plugin, sondern eine feste Eigenschaft des atomaren, abhängigen Plugins. Die Betrachtung der Abhängigkeitsdeklaration als Kompositionsoperation erscheint dennoch gerechtfertigt: Ein ähnliches Ergebnis (hinsichtlich der Gesamtfunktionalität des Systems) ließe sich erzielen, indem eine echte Komposition des erweiterbaren Plugins mit seinem erweiternden Plugin zu einer kom-

binierten (=erweiterten) Komponente durchgeführt wird, welche dann im System zum Einsatz kommt.

Plugin-Systeme unterstützen demnach eine Komposition, deren *Möglichkeit* in der Entwurfsphase festgelegt wird, über deren *Durchführung* aber erst in der Einsatzphase entschieden wird.

In Unterabschnitt 4.1.9.1 wird die Diskussion der Gemeinsamkeiten und Unterschiede der Konzepte der Komponente und des Plugins erneut aufgegriffen werden. Dort wird im Anschluss an die Definition des Plugin-Begriffs, wie er in dieser Arbeit verstanden wird, ein Abgleich mit allen architekturprägenden Konzepten, wie sie in diesem und den anderen Abschnitten dieses Kapitels vorgestellt werden, stattfinden.

## 2.3 Beschreibung von Softwarearchitekturen

In den neunziger Jahren des 20. Jahrhunderts entwickelte sich eine Disziplin, welche die Beschreibung von Softwarearchitekturen zum Thema hat. Als Technik zur Modellierung und Spezifikation von Softwarearchitekturen wurden Architekturbeschreibungssprachen (*Architecture Description Language, ADL*) entwickelt.[14] Die Technik zur Beschreibung von Softwarearchitekturen ist im Kontext dieser Arbeit aus drei Gründen relevant. Zum einen erfordert es die Modellierung zentraler Konzepte von Plugin-Systemen, die Softwarearchitektur der Systeme ausschnittsweise zu betrachten. Daher ist zu untersuchen, ob sich existierende ADLs zur Modellierung dieser Konzepte eignen. Zum anderen weist das Gebiet der Beschreibung von Softwarearchitekturen eine zeitliche und konzeptionelle Nähe zum in Abschnitt 2.2 vorgestelltem Paradigma der komponentenbasierten Softwareentwicklung (CBSE) auf. Das Vokabular von Architekturbeschreibungen erweitert somit den Blickwinkel auf komponentenbasierte Softwarearchitekturen. Zum dritten wird in dieser Disziplin der Begriff des Architekturstils oder -musters definiert, welcher – auf einer höheren Abstraktionsebene – dem in Abschnitt 2.4 vorgestellten Konzept des Entwurfsmusters entspricht.

Zu Beginn des Kapitels werden in Unterabschnitt 2.3.1 zunächst einige Grundkonzepte eingeführt, welche bei der Betrachtung von Softwarearchitekturen verwendet werden. Daran anschließend werden in Unterabschnitt 2.3.2 die Charakteristika und Leistungsfähigkeit von Architekturbeschreibungssprachen überblicksweise erläutert. Um die in später in Kapitel 4 praktizierte Modellierung einer dynamischen Softwarearchitektur mit Petrinetzen einordnen zu können, werden drei Themenschwerpunkte aus dem Gebiet herausgegriffen: Erstens wird in Unterabschnitt 2.3.3 die Eignung der weit verbreiteten Unified Modeling Language (UML) zur Beschreibung von Softwarearchitekturen untersucht. Zweitens wird in Unterabschnitt 2.3.4 das Gebiet der dynamischen ADLs beleuchtet, also jener ADLs, welche dynamische Veränderungen in einer Softwarearchitektur beschreiben können. Und drittens wird speziell auf ADLs eingegangen, welche Petrinetze als Ausdrucksmittel verwenden (siehe Unterabschnitt 2.3.5).

[14] Die Technik einer Architekturbeschreibungssprache wird nicht nur im Bereich von Softwarearchitekturen, sondern auch von Hardwarearchitekturen verwendet. Im Rahmen dieser Arbeit ist aber nur ersterer Bereich von Interesse, da das Einsatzszenario für Hardware-ADLs stark von dem einer Softwarearchitektur abweicht.

Die Einordnung der in dieser Arbeit verwendeten Modellierungstechnik in die hier vorgestellten Grundlagen wird später in Unterabschnitt 4.5.4 stattfinden.

### 2.3.1 Grundbegriffe

Der Begriff der Architektur lässt sich auf vielen Ebenen einsetzen. Als Grundlage für Architekturbeschreibungssprachen gilt ein Verständnis von Softwarearchitektur, wie es Shaw und Kaplan in [SG96] umschreiben. Eine knappe auf Shaw und Garlan basierende Definition des Begriffs geben Councill und Heinemann:

> „*Software architecture* is a level of design that specifies the overall system structure of a software application.“ [HC01, S. 368]

Es geht also um die Beschreibung der übergreifenden Struktur eines Softwaresystems. Taylor, Medvidović und Dashofy definieren den Begriff der Softwarearchitektur noch allgemeiner als „Menge grundlegender Entwurfsentscheidungen“[15] bezüglich eines Softwaresystems. Die eine Softwarearchitektur ausmachenden Entwurfsentscheidungen betreffen alle Aspekte eines Systems, unter anderem seinen strukturellen Aufbau, das Verhalten seiner Funktionalität und die Festlegung zu verwendender Kommunikationstechniken sowie nichtfunktionaler Eigenschaften und Implementierungstechniken. Ob eine Entwurfsentscheidung grundlegend genug ist, um als Teil der Softwarearchitektur zu gelten, ist dabei von Kontext und Betrachtungswinkel der Personen abhängig, die sich mit dem System beschäftigen. [Vgl. TMD10, S. 58f.]

Die Disziplin der Beschreibung von Softwarearchitekturen geht mit einer abstrakten Sichtweise einher, aus der heraus Paradigmen wie die objektorientierte oder komponentenbasierte Softwareentwicklung als *Architekturstile* betrachtet werden können. Shaw und Garlan führen in [SG96] den Begriff des Architekturstils (*architectural style*, *architectural idiom* oder *architectural pattern*) ein und bezeichnen damit die beim Entwurf von Softwarearchitekturen verwendeten idiomatischen Muster zur Organisation von Systemen [vgl. SG96, S. 19f.]. Die Begriffe des Architekturstils und des Architekturmusters liegen nahe beieinander. Einige Autoren gehen sogar soweit, die Begriffe nicht zu unterscheiden [vgl. Gor06, S. 5]. Taylor, Medvidović und Dashofy unterscheiden die Begriffe nach Größe und Abstraktionsebene des Anwendungsgebietes – ein Architekturstil beinhaltet grundlegendere, allgemeinere Vorgaben als ein auf Lösung eines konkreten Problems ausgerichtetes Architekturmuster [vgl. TMD10, S. 73]. Der ähnliche Begriff des Entwurfsmusters, der in Abschnitt 2.4 besprochen wird, unterscheidet sich vom Architekturmuster abermals durch das zugeordnete Anwendungsgebiet. Entwurfsmuster werden vorwiegend zur Lösung wiederkehrender Problemstellungen im objektorientierten Entwurf eingesetzt, so dass sie in der Regel kleinere Ausschnitte eines Softwaresystems als ein Architekturmuster behandeln. Die Grenzen sind allerdings fließend. Die Einordnung von Plugin-Systemen mit Bezug auf Entwurfsmuster ist Gegenstand des Abschnitts 2.4. Da dort Muster auf mehreren Abstraktionsebenen betrachtet werden, sollen Entwurfsmuster ebenso wie Architekturmuster in diesem Abschnitt hier nicht weiter diskutiert werden.

[15] *A software system's* architecture *is the set of principal design decisions made about the system.* [TMD10, S. 58f.]

Im von der ISO übernommenen IEEE-Standard 1471 [IEE00, ISO07, Abschnitt 4] werden verschiedene Einsatzzwecke für Architekturbeschreibungen und mögliche Rollen beteiligter Personen ausgemacht. Demnach dienen Architekturbeschreibungen unter anderem der Dokumentation und der Kommunikation zwischen Entwicklern, Anwendern, Auftraggebern oder Gutachtern eines Softwaresystems während seiner Entstehung, Weiterentwicklung, nachträglichen Analyse oder Bewertung. Die meisten Architekturbeschreibungssprachen tragen diesem großen Nutzerkreis nur bedingt Rechnung, da sie auf einen Schwerpunkt im Bereich der formalen Spezifikation von Softwarearchitekturen hin ausgerichtet sind. Diese Ausrichtung resultiert aus dem Wunsch, die Eigenschaften eines Systems exakt spezifizieren und die Konformität der Systemimplementierung verifizieren zu können.

Kruchten stellt in [Kru95] fest, dass es nicht sinnvoll sei, alle Aspekte einer Architektur auf einmal erfassen zu wollen. Er schlägt daher die Verwendung von 4+1 Sichten auf das System vor, so dass in jeder Sicht eine Fokussierung auf bestimmte Aspekte des Systems erfolgen kann. Zudem kann zu jeder Sicht eine jeweils passende grafische Notation und ein anderer Architekturstil verwendet werden. Die Aufteilung in Sichten ermöglicht somit auch die Verwendung mehrerer Architekturstile innerhalb eines Softwaresystems.

Kruchten unterscheidet die logische, prozessorientierte, physische und Entwicklungssicht, die durch die fünfte Sicht der Szenarien konsistent gehalten werden. Die logische Sicht modelliert die funktionalen Anforderungen an das System und wird beispielsweise durch ein objektorientiertes Klassendiagramm dargestellt. Die prozessorientierte Sicht dient der Erfassung von Nebenläufigkeit, Verteilung, Performanz oder Fehlertoleranz, gruppiert zusammenhängende Aufgaben zu Prozessen und beschreibt die Kommunikation zwischen diesen. Die Entwicklungssicht bringt softwaretechnische Prinzipien in die Architektur, beispielsweise eine schichtenweise Organisation der einzelnen Teile. Die physische Sicht erfasst ähnliche nichtfunktionale Eigenschaften wie die prozessorientierte Sicht, setzt den Schwerpunkt aber auf die Verteilung der Software auf Hardware-Einheiten. Dabei kann die Kommunikation wie in der Prozesssicht mit eingearbeitet werden. Szenarien entsprechen den heute aus der UML bekannten Use-Case-Diagrammen. Sie setzen die Elemente der anderen vier Sichten zueinander in Beziehung und dienen dem Entdecken notwendiger Architekturbestandteile. [Vgl. Kru95, S. 43–47]

Bass greift in [Bas01, Abschnitt 21.9, S. 400] Kruchtens Sichten auf, verallgemeinert aber die Prozesssicht zur dynamischen Sicht, indem er die Fixierung auf Betriebssystemprozesse fallen lässt. Stattdessen setzt er die Prozesssicht allgemein zur Beschreibung von Mehrbenutzersystemen, Ressourcenverwaltung oder Initialisierungsphasen ein. Bass betont zudem, dass die verschiedenen Sichten keine statischen Beschreibungen einer Architektur darstellen, sondern sich während des Systementwurfs dynamisch entwickeln.

Wie dynamische Veränderungen der beschriebenen Softwarearchitektur in den Sichten erfasst werden können, erwähnt Bass eben so wenig wie Kruchten. Bass berücksichtigt aber das Qualitätsmerkmal der *Modifizierbarkeit* einer Architektur, um beispielsweise die Architektur einer Komponenten-Infrastruktur (siehe Unterabschnitt 2.2.1) beschreiben zu können. Modifizierbarkeit lässt sich beispielsweise durch geeignete Aufteilung von Funktionalität erreichen und erlaubt so das Verwenden einer einzigen Ar-

chitektur für mehrere Softwaresysteme im Rahmen einer Produktfamilie. [Vgl. Bas01, S. 389 u. S. 392]

### 2.3.2 Eigenschaften von Architekturbeschreibungssprachen

Es gibt viele verschiedene Architekturbeschreibungssprachen (ADLs). Die Gemeinsamkeiten von ADLs lassen sich am besten über einheitliche Anforderungen und häufig verwendete Begriffe zur Beschreibung von Softwarearchitekturen identifizieren. Die verschiedenen ADLs sind sehr unterschiedlich in ihrer Leistungsfähigkeit.

**Anforderungen** Shaw und Garlan erarbeiten in [SG96] folgende Anforderungen für Architekturbeschreibungssprachen:

**Komposition** (*Composition*) Diese Fähigkeit erlaubt dem Designer einer Softwarearchitektur die hierarchische Zerlegung eines komplexen Systems in einzelne Komponenten (*components*) und Verbinder (*connectors*). Dabei sollte die Sprache es erlauben, die Einzelteile isoliert vom Gesamtsystem verstehen zu können. Implementierungsdetails wie Algorithmen oder Datenstrukturen sollten unabhängig von der Systemstruktur modelliert werden können.

**Abstraktion** (*Abstraction*) Eine ADL muss die Beschreibung von Komponenten und ihren Interaktionen auf eine Weise ermöglichen, die ihre abstrakten Rollen im System klar und explizit festlegt.

**Wiederverwendbarkeit** (*Reusability*) Sowohl Komponenten als auch Verbinder sollten in anderen Architekturbeschreibungen wiederverwendet werden können. Gleiches sollte auch für komplexe Architekturmuster gelten.

**Konfiguration** (*Configuration*) Die Struktur eines Systems sollte durch eine ADL auch ohne Betrachtung der einzelnen Komponenten verständlich und veränderlich darstellbar sein. Hierzu gehört auch die Unterstützung dynamischer Rekonfigurierbarkeit, da beispielsweise die dynamische Erzeugung von Objekten oder Prozessen in laufenden Systemen gang und gäbe ist.

**Heterogenität** (*Heterogenity*) Eine ADL sollte die Kombination unterschiedlicher Architekturstile innerhalb eines einzelnen Systems, beispielsweise einer Komponente, zulassen. Ebenso sollte eine ADL die Kombination von Komponenten mit unterschiedlichen Implementierungssprachen erlauben.

**Analyse** (*Analysis*) Eine Analyse von beschriebenen Architekturen sollte im Hinblick auf vielfältige Ziele und Probleme möglich sein. Shaw und Garlan merken an, dass vermutlich kein einzelner Formalismus für alle verschiedenen Analysewünsche ausreichen wird, sondern dass eine ADL die Annotation von Spezifikationen an einzelnen Komponenten, Verbindern oder Mustern erlauben sollte.

[Vgl. SG96, Abschnitt 7.1.2, S. 151ff.]

Diese Anforderungen beschreiben die Modellierungsfähigkeiten einer Architekturbeschreibungssprache auf abstrakter Ebene. Wird eine Architekturbeschreibungssprache

durch Werkzeuge oder eine Ausführungsumgebung unterstützt, so kann auch von einem *Architektur-Rahmenwerk* gesprochen werden. Dieser Begriff kombiniert die Anforderungen an die ADL mit den Anforderungen an ein Komponentenmodell oder eine Komponentenmodellimplementierung (siehe Unterabschnitt 2.2.1). Ein Architektur-Rahmenwerk ist meist auf einen bestimmten Architekturstil festgelegt. V. d. Aalst et al. haben in [ABH+07b, Abschnitt 2.3, S. 9f] einen Anforderungskatalog für Architektur-Rahmenwerke zusammengetragen, der z. B. die Vorgabe konkreter Eigenschaften von Komponenten wie Fehlerbehandlungsmechanismen oder Benutzungs- und Beobachtungsschnittstellen umfasst. Für die Betrachtung im Rahmen dieser Arbeit reichen die abstrakten Anforderungen an ADLs aus, da die konkreteren Anforderungen bereits im Rahmen der CBSE (siehe Abschnitt 2.2) behandelt werden.

**Vokabular und Abgrenzung** Medvidović und Taylor erarbeiten in [MT00] eine Charakterisierung von ADLs und ordnen bekannte ADLs in diese Charakterisierung ein. Eine ADL muss demnach *Komponenten*, *Verbinder* (*connectors*) sowie deren *Konfiguration* bzw. *Topologie* (*architectural configurations*) explizit modellieren. Darüber hinaus muss die ADL, um wirklich verwendbar und nützlich zu sein, *Werkzeugunterstützung* zur architekturbasierten Entwicklung und Evolution bieten. Eine ADL zeichnet sich also dadurch aus, dass sie Möglichkeiten zur Modellierung der *konzeptionellen* Architektur eines Softwaresystems (zu unterscheiden von dessen *Implementierung*) bietet. ADLs bieten damit sowohl eine konkrete Syntax als auch ein konzeptionelles Rahmenwerk zur Beschreibung von Architekturen.

Eine *Komponente* ist im Kontext einer ADL eine rechnende oder zustandsbehaftete Einheit, sie wird unbedingt mit expliziten Schnittstellen modelliert. Daneben kann eine ADL die Modellierung von Komponenten-Typen, -Semantik, -Eigenschaften bzw. -Zusicherungen und -Evolution unterstützen. Ebenfalls denkbar ist eine Unterstützung der Modellierung nichtfunktionaler Eigenschaften von Komponenten. Die Größe einer Komponente kann zwischen Prozedur und Anwendung variieren [vgl. MT00, Abschnitt 3.1.1, S. 74]. Der Begriff der Komponente in der Beschreibung einer Softwarearchitektur ist sowohl allgemeiner als auch spezieller gefasst als die in Unterabschnitt 2.2.1 genannten Definitionen. Zum einen ist der Komponentenbegriff in der Beschreibung von Softwarearchitekturen nicht auf eine bestimmte Granularitätsebene festgelegt – auch eine einzelne Operation kann als Komponente betrachtet werden. Zum anderen beschränkt sich der Betrachtungswinkel dieses Komponentenbegriffs auf die (technische) Komposition von Komponenten im System, es fehlen z. B. geschäftliche Aspekte wie im Bereich der CBSE. Taylor, Medvidović und Dashofy erläutern in [TMD10, S. 70] ihre Sicht des Unterschieds.

Ein *Verbinder* modelliert Interaktionen bzw. Interaktionsregeln zwischen Komponenten. Daher müssen Verbinder nicht zwangsläufig kompilierbaren Einheiten im Softwaresystem entsprechen – beispielsweise wird auch die Technik des Prozeduraufrufs als Verbinder gesehen. Verbinder werden ebenso wie Komponenten in ADLs durch Schnittstellen, Typen, Semantik, Eigenschaften bzw. Zusicherungen, Evolution und nichtfunktionale Eigenschaften charakterisiert [vgl. MT00, Abschnitt 3.1.2, S. 74f.]. Szyperski kritisiert in [Szy02, Abschnitt 21.1.2, S. 428f.], dass die Unterscheidung zwischen Verbindern und Komponenten eine künstliche sei, da Verbinder wie Komponen-

ten Funktionalität kapseln und eine detailliertere Dekomposition sinnvoll erscheinen lassen können. Taylor, Medvidović und Dashofy betonen dagegen in [TMD10, Abschnitt 3.1.3, S. 70f.] die Anwendungsunabhängigkeit von Verbindern im Gegensatz zu Komponenten. Die zentrale Aufgabe von Verbindern – das Ermöglichen und Regeln der Interaktion zwischen Komponenten bzw. die Festlegung von Kontroll- und Datenfluss – sei ein ganz anderer Belang als die funktionalitäts- oder datenorientiert definierten Aufgaben von Komponenten. Das Trennen dieser Belange befördere die einfache Anpassbarkeit einer Softwarearchitektur an geänderte Bedürfnisse.

Eine *Konfiguration* oder *Topologie* ist ein zusammenhängender Graph von Komponenten und Verbindern, der die Struktur der Architektur beschreibt. Zur Charakterisierung einer ADL ist von Interesse, welche Konfigurationsbeschreibungsfähigkeiten die ADL mitbringt sowie welche Qualitäten und Eigenschaften des beschriebenen Systems modelliert werden können. Konkret wird die Verständlichkeit der Beschreibungen, die Kompositionalität der Sprache, die Verfeinerbarkeit (und deren Nachvollziehbarkeit) der Beschreibungen, die Eignung für heterogene Einsatzgebiete, die Skalierbarkeit, Evolutionsfähigkeit und Dynamik der beschriebenen Systeme, sowie die Darstellbarkeit notwendiger funktionaler und nichtfunktionaler Eigenschaften der beschriebenen Systeme betrachtet.

Medvidović und Taylor grenzen den Begriff der Architekturbeschreibungssprache gegen formale Spezifikationssprachen, Modulverbindungssprachen (*module interconnection language, MIL*), Simulationssprachen, objektorientierte Entwurfssprachen und Programmiersprachen ab. Ein wesentliches Merkmal einer ADL ist demnach die explizite Betrachtung von Verbindern als erstrangigen Elementen. In den anderen genannten Sprachgruppen werden Verbindungen zwischen Komponenten, Objekten, Modulen etc. nicht als eigenständige Elemente mit variablen Eigenschaften modelliert oder gar nur implizit verwendet. Ein zweites Merkmal von ADLs ist die Einbeziehung des Verhaltens sowie nichtfunktionaler Eigenschaften von Komponenten und Verbindern. Dieses Merkmal dient insbesondere zur Abgrenzung gegen MILs, da eine MIL lediglich die Struktur eines komplexen, aus verbundenen Komponenten bestehenden Systems beschreibt. [Vgl. MT00, Abschnitt 3, S. 73–78]

**Leistungsfähigkeit und Schwerpunkte** Die auch in der eingangs wiedergegebenen Definition des Softwarearchitekturbegriffs zu bemerkende Fokussierung auf strukturelle Aspekte soll durch ADLs genau *nicht* betrieben werden. Stattdessen ergibt sich aus den Anforderungen und dem üblichen Vokabular einer ADL, dass Verhaltensaspekte von und Verbindungen zwischen strukturellen Elementen in einer ADL erfassbar sein sollen. Dennoch ordnet beispielsweise der IEEE-Standard zur Beschreibung von Softwarearchitekturen ADLs als Ausdrucksmittel für die strukturelle Sicht auf ein System ein [siehe IEE00, ISO07, Anhang C.1].

Medvidović und Taylor stellen in [MT00, Abschnitt 4.2, S. 80] fest, dass einige ADLs Verbinder nicht als erstrangige Modellierungselemente unterstützen. Zudem kritisieren sie in [MT00, Abschnitt 4.4.2, S. 87], dass die zu den ADLs angebotenen Werkzeuge nur in wenigen Fällen mehr als eine Sicht bieten, die auf zwei Weisen – einmal textuell und einmal grafisch – dargestellt werden kann. Die Beschränkung auf nur eine Sicht führt zu dem Eindruck, dass die eigentlich als gleichrangig geforderte Modellierung von

Verhalten in den existierenden ADLs nur der formalen Verifizierbarkeit der Korrektheit einer Architektur dient und in der grafischen Modellierung nicht wirklich gleichrangig behandelt wird.

Das Ziel einer formalen, präzisen und vollständigen Semantik führt dazu, dass die meisten ADLs eine Modellierung unvollständiger Architekturen als fehlerhaft betrachten und ausschließen [vgl. MT00, Abschnitt 4.3.6, S. 85]. Die dynamische Veränderung einer Softwarearchitektur unterstützen nur wenige der von Medvidović und Taylor untersuchten ADLs. Von denen erlauben wiederum einige nur im Voraus spezifizierte Änderungen an einer Softwarearchitektur [vgl. MT00, Abschnitt 4.3.7, S. 86]. Die Thematik von ADLs für dynamisch veränderliche Architekturen wird in Unterabschnitt 2.3.4 noch einmal aufgegriffen werden.

Stafford und Wolf kommentieren in [SW01] den Entwicklungsstand von Architekturbeschreibungssprachen. Trotz einer bereits über 20 Jahre andauernden Forschung befinden sich existierende ADLs in sehr unterschiedlichen Reifegraden, variieren in ihrer Ausdrucksfähigkeit und fokussieren stets nur einzelne Aspekte der breiten Anforderungspalette [vgl. SW01, Abschnitt 20.3.1, S. 379f.]. Ähnlich fällt die Diagnose von Medvidović und Taylor aus, welche die sehr unterschiedlichen Modellierungsmöglichkeiten für Verbinder sowie die unvollständige Unterstützung der Modellierung von nichtfunktionalen Eigenschaften oder von Dynamik bemängeln [vgl. MT00, S. 91]. Gorton verweist in [Gor06, S. 9] darauf, dass ADLs zwar einen gewissen Einfluss auf die Diskussion von Softwarearchitektur haben, aber im täglichen Einsatz kaum eine Rolle spielen.

Medvidović, Dashofy und Taylor sehen in [MDT07] eine Ursache für den mäßigen Erfolg von Architekturbeschreibungssprachen in der zu technischen Ausrichtung der ADLs. So sei die Unified Modeling Language (UML) flexibler einsetzbar. Sie habe sich zudem schneller weiter entwickelt, so dass sie sich trotz ihrer Unzulänglichkeiten im Bereich der Beschreibung von Softwarearchitekturen (siehe folgenden Unterabschnitt) durchgesetzt habe.

### 2.3.3 UML als Architekturbeschreibungssprache

Die inzwischen in Version 2.0 spezifizierte Unified Modeling Language (UML, siehe [OMG05a] und [OMG05b]) ist nicht unmittelbar als ADL konzipiert, sondern als eine – wie der Name schon sagt – für viele Zwecke geeignete, vereinheitlichte Modellierungssprache. Die „Vereinheitlichung“ besteht allerdings im Wesentlichen aus einer Kombination mehrerer zuvor unabhängiger Modellierungstechniken, welche nun zwar eine einheitliche textuelle und grafische Notation nutzen, aber semantisch nur lose zusammenhängen.

Die UML definiert zunächst ein Meta-Modell, welches dann zur Definition verschiedener Diagramme herangezogen wird. Die Bereitstellung verschiedener Diagrammarten zur Modellierung verschiedener Aspekte eines Systems kann im Sinne des Sichtenansatzes von Kruchten genutzt werden. Allerdings bietet die UML keine Hilfsmittel zur Wahrung der Konsistenz zwischen mehreren Sichten auf ein System. [vgl. MDT07, Abschnitt 4.1].

Taylor, Medvidović und Dashofy beschränken sich zum Zweck der Einordnung der UML als ADL in [TMD10, S. 203ff.] auf die Vorstellung einiger UML-Diagramme am

Beispiel. Sie diskutieren begleitend die gewollte Mehrdeutigkeit von UML-Diagrammen und die Möglichkeit der Präzisierung durch UML-Profile. Ein Profil definiert Stereotypen, welche zur Konkretisierung der Semantik von einzelnen Elementen eines UML-Diagramms eingesetzt werden können.

Einige der von UML bereitgestellten Diagramme eignen sich besonders zur Beschreibung einzelner Aspekte einer Softwarearchitektur im Sinne einer ADL. Eine kurze Einführung, wie die Neuerungen von UML 2.0 zur Modellierung von Softwarearchitekturen eingesetzt werden könnten, geben Björkander und Kobryn in [BK03].

Ein besonderes Augenmerk im Kontext der Beschreibung von Architekturen verdienen insbesondere die in der Version 2 erweiterten Komponentendiagramme. Neben den Komponentendiagrammen existieren in der UML 2.0 mit den Deployment-Diagrammen und den Montage-Diagrammen zwei weitere Sichten auf die Architektur eines Softwaresystems. Die in der UML 2.0 neu eingeführten Montagediagramme [siehe OMG05b, Kapitel 9, S. 157ff.] bieten ähnliche Ausdrucksmöglichkeiten wie Komponentendiagramme, wobei die innere Struktur eines Systems oder einer Klasse auf der Basis von Klassen anstelle von Komponenten dargestellt wird. Montagediagramme werden daher im Folgenden nicht explizit vorgestellt.

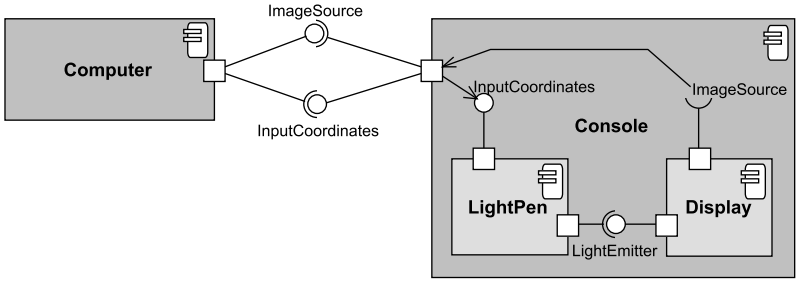
Deployment-Diagramme [siehe OMG05b, Kapitel 10, S. 189ff.] beschreiben den physischen Aspekt einer Softwarearchitektur, also die Aufteilung von Artefakten wie z. B.-Dateien oder Datenbanktabellen auf physische Geräte oder Knoten einer Software-Infrastruktur wie z. B. einer Komponentenmodellimplementierung. In einem Deployment-Diagramm sind insbesondere die Kommunikationsverbindungen zwischen und die hierarchische Einbettung von Geräten bzw. Knoten darstellbar. Ein Bezug zur eigentlichen Softwarearchitektur lässt sich nur in Form von Abhängigkeiten mit dem Stereotyp `«manifest»` ausdrücken. Eine `«manifest»`-Beziehung setzt Elemente des Softwareentwurfs wie z. B. Komponenten oder Klassen mit dem Artefakt, welches diese Elemente beinhaltet, in Beziehung. Ein Deployment-Diagramm stellt einen Schnappschuss der Verteilung physischer Komponenten eines Softwaresystems im Einsatz dar und bietet keine Ausdrucksmöglichkeit für dynamische Veränderungen. Es trägt somit zur Beschreibung abstrakter Softwarearchitekturen nur eine sehr eingeschränkte Sicht bei und wird im Folgenden nicht weiter behandelt.

**Komponentendiagramme** Als Grundlage der Komponentendiagramme [OMG05b, Kapitel 8] wird zunächst der Komponentenbegriff wie folgt definiert:

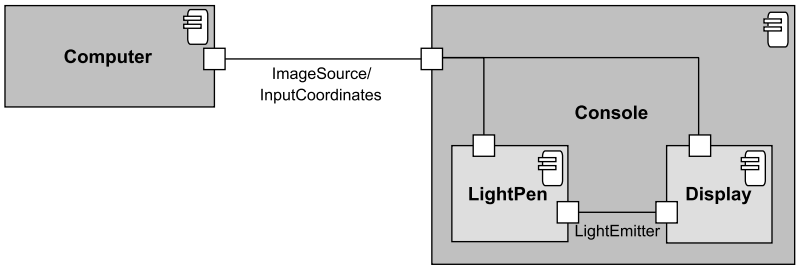
> „In particular, the [Components] package specifies a component as a modular unit with well-defined interfaces that is replaceable within its environment.“ [OMG05b, S. 139]

Der Komponentenbegriff wird als Spezialisierung des Klassenbegriffs gesehen, eine genaue Abgrenzung steht noch aus [vgl. Stö05, S. 117]. Eine Komponente kann mit ihren angebotenen und benötigten Schnittstellen (*interfaces*) und Anschlüssen (*ports*) quasi als *Black-Box* spezifiziert werden. Ein Anschluss ist eine Eigenschaft (*property*), ein Verhalten (*behavior*) oder ein Dienst (*service*).[16] Die gelieferten (*provided*) und benö-

[16] Björkander und Kobryn fassen diese Anschlussvarianten als vorsätzliche Lücke in der Kapselung einer Klasse zusammen, durch die Nachrichten hinein, heraus oder in beide Richtungen geschickt werden können. [Vgl. BK03, S. 58]




(a) Explizite Darstellung der Interfaces




(b) Kompakte Darstellung als Verbinder

Abbildung 2.11: Beispiel für ein UML-Komponentendiagramm

tigten (*required*) Schnittstellen und Anschlüsse können miteinander verdrahtet (*wired*) werden, um die Zusammenarbeit mehrerer Komponenten zu beschreiben. Zudem lässt sich in der *White-Box-Sicht* einer Komponente die Delegation extern spezifizierter Schnittstellen und Anschlüsse einer Komponente an interne Bestandteile der Komponente modellieren. [Vgl. OMG05b, Abschnitt 8, S. 139–156]

Wird ein Anschluss in der White-Box-Sicht als *behavioural port* klassifiziert, lässt sich die Delegation des Verhalten einer Komponente an ihre Einzelteile beschreiben. In der Black-Box-Sicht ist die Verhaltenseigenschaft eines Anschlusses allerdings nicht zu erkennen. [Vgl. BK03, S. 60]

In Abbildung 2.11 ist sowohl eine Komponente in Black-Box-Sicht (links) als auch eine in White-Box-Sicht (rechts) dargestellt. Graue Kästen stellen Komponenten dar und weiße Quadrate bezeichnen Anschlüsse der Komponenten. In Abbildung 2.11(a) sind die Verbindungen zwischen den Anschlüssen explizit durch angebotene (geschlossene Kreise) und genutzte (offene Halbkreise) Interfaces modelliert. Im Beispiel bedeutet dies, dass der nach außen geführte Anschluss der Konsole sowohl ein Interface nutzt (eine Bilddaten-Quelle) als auch eines bereitstellt (eine Bildschirmkoordinaten-Quelle). Die beiden Komponenten können nur verbunden werden, weil ihre Anschlüsse ein passendes Paar an Schnittstellen benötigen und fordern. In der rechten Komponente werden die beiden Schnittstellen des Anschlusses an die Anschlüsse der internen Subkomponenten delegiert. Zwischen den Subkomponenten besteht eine weitere Ver-

bindung, welche nur auf einem Interface basiert. In Abbildung 2.11(b) sind dieselben Verbinder ohne explizite Darstellung der Interfaces modelliert.

Um das Verhalten von Komponentenanschlüssen im Sinne von Software-Verträgen beschreiben zu können, lassen sich Anschlüsse mit *Protocol State Machines* in Form von Statecharts versehen. So können erlaubte Übergänge zwischen Operationen der Schnittstelle nach außen *deklariert* werden. Zusätzlich lassen sich die *Protocol State Machines* zur *exekutiven* Beschreibung aller Ereignisse, Übergänge und Operationen des Anschlusses einsetzen. [Vgl. And03, S. 333]

Es können sowohl logische Komponenten (z. B. Geschäftsobjekte) als auch „physische“ Komponenten mit Bezug auf ein konkretes Komponentenmodell (z. B. Enterprise Java Beans) beschrieben werden. Die Konkretisierung der Diagrammelemente durch ein UML-Profil ist explizit vorgesehen, um das Diagramm z. B. an ein bestimmtes Komponentenmodell anpassen zu können.

Eine dynamische Veränderung der im Komponentendiagramm dargestellten Struktur ist in UML allerdings nicht vorgesehen. Die in der UML spezifizierten dynamischen (oder Verhaltens-)Diagramme eignen sich ebenfalls nicht zur Darstellung von strukturellen Ände1rungen.

**Profile und Stereotypen** Medvidović, Rosenblum, Redmiles und Robbins erarbeiten und vergleichen in [MRRR02] verschiedene Ansätze, wie sich UML als Architekturbeschreibungssprache einsetzen lässt. Dabei führen sie drei verschiedene Ansätze an:

1. die Modellierung mit unverändertem UML, indem Komponenten und Verbinder direkt als Klassen in UML modelliert werden

2. die Einschränkung der Modellierungsmöglichkeiten von UML durch Definition neuer Stereotypen, deren Ausdrucks- und Verwendungsmöglichkeiten in der *Object Constraint Language* (OCL) spezifiziert werden, und

3. die Erweiterung des UML-Meta-Modells um fehlende Konzepte.

Da der dritte Ansatz einer grundlegenden Zielsetzung von UML (der Vereinheitlichung der Modellierungssprachen) zuwiderläuft, wird er von Medvidović et al. nicht weiter verfolgt. Der erste Ansatz eignet sich zur Modellierung der Struktur eines Systems, fordert aber vom Modellierer die Beachtung architekturstilspezifischer Konventionen und büßt durch die Abbildung abstrakter Architekturkonzepte auf konkrete Klassen die Anwendungsunabhängigkeit der Softwarearchitektur ein. Der zweite Ansatz wird veranschaulicht, indem die Ausdrucksmöglichkeiten dreier bekannter, recht unterschiedlicher ADLs (namentlich C2, Wright und Rapide) in UML als Stereotypen eingebracht werden. Durch die Verwendung von Statecharts lässt sich auch das Verhalten von Komponenten und Verbindern bzw. die Kausalität von Ereignissen in UML modellieren. Der Nachteil dieses Ansatzes liegt im stark formalen Charakter der OCL, die nicht allen Modellierern geläufig ist, sowie im optionalen Charakter der OCL, die nicht von allen UML-Werkzeugen unterstützt wird. [Vgl. MRRR02, Abschnitte 1–5, S. 3–46]

Als größtes Problem der UML im Vergleich zu ADLs sehen Medvidović et al. die möglicherweise fehlende Übereinstimmung der zur Verfügung stehenden Model-

lierungsmöglichkeiten mit dem mentalen Modell des Softwarearchitekten und die fehlende Unterstützung für Architekturstile. Wenn die gewünschten Abstraktionen (z. B. Verbinder) fehlen und durch andere Abstraktionen (z. B. Klassen) nachgebildet werden müssen, verkompliziert sich die Arbeit des Architekten. Ebenso bedeutet das Fehlen der Unterstützung für Architekturstile, dass der Architekt selber darauf achten muss, dass die mit dem gewünschten Stil verbundenen Konventionen eingehalten werden. [Vgl. MRRR02, Abschnitte 7.1.4 und 7.1.5, S. 52]

Umgekehrt fehlen den meisten ADLs Modellierungsmöglichkeiten für die fachliche Domäne der Anwendung. Ebenso fehlen den Code-Generierungswerkzeugen der ADLs Informationen über die Funktionalität einzelner Komponenten, so dass kein vollständiges System erzeugt werden kann. [Vgl. MRRR02, Abschnitte 7.1.3 und 7.1.6, S. 51ff.]

Als Beispiel für die Definition eines UML-Profils zu einer ADL sei das von Oquendo in [Oqu06] vorgestellte Profil für die $\pi$-ADL genannt. Die $\pi$-ADL ist zunächst als ArchWare-ADL [MKB+04] definiert worden. Sie baut auf dem $\pi$-Kalkül auf und umfasst eine textuelle ADL mit Schwerpunkt auf der formalen Spezifikation und Verifikation sich mit der Zeit weiterentwickelnder Systeme. Oquendo definiert UML-Stereotypen für Komponenten, Verbinder, Kompositionen, Anschlüsse und Verbindungen. Dabei werden neben Komponenten und Kompositionen auch die Verbinder als stereotypisierte UML-Komponenten modelliert. Die Stereotypen werden in Form von Metaklassen mit typisierten Attributen definiert und mit wiedererkennbaren Icons versehen. Ferner definiert er einen Satz von Verhaltensweisen (z. B. Sequenz, Auswahl, Wiederholung, Rekursion) in Form von einfachen Statecharts. Zur Beschreibung eines Systems werden die stereotypisierten Komponenten und Verbinder gezeichnet und miteinander verdrahtet. In einer Detail-Sicht werden die Attribute der einzelnen Elemente textuell mit Werten belegt, um die Verdrahtung zu konkretisieren.

Oquendo demonstriert so, wie die UML zur Visualisierung der textuellen ADL herangezogen werden kann. Allerdings fällt in seiner Fallstudie die strenge Trennung zwischen dem übersichtlichem Komponentendiagramm und den textlastigen Diagrammen zur Spezifikation einzelner Komponenten auf. Die Verhaltensweisen werden in den Komponenten ausschließlich textuell spezifiziert, eine grafische Komposition komplexer Statecharts findet nicht statt.

**Model Driven Architecture (MDA)** Die Eignung der UML als ADL ist auch deshalb von Interesse, weil die OMG die Softwareentwicklungs-Methodik der „*Model Driven Architecture*“ verfolgt.

> „MDA provides an approach for, and enables tools to be provided for:
>
> - specifying a system independently of the platform that supports it,
> - specifying platforms,
> - choosing a particular platform for the system, and
> - transforming the system specification into one for a particular platform.
>
> The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns.“ [OMG03, S. 2-2]

Obwohl diese Zielsetzung sehr ähnlich zu der von Architekturbeschreibungssprachen zu sein scheint, verfolgt die OMG doch einen eher technisch orientierten Ansatz. Statt einer für Softwarearchitekten geeigneten Ausdrucksmöglichkeit oder einer zur Systemverifikation geeigneten Sprache sucht die OMG eine technische Lösung zur Ausführung verteilter Anwendungen auf heterogenen Plattformen. Das Ziel ist die Bereitstellung maschinenlesbarer Anwendungs- und Datenmodelle, aufbauend auf anderen OMG-Standards wie CORBA und UML, so dass neben der Implementierung auch die Architekturbeschreibung zur Laufzeit zur Verfügung steht.

Bei der MDA wird zwischen plattformunabhängigen (*Platform Independent Model, PIM*), plattformspezifischen (*Platform Specific Model, PSM*) und systemspezifischen (*Enterprise Deployment Model, EDM*) Modellen unterschieden. Im PIM werden Struktur und Funktionalität des Systems formal beschrieben, zur Anwendung kommen Modellierungs- und Spezifikationssprachen wie UML oder OCL. Im PSM wird konkreten Komponentenmodellen wie z. B. EJB, CCM oder COM Rechnung getragen, während im EDM die systemspezifische Infrastruktur wie Webserver oder Komponenten-Container berücksichtigt werden. Wird ein PIM auf einem PSM eingesetzt, lassen sich automatisch Code-Gerüste erzeugen, die dann wiederum in einem EDM ausgeführt werden können.

### 2.3.4 Dynamische Architekturbeschreibungssprachen

Viele der anfänglich definierten Architekturbeschreibungssprachen waren lediglich in der Lage, *eine* Systemkonfiguration aus Komponenten und Verbindern zu beschreiben. Eine „dynamische ADL“ ist dagegen fähig, eine Softwarearchitektur beschreiben, welche sich dynamisch verändern kann. Daher muss eine dynamische ADL Veränderungen an Komponenten, Verbindern oder Konfigurationen ausdrücken können. Wie inzwischen von verschiedenen Autoren angemerkt wird, ist diese Sichtweise auf dynamische Architekturen insofern irreführend, als dass eine ADL mit Unterstützung für dynamische Verbinder und Rekonfiguration letztlich nur dynamische Aspekte eines Systems innerhalb einer statischen Architektur beschreibt – die Punkte im System, an denen Veränderungen auftreten können, sind letztendlich festgelegt (siehe z. B. [Szy02, S. 429] oder [OMT08, S. 902]).[17]

Taylor, Medvidović und Dashofy betrachten in [TMD10, Kapitel 14, S. 525–564] aufbauend auf dem Artikel [OGT+99] die Anpassung (*adaptation*) von Softwarearchitekturen aus einem deutlich abstrakteren Blickwinkel heraus. Unter diesem Blickwinkel soll eine gute Beschreibung einer Softwarearchitektur den Softwareentwicklern die Anpassung der Architektur an neue Gegebenheiten und Bedürfnisse erleichtern. Der Beitrag einer Architekturbeschreibungssprache beschränkt sich somit darauf, die anzupassenden Elemente (Komponenten, Verbinder und Konfigurationen) zu identifizieren, so dass Entwickler daran Änderungen vornehmen können. Der durch eine Archi-

[17] Die Eigenschaft einer *dynamischen Architektur* darf nicht mit der oftmals ebenfalls als *dynamisch* bezeichneten Beschreibung des *Verhaltens* von Elementen im Rahmen der Architektur verwechselt werden (wie z. B. den Diagrammarten zur Beschreibung dynamischen Verhaltens in der UML, also Zustands-, Aktivitäts- oder Interaktionsdiagrammen). Von einer dynamischen Architektur kann erst die Rede sein, wenn sich Elemente der Architekturbeschreibung (also Komponenten, Verbinder oder die Konfiguration) verändern. Die Grenze zwischen diesen beiden Begriffen verläuft allerdings – wie so oft – fließend.

tekturbeschreibungssprache unterstützte Architekturstil kann allerdings einen großen Einfluss auf die Leichtigkeit von Anpassungen haben.

In [BCK03, Abschnitt 5.3, S. 105ff.] erläutern Bass, Clements und Kazman im Überblick, mit welchen Maßnahmen (*Tactics*) eine Softwarearchitektur auf das Qualitätsmerkmal der Modifizierbarkeit (*Modifiability*) hin ausgerichtet werden kann. Unter „Modifizierbarkeit“ verstehen die Autoren einen sehr allgemeinen Begriff, der Änderungen an einem Softwaresystem sowohl während der Entwicklungsphasen als auch während der Einsatzphasen umfasst. So zählen Bass et al. Maßnahmen auf, welche die Kapselung, den technischen und fachlichen Zusammenhang sowie die generelle Verwendbarkeit von Modulen stärken, um die Auswirkungen von Änderungen auf einen kleinen Teil des Gesamtsystems zu begrenzen. Diese Maßnahmen können jedoch nicht von Nicht-Entwicklern wie z. B. Anwendern unkompliziert genutzt werden. Hierfür ist es erforderlich, den Kompositionszeitpunkt in die Laufzeit zu verlagern, also dynamische Änderbarkeit in der Softwarearchitektur *vorzusehen*. Um den Kompositionszeitpunkt zu verlagern, schlagen Bass et al. Techniken wie z. B. das Einführen einer Registry (siehe auch das gleichnamige Entwurfsmuster in Unterabschnitt 2.4.2), die Verwendung von Konfigurationsdateien oder das Ausnutzen dynamischer Bindung von Methoden mittels Polymorphie vor. Die Autoren erwähnen weiterhin, dass unabhängige Prozesse nur dann zur Laufzeit gekoppelt werden können, wenn sich die Prozesse an einheitlich spezifizierte Protokolle halten. Bass et al. gehen allerdings nicht weiter ins Detail, so dass offen bleibt, wie diese Maßnahmen konkret in einer Softwarearchitektur umgesetzt werden können.

Für Plugin-Systeme sind solche Techniken wie die von Bass et al. angedeuteten relevant, da die Softwarearchitektur eines zur Laufzeit durch Plugins erweiterbaren Systems auf dynamische Änderbarkeit ausgelegt sein muss. Direkte Änderungen an Komponenten oder Verbindern sind in einem Plugin-System dagegen nicht zu erwarten, da ein Plugin lediglich im gegebenen Kontext an vorgesehener Stelle Funktionalität ergänzt. In der Sichtweise von Architekturbeschreibungssprachen sind die zu erwartenden Änderungen somit auf die *Konfiguration* des Systems beschränkt. In einem Plugin-System ist insbesondere das Hinzufügen und Entfernen von Komponenten einzuplanen. Taylor, Medvidović und Dashofy ordnen Plugins als einen auf Schnittstellen fokussierten Architekturstil ein, der einen „Entwurf mit dem Ziel der Änderbarkeit“ (*design for change*) unterstützt [vgl. TMD10, S. 552f.]. Mit dem speziellen Ziel der Modellierung der Architektur eines Plugin-Systems ist die Betrachtung von dynamischen ADLs durchaus sinnvoll.

Beispielsweise nutzt Chatley in [Cha05] eine moderne Variante einer der ältesten dynamischen ADLs namens *Darwin* [siehe Dar, MDEK95] zur Modellierung der Softwarearchitektur des Plugin-Systems *MagicBeans* (siehe auch Unterabschnitt 3.3.4). Darwin erlaubt die Modellierung von Komponenten, deren ein- und ausgehenden Anschlüssen (*ports*) sowie von Bindungen (*bindings*) zwischen den Anschlüssen. Anders als das Konzept des Verbinders in anderen ADLs ist eine Bindung kein der Komponente gleichrangig gestelltes Modellierungselement. Darwin beherrscht die hierarchische Komposition von Komponenten sowie die Beschreibung verteilter Systeme. Der dynamische Anteil eines Darwin-Modells besteht darin, die Konfiguration einer Architektur auf Basis einer gegebenen Menge von Komponenten durch programmiersprachenähnliche Konstrukte zu erstellen und zu ändern. Es können Instanzen der Komponenten

erzeugt sowie Bindungen zwischen den Anschlüssen dieser Instanzen eingerichtet und aufgelöst werden. Die grafische Darstellung von Konfigurationen in Darwin ist einfach, aber intuitiv – Komponenten (Rechtecke) verfügen über Anschlüsse (schwarze und weiße Kreise auf dem Rand der Komponenten), jede Bindung wird durch eine Linie zwischen einem schwarzen und einem weißen Port visualisiert.

Die von Chatley genutzte Variante *Darwin*$^{i}$ [siehe Wae04] ersetzt die explizite Programmierung von Konfigurationen und Konfigurationsänderungen durch Deklarationen, welche die Kardinalität von Komponenteninstanzen sowie von Bindungen beschränken. Eine konkrete Softwarearchitektur ergibt sich somit implizit durch die deklarierten Beschränkungen. Die Ausdrucksmöglichkeiten von Darwin und Darwin$^{i}$ sind allerdings vorrangig auf strukturelle Eigenschaften einer Softwarearchitektur hin ausgelegt. Eine Spezifikation von Verhalten wird nicht thematisiert, ist aber im zur Definition der Darwin-Semantik verwendeten $\pi$-Kalkül teilweise möglich.

Unter den von Medvidović und Taylor in [TMD10, Kapitel 6, S. 185–247] vorgestellten ADLs älteren und jüngeren Entstehungsdatums bieten neben Darwin einige weitere Sprachen die Möglichkeit dynamischer Anpassungen einer Softwarearchitektur. Allerdings werden solche Änderungen immer in Form von Skripten oder über zur Programmierung bereitgestellte APIs bewirkt. Eine grafische Darstellung von Änderungen ist lediglich in der Form möglich, dass die aus den programmierten Änderungen resultierende Architektur in einem Diagramm visualisiert wird. Durch Werkzeugunterstützung kann auch der umgekehrte Weg eingeschlagen werden: Beispielsweise unterstützt das Werkzeug ArchStudio [Arc] die Anpassung von Architekturen an ein neues Modell, indem es die Unterschiede zum bisherigen Modell ermittelt und die notwendigen Operationen zur Anpassung ausführt. Da allen Ansätzen letztlich eine programmgesteuerte Vorgehensweise gemein ist, erfolgt konsequenterweise die Warnung, dass Änderungen an der Architektur eines laufenden Softwaresystems eine vorübergehende Stilllegung (*quiescence*) desselben erfordern [siehe TMD10, S. 549f.].

Bradbury et al. haben in [Bra04, BCDW04] einen Überblick über verschiedene Spezifikationstechniken für dynamische Softwarearchitekturen zusammengetragen und decken dabei auch einige ADLs ab. Eines der Klassifikationskriterien ist die Möglichkeit des Initiierens von Änderungen aus dem modellierten System heraus. Bradbury et al. attestieren fast allen 14 untersuchten Formalismen diese Fähigkeit, unter anderem auch Darwin.[18] Selbst-initiierte Konfigurationsänderungen werden in der Regel durch eine Komponente als Reaktion auf ein Ereignis hin ausgeführt und in dem jeweils zugrunde liegenden Formalismus (Graphersetzung, Prozessalgebra etc.) spezifiziert.

Die im Rahmen des ArchWare-Projekts entstandene $\pi$-ADL [MKB$^{+}$04, Oqu06, siehe auch Abschnitt 2.3.3] bietet ebenfalls Operationen zur Veränderung der Architektur in der Verhaltensbeschreibung von Komponenten. So ist es Komponenten möglich, Komponenten zu instantiieren, diese zu einem System zu komponieren und Kompositionen wieder aufzulösen.

[18] Allerdings stellen Bradbury et al. fest, dass gerade Darwin besonders wenige Operationen zum Ändern der Architektur anbietet, da zum Beispiel keine Änderungen an der Menge der Anschlüsse einer deklarierten Komponente möglich sind.

### 2.3.5 Architekturbeschreibungssprachen und Petrinetze

Viele Architekturbeschreibungssprachen haben ihren Schwerpunkt in der formalen Spezifikation und Analyse einer Architektur. Da die Semantik der Architektur-Elemente formal ausgedrückt werden können muss, greifen ADLs auf verschiedene Formalismen zurück. Medvidović und Taylor nennen in [MT00] Statecharts nach Harel, PO-Sets (partiell geordnete Ereignismengen), CSP (kommunizierende sequentielle Prozesse nach Hoare), sowie modellbasierte, algebraische und axiomatische Formalismen. Keiner dieser Formalismen eignet sich alleine zur vollständigen Beschreibung von Struktur *und* Verhalten der Architektur, daher werden sie stets nur zur Unterstützung einzelner Aufgabenbereiche der ADL herangezogen.

Einige ADLs greifen zur semantischen Spezifikation des Verhaltens von Komponenten, Schnittstellen und/oder Verbindern auf Petrinetze zurück. Deutlich seltener sind ADLs, welche Petrinetze zur Beschreibung der Architekturkonfiguration einsetzen.

**Petrinetze zur Verhaltensspezifikation** Bemerkenswert ist, dass keine der von Medvidović und Taylor in [MT00] untersuchten oder in [TMD10, Kapitel 6] vorgestellten Architekturbeschreibungssprachen Petrinetze zur Spezifikation von Verhalten nutzt. Unter den von Bradbury et al. in [Bra04, BCDW04] untersuchten formalen Spezifikationstechniken für dynamische Softwarearchitekturen finden sich zwar Ansätze auf der Basis von Graphersetzungssystemen und Prozessalgebra, aber keine auf der Basis von Petrinetzen. Dennoch gab es schon früh Formalismen zur Beschreibung von Softwarearchitekturen, welche zur formalen Spezifikation und Verifikation von Eigenschaften des beschriebenen Systems ergänzend zur strukturellen Beschreibung der Architektur eine Verhaltensspezifikation von Komponenten, Schnittstellen oder Verbindern anhand von Petrinetzen vorsehen.

Ermel, Bardohl und Padberg stellen in [EBP01] eine allgemeine, auf Graphersetzungssystemen basierende Technik zur formalen Integration von struktureller Spezifikation und Verhaltensspezifikation vor. Das den Ansatz illustrierende Beispiel kombiniert eine strukturelle Beschreibung in Form einer Architekturbeschreibung mit einer Verhaltensspezifikation in Form eines Petrinetzes. Die Technik ist generisch verwendbar, so dass beliebige ADLs mit beliebigen Formalismen zur Verhaltensspezifikation kombiniert werden können, indem jeweils eine Abbildung auf den zugrundeliegenden Graphersetzungsformalismus definiert wird.

Die Verwendung von Petrinetzen zur Verhaltensspezifikation im Rahmen einer Architekturbeschreibung findet sich besonders oft im Kontext der Entwicklung von Web-Service-basierten Anwendungen. Der Einsatz in diesem Gebiet ist naheliegend, da formal verifizierbare Spezifikationen für Interaktionen zwischen Dienstanbietern und -nutzern benötigt werden.

Ling, Poernomo und Schmidt ergänzen beispielsweise in [LPS03] die ursprünglich auf Statecharts zur Verhaltensbeschreibung zurückgreifende RADL („Reliable ADL“, [RPS02]) um v.d. Aalsts Workflow-Netze [Aal97]. Die Netze werden zur Spezifikation von Interaktionsprotokollen an den benötigten und angebotenen Schnittstellen einer Komponente eingesetzt. Um eine Unterscheidung von Black-Box- und White-Box-Sicht zu ermöglichen, setzen Ling et al. die von v.d. Aalst in [Aal00] definierte Vererbung (*inheritance*) von Verhalten in Workflow-Netzen ein. Der öffentlich sichtbare

Workflow der Komponentenschnittstellen muss verhaltensähnlich zum internen Workflow der Komponente, eingeschränkt auf die Schnittstellen-Transitionen, sein. Auf der partiellen Ordnung der Verhaltensähnlichkeit aufbauend definieren sie Kompatibilität und Ersetzbarkeit (*substitutability*) von Komponenten.

Im Vergleich zu anderen ADLs, die Schnittstellen von Komponenten nur durch Namen bzw. Signaturen beschreiben und Verhaltensmodellierung nur für die gesamte Komponente durchführen, liegt der Vorteil der RADL darin, dass die Interoperabilität einzelner Schnittstellen formal verifiziert und ggf. ein geeigneter Adapter automatisch konstruiert werden kann. Mit Petrinetzen wurde im Vergleich zu Statecharts eine formal klarere Verhaltensbeschreibungssprache gewählt. Allerdings muss in der RADL mit mehreren Formalismen umgegangen werden, um das Gesamtsystem zu beschreiben, da die Struktur der Architektur nicht mit Workflow-Netzen oder Statecharts beschreibbar ist. Die in der RADL mögliche hierarchische Dekomposition eines Systems in Komponenten ist nicht dynamisch veränderbar, sondern muss vom Architekten vorgegeben werden.

V. d. Aalst et al. stellen in [ABH+07a, ABH+07b] ein Architektur-Rahmenwerk vor, welches an den Spezifikationen der *Service Component Architecture* (SCA, siehe Unterabschnitt 2.2.3) ausgerichtet ist. Es handelt sich somit um ein Rahmenwerk, welches auf die Spezifikation und Implementierung dienstorientierter Architekturen spezialisiert ist. In diesem Rahmenwerk können unter anderen Alternativen auch Workflow-Netze zur Implementierung von Prozessen eingesetzt werden. Aber auch hier wird der Formalismus zur Beschreibung von Prozessen mit anderen Formalismen zur Beschreibung von Daten und Operationen kombiniert.

**Ausschließliche Beschreibung mit Petrinetzen** Zur grafischen Darstellung von Konfigurationen nutzt kaum eine ADL Petrinetze. Medvidović und Taylor argumentieren in [MT97, Abschnitt 3.1] gegen die Betrachtung von Petrinetzen als ADL. Allerdings liegt der Kritik eine sehr primitive Verwendung von Petrinetzen zugrunde, in der Stellen als zustandbehaftete und Transitionen als Operationen durchführende Komponenten betrachtet werden. Petrinetz-Kanten sollen dabei das Konzept der Verbinder abbilden. Unter dieser Sichtweise, welche den dynamischen Aspekt von Petrinetzen (Markierungen und Schaltregel) ignoriert, ergibt sich notwendigerweise die Kritik, dass Petrinetze zu restriktiv in den zulässigen Verbindungsmöglichkeiten zwischen Stellen und Transitionen seien, um als ADL gelten zu können.

Yu und Cai definieren eine auf objektorientierten Petrinetzen basierende ADL namens „OPNADL“ [YC04], „$\pi$-net-ADL“ [YCWH05] bzw. „ADLMAS“ [YC06]. Zugrunde liegt ein aufwändiger objektorientierter Petrinetzformalismus, der spezielle Ein- und Ausgangsstellen und -kanten für Objektnetze sowie eine globale Nachrichtenweiterleitungsstelle (*intelligent linking place*) definiert. Komponenten werden als Objektnetze und Verbinder als Nachrichtenweiterleitungsstelle modelliert.

Die beiden neueren Varianten der ADL modellieren Komponenten und Verbinder als Agenten, denen ein BDI-Modell zugrunde liegt. Ein Verbinder-Agent besteht aus einer Weiterleitungsstelle sowie einer Wissensbasis. Ein Komponenten-Agent besteht aus einer Ein- und einer Ausgangsstelle, sowie mehreren Transitionen und Stellen, die seine Wissensbasis, Ziele und Pläne manipulieren und speichern.

Durch das Markenspiel der Netze lässt sich das dynamische Verhalten der Komponenten und Verbinder im System betrachten. In der $\pi$-net-ADL-Variante wird zusätzlich das $\pi$-Kalkül zur Modellierung des Hinzufügens und Entfernens von Agenten herangezogen. Es wird jedoch nicht klar, ob das Hinzufügen und Entfernen tatsächlich die Systemstruktur dynamisch verändert oder nur die in den Wissensbasen gespeicherten Beziehungen zwischen bestehenden Komponenten und Verbindern manipuliert. Da die Mengen der Stellen, Transitionen und Kanten endlich definiert sind und es keine Instanzen von Netzen gibt, sind dynamischen Veränderungen des Systems auf jeden Fall enge Grenzen gesetzt. Zudem scheint der Formalismus nur einen Verbinder im System zu unterstützen, da es nur eine Nachrichtenweiterleitungsstelle gibt. Im Vergleich zu anderen Techniken im Bereich der agentenorientierten Softwareentwicklung (siehe Abschnitt 2.5) ist anzumerken, dass die Komponenten-Agenten nicht adaptiv sind, da ihre zur Verfügung stehenden Verhaltensweisen fest in ihren Transitionen kodiert sind.

Einige Autoren verzichten im Kontext von WebServices gänzlich auf eine spezielle Sprache zur strukturellen Beschreibung der Architektur und definieren eine rein petrinetzbasierte Technik zur Verhaltensbeschreibung. An die Stelle der strukturspezifizierenden ADL treten in einem solchen Ansatz die verschiedenen unter der Sammelbezeichnung der WebService-Standards bekannten XML-Sprachen zur Definition von Diensten, Prozessen und Orchestrierung. Die gefärbten Petrinetze dienen in diesen Ansätzen der Modellierung des Verhaltens des Gesamt-Systems, wobei das Netzmodell die durch WebService-Spezifikationssprachen beschriebene Systemstruktur nachbildet.

Beispielsweise setzen die WS-Netze von Zhang et al. [ZCCK04] auf früheren Techniken zur strukturellen Analyse und Entwurf auf, welche gefärbte Petrinetze verwenden. Jede Komponente, jede Schnittstelle und jeder Verbinder des Systems wird als gefärbtes Netz repräsentiert. Die Komposition komplexer Komponenten ist mit Hilfe der im Formalismus der hierarchischen gefärbten Petrinetze enthaltenen Technik der Substitutionstransition möglich. Die Netze kommunizieren über Fusionsstellen, also über asynchrone Nachrichtenkommunikation.

Fu, Dong und He stellen in [FDH06] eine Technik namens SO-SAM (*Service-oriented Software Architecture Model*) vor, welche auf der Architekturbeschreibungssprache SAM [HD02] aufbaut. Der Ansatz kombiniert einen Petrinetzformalismus zur Verhaltensspezifikation mit einer Temporallogik zur Spezifikation der gewünschten Eigenschaften von Komponenten und Verbindern. Fu, Dong und He verwenden Prädikat/Transitionsnetze (PrTN) und Lineare Temporallogik (LTL). In SO-SAM wird zu jeder Komponente ein Dienstnetz (*service net*) angegeben, welches das Verhalten der Komponente beschreibt. Die Anschlüsse (*ports*) der Komponenten werden als Stellen modelliert, so dass Nachrichten als Marken in den Stellen repräsentiert werden können. Ein Verbinder wird durch einfache Petrinetzelemente modelliert, welche Anschlussstellen verbinden. Komponenten können im Gegensatz zu Verbindern hierarchisch komponiert werden. Die oberste Ebene des Modells stellt Verbinder gemeinsam mit vergröberten Komponenten dar. Die temporallogisch spezifizierten Eigenschaften können überprüft werden, indem das gesamte Modell ins Werkzeug Maude übertragen und dort analysiert wird.

Beide Ansätze, WS-Netze und SO-SAM, nutzen einen hierarchischen Petrinetzformalismus, in dem Elemente wie z. B. Komponenten auf der nächsthöheren Abstraktionsebene vergröbert dargestellt werden. Die Architektur des modellierten Systems

ist somit statisch im Petrinetzmodell kodiert und nicht veränderlich. Die Kommunikation über Fusionsstellen schließt die Modellierung von Verbindern mit synchronem Verhalten aus.

In dieser Arbeit wird anstelle der bisher vorgestellten ADLs eine eigene, petrinetzbasierte Modellierungstechnik verwendet werden, die in Abschnitt 4.2 eingeführt wird. Eine Einordnung dieser Technik in den Kontext der Architekturbeschreibungssprachen findet in Unterabschnitt 4.5.4 statt.

## 2.4 Entwurfsmuster

In der Objektorientierten Softwareentwicklung lassen sich oft wiederkehrende Muster von Beziehungen zwischen kleineren Gruppen von Objekten und ihren Schnittstellen beobachten. Da diese Muster auf regelmäßig wiederkehrende Probleme im objektorientierten Entwurf angewendet werden, spricht man von Entwurfsmustern (*Design Pattern*). Dieser Begriff ist wesentlich durch das 1995 erschienene Buch [GHJV98] von Gamma et al. geprägt worden, in dem die Autoren eine Sammlung grundlegender Entwurfsmuster veröffentlicht haben, die inzwischen zum Grundkanon der objektorientierten Softwareentwicklung gehört. Seitdem sind von etlichen Autoren weitere Muster identifiziert und in diversen Sammlungen veröffentlicht worden. Die Entdeckung, Katalogisierung und Einordnung von Entwurfsmustern ist eine noch längst nicht abgeschlossene Aufgabe, der sich z. B. der im Mai 2008 aufgelöste Arbeitskreis Architektur- und Entwurfsmuster der GI widmete [AKA08].

Das Konzept des Entwurfsmusters ist der (klassischen) Architekturtheorie entlehnt. Alexander et al. haben in [AISJ77] im Kontext des Entwurfs von Gebäuden bzw. der Stadtplanung einen Bedarf an Mustern zur Lösung von Problemen erkannt, die immer wieder auftreten, wobei der Kern der Lösung mehrfach wiederverwendbar ist, während die konkrete Lösung bei jeder Anwendung des Musters variiert. Auf die objektorientierte Softwareentwicklung übertragen beschreibt ein Entwurfsmuster ein *Problem* inklusive dessen Kontext, eine allgemeine Vorlage zur Anordnung zentraler Objekte, Klassen und Schnittstellen, welche eine prinzipielle *Lösung* des Problems ausmachen, sowie die *Konsequenzen* des Einsatzes des Musters [vgl. GHJV98, S. 3]. Das für die resultierende Softwarearchitektur entscheidende Merkmal eines Entwurfsmusters ist also das Arrangement der Objekte und Schnittstellen inklusive deren Verantwortlichkeiten und Beziehungen untereinander. Damit lassen sich Entwurfsmuster als Mikroarchitekturen auffassen [vgl. Szy02, S. 157].

Eng verwandt mit dem Begriff des Entwurfsmusters ist der in Unterabschnitt 2.3.1 vorgestellte Begriff des Architekturmusters. Wie bereits erläutert, unterscheiden sich die Begriffe nur in der Abstraktionsebene bzw. dem Umfang ihres Anwendungsgebiets. Da die Grenzen zwischen den Begriffen fließend sind, soll hier nicht auf einer strengen Unterscheidung bestanden werden.

Auch bei Plugin-Systemen lassen sich wiederkehrende Entwurfsmuster identifizieren. Zum einen erfordert die Realisierung flexibel erweiterbarer Software einen häufigen Einsatz bestimmter Entwurfsmuster zur statischen Entkopplung von Klassen sowie zum Herstellen dynamischer Verbindungen zwischen den entkoppelten Objekten. Zum anderen kann die Idee der Modularisierung von Software durch Plugins selber als

Entwurfsmuster verstanden werden. Zuletzt lassen sich auch einige Muster auf der nächsten Abstraktionsebene identifizieren, welche Richtlinien zur (De-)Komposition von Software aus bzw. in Plugins beschreiben.

Die Listen der im Folgenden vorgestellten Muster sind keineswegs vollständig, sie beschreiben nur die prominentesten Beispiele und deren Vorkommen in Plugin-Systemen.

### 2.4.1 Muster zur Entkopplung von Objekten

Für die Gestaltung lose gekoppelter Dienst- oder Erweiterungsbeziehungen können zunächst einmal solche Entwurfsmuster herangezogen werden, die der Trennung von Schnittstelle und Implementierung dienen. So können statische Abhängigkeiten im Code durch dynamische Abhängigkeiten zur Laufzeit ersetzt werden.

**Factory Method und Abstract Factory** [GHJV98, S. 107 u. S. 87] sind zwei Erzeugungsmuster, die häufig unter dem schlichten Namen *Factory* referenziert werden. Beide Muster erlauben die Spezialisierung von Objektinstantiierungen durch Subklassen, ohne dass die Spezialisierung dem Nutzer der *Factory* bekannt sein muss. Während die *Factory Method* lediglich in der Subtypenhierarchie einer Klasse wirkt, dient die *Abstract Factory* dem Erzeugen von Instanzen aus mehreren, thematisch zusammenhängenden Klassenhierarchien. Gamma et al. betonen die Eignung der Muster für Rahmenwerke, da ein Rahmenwerk oftmals nur mit generischen Typen arbeitet, deren konkrete Implementierungen dann durch *Factories* erzeugt werden können.

Die für Plugin-Systeme typische Erweiterungsbeziehung kann oftmals mithilfe von *Factories* realisiert werden. Eine von der erweiterbaren Komponente spezifizierte Fabrikmethode kann von Erweiterungen konkretisiert werden, um spezialisierte Objekte mit zusätzlicher Funktionalität zu erzeugen. Ein Beispiel ist die Prozedur `NPP_New` der NPAPI (siehe Unterabschnitt 3.1.1.4), welche Plugin-Instanzen erzeugt. Um der erweiterbaren Komponente Referenzen auf konkrete Implementierungen der Fabrik zugänglich zu machen, können die in Unterabschnitt 2.4.2 beschriebenen Muster herangezogen werden.

**Bridge und Strategy** [GHJV98, S. 151 u. S. 315] sind zwei Entwurfsmuster, welche Schnittstelle (Abstraktion) und Implementierung voneinander lösen. Während bei *Strategy* alternative Algorithmen hinter einer einheitlichen Schnittstelle angeboten werden, bietet die *Bridge* allgemein die Möglichkeit, getrennte Spezialisierungshierarchien für Abstraktionen und Implementierungen zu pflegen. Um zur Laufzeit die Abstraktion zu konkretisieren, kapselt ein Kontext-Objekt die Referenz zur gewählten Implementierung.

Plugin-Systeme können kaum ohne diese Muster realisiert werden, sie sind sowohl für Erweiterungs- als auch für Dienstbeziehungen unentbehrlich. Eine erweiterbare Komponente spezifiziert die abstrakte Schnittstelle, während Erweiterungen beliebige Implementierungen bereitstellen. Das Kontext-Objekt wird meist von der erweiterbaren Komponente in Folge eines Konfigurationsprozesses zusammengestellt, der mit Hilfe der Muster aus Unterabschnitt 2.4.2 realisiert werden

kann. Bei Dienstbeziehungen stellt der Dienstanbieter die Implementierung bereit, während die abstrakte Schnittstelle häufig separat (siehe Muster *Separated Interface*) definiert wird. Meist konfiguriert der Dienstnutzer sein Kontext-Objekt selber, wobei er dieselben Entwurfsmuster wie zur Konfiguration der Erweiterungsbeziehung herangezogen werden können.

**Observer** [GHJV98, S. 293] wird auch *Publish and Subscribe* oder *Listener* genannt und realisiert automatische Benachrichtigungen, ohne dass eine statische Abhängigkeit von der Nachrichtenquelle zum Empfänger der Benachrichtigung benötigt wird. Die Quelle (*Subject, Publisher*) verwaltet eine Menge von Empfängern (*Observer, Subscriber, Listener*), welche ihm nur als Implementierungen einer einheitlichen Empfangsschnittstelle bekannt sind. Somit können beliebige Objekte diese Schnittstelle implementieren und sich bei der Quelle anmelden. Sie werden dann automatisch über Veränderungen am Zustand der Quelle informiert.

Die Eignung dieses Musters für Rahmenwerke, die zentrale Ereignisse an beliebige Interessenten ausliefern können sollen, ist offensichtlich. Gleiches gilt natürlich für Plugin-Systeme: Sowohl Erweiterungsbeziehungen als auch Dienstbeziehungen haben mit diesem Muster die Möglichkeit, Informationen vom erweiterbaren System an alle Erweiterungen bzw. vom Dienstanbieter zum Dienstnutzer zu senden, ohne dass statische Abhängigkeiten entstehen. Das Muster stellt auch eine wesentliche Grundlage für Callback-Mechanismen dar, da es eine Anwendung des *Inversion of Control*-Prinzips ist (siehe Diskussion zu *Dependency Injection* in Unterabschnitt 2.4.2).

**Separated Interface** [Fow03, S. 476ff.] beschreibt die Trennung von Interface und Implementierung. Fowler empfiehlt den Einsatz, um beidseitige Abhängigkeiten zu unterbrechen oder um alternative Implementierungen zu erlauben. Zur Compile- oder Laufzeit muss die Verbindung zwischen Interface und Implementierung hergestellt werden, hierzu empfiehlt Fowler den Einsatz von der Muster *Factory* und/oder *Plugin*.

Das Muster *Separated Interface* ist offensichtlich in allen Erweiterungs- und Dienstbeziehungen notwendig, da nur so eine erweiterbare Systemkomponente oder ein Dienstnutzer frei von Abhängigkeiten zu verfügbaren Implementierungen werden kann. Es ist gut mit vielen Entwurfsmustern zur losen Kopplung von Objekten kombinierbar, so dass es selten alleine auftritt.

### 2.4.2 Muster zum Herstellen dynamischer Beziehungen

Die im vorigen Unterabschnitt vorgestellten Entwurfsmuster sorgen zwar für weniger Abhängigkeiten in der Klassenhierarchie der gesamten Anwendung, aber sie erzeugen das Problem, zur Laufzeit dynamische Verbindungen zwischen den entkoppelten Objekten herstellen zu müssen. Dieses Problem wird durch die folgenden Entwurfsmuster behandelt.

**Registry** [Fow03, S. 480ff.] dient dem Auffinden von Objektreferenzen anhand von eindeutigen Schlüsseln. Als Schlüssel können textuelle Bezeichner oder Klassen- bzw. Schnittstellentypen dienen, mitunter wird statt eines Schlüssels auch

ein komplexer Filter zugelassen. Eine *Registry* ist typischerweise global ansprechbar. Nichtsdestotrotz kann innerhalb der Registry eine Unterteilung der Anfragen nach ihrem Ursprung vorgenommen werden. Fowler empfiehlt, das *Registry*-Muster möglichst sparsam einzusetzen, da globale, veränderliche Daten prinzipiell eine Fehlerquelle darstellen (insbesondere in nebenläufigen Systemen). Ein sehr ähnliches Entwurfsmuster ist der *Service Locator* [ACM03, S. 315ff.], der zusätzlich die Komplexität des Findens und Aufrufens von Fabriken für Dienstimplementierungen in verteilten Systemen kapselt.

In Plugin-Systemen wird üblicherweise mindestens ein Vorkommen des *Registry*-Musters vorgesehen, um navigierbare Verbindungen zwischen Objekten verschiedener Plugins aufzubauen. Die verwendeten Schlüssel können z. B. Bezeichner für Plugins sowie Dienst- oder Erweiterungsspezifikationen sein, Beispiele dafür lassen sich in fast allen in Kapitel 3 untersuchten Plugin-Architekturen finden (siehe Tabelle 2.2). Bis auf den OSGi-Bundle-Context, der jedem Bundle individuell zur Verfügung gestellt wird, sind alle diese Registries global. Die Lokalität des Bundle-Context wird allerdings dadurch relativiert, dass er auf Anfrage Referenzen auf sämtliche Bundle-Repräsentanten bereitstellt.

Das *Registry*-Muster bündelt auf einfache Weise alle Informationen, die zur dynamischen Rekonfiguration eines Systems notwendig sind. Im Rahmen der konzeptionellen Modellierung in Abschnitt 4.3 wird deutlich werden, dass Plugins, welche ausschließlich als Erweiterung über eine spezielle Schnittstelle ins System integriert werden, auch ohne eine zentrale *Registry* auskommen können. Unentbehrlich ist das Muster dagegen für die Umsetzung dienstorientierter Architekturen, da diese auf eine zentrale, während der Laufzeit des Systems permanent aktuell gehaltene Auskunftei für verfügbare Dienstanbieter angewiesen sind. Das einzige der in dieser Arbeit untersuchten existierenden Plugin-Systeme, welches gänzlich auf eine Registry verzichtet, ist das MagicBeans-System (siehe Unterabschnitt 3.3.4).

**Dependency Injection** [Fow04], auch *Third-Party-Binding* [Esk99] genannt, ist ein Muster, welches das Prinzip *Inversion of Control* anwendet, um dynamische Verbindungen zwischen Objekten aufzubauen. Lose gekoppelte Klassen, deren Objekte Referenzen auf konkrete Implementierungen einer Schnittstelle benötigen, nehmen diese entweder im Konstruktor oder über eine `Set`-Methode entgegen. Die Kontrolle über das Herstellen der Verbindung liegt statt wie beim *Registry*-Muster nicht beim Empfänger der Referenz, sondern bei einem Dritten (*third party*), also einer separaten, für den Zusammenbau der Objekte verantwortlichen Instanz.

Obwohl dieses Entwurfsmuster sich gut eignet, um die deklarative Konfiguration von Plugin-Systemen zu unterstützen, wird es seltener als das *Registry*-Muster eingesetzt. Ein Beispiel für die Anwendung des Musters ist die Übergabe des `BundleContext` an ein Bundle während seiner Aktivierung im OSGi-Rahmenwerk (siehe Unterabschnitt 3.2.1.6). Das MagicBeans-Plugin-System (siehe Unterabschnitt 3.3.4) setzt zur Kopplung der Plugins ausschließlich auf dieses Entwurfsmuster.

| System | Registry | Anfrageschlüssel | Rückgabewert | Z | A | siehe |
|---|---|---|---|---|---|---|
| NPAPI | *(intern)* | MIME-Typ | Plugin-Instanz | X | | 3.1.1.3 |
| Mozilla-Extensions | Component Registry | Vertrags- oder Klassenname | Komponentenfabrik ($\hat{=}$ Klasse) | | X | 3.1.2.1 |
| | Type Registry | Interface-Name | instantiierte Klasse | | X | 3.1.2.1 |
| | Chrome Registry | Chrome-URI | Datei-URI | | X | 3.1.2.2 |
| Eclipse (klassisch) | Plugin Registry | Name eines Erweiterungspunkts | Erweiterungspunkt-Objekt (Container für Erweiterungsobjekte) | X | (a) | 3.2.1.5 |
| OSGi | Bundle Context (b) | Dienstname | Menge von Dienstreferenzen | (c) | X | 3.2.1.6 |
| | | Dienstreferenz | implementierendes Objekt | (c) | X | 3.2.1.6 |
| | | – | Menge aller Bundles | | X | 3.2.1.6 |
| NetBeans | Default Lookup | Dienstname | Menge von implementierenden Klassen | X | X | 3.3.2 |
| | virtuelles Dateisystem | Dateipfad | Objekt | X | X | 3.3.2 |

*Erläuterung:*
Z Registry wird durch zentrale Verwaltung gefüllt (z.B. aus Metadaten).
A Registry wird aktiv durch Komponenten gefüllt (programmatisch).
(a) Erst ab Eclipse-Version 3.0.
(b) Der BundleContext ist lokal für jedes Bundle.
(c) Erst seit Einführung der „Declarative Services“ in OSGi.

Tabelle 2.2: Beispiele für das Registry-Muster in den untersuchten Plugin-Systemen

**Pluggable Factories** gibt es in mehreren Varianten [Vli98, Vli99, Cul99, Lau99]. Sie reduzieren einerseits den Aufwand, der beim Muster *Abstract Factory* entsteht, wenn weitere Implementierungen oder Klassenhierarchien zu einer bereits bestehenden Hierarchie hinzugefügt werden sollen. Statt explizit zu kodierende Subklassen der *Abstract Factory* zu verwenden, werden in der *Factory* Prototypen der zu erzeugenden Objekte verwaltet, um so die doppelte Klassenhierarchie zu vermeiden. Gleichzeitig erhöht der Einsatz von Prototypen die Flexibilität der *Factory*, da anstelle eines statischen Satzes an Fabrikmethoden eine dynamisch veränderliche Menge von Prototypen das Ergebnis der Fabrikation bestimmt. Die Konfiguration der Prototypen kann z. B. per *Dependency Injection* geschehen.

Die Varianten *Named Product Pluggable Factory* oder *Dynamic Pluggable Factory* referenzieren die zu erzeugenden Produkte nur noch anhand von Namen bzw. integrieren zusätzlich das Muster *Separated Interface*, so das der Code zum Anfordern der konkreten Produkte komplett unabhängig vom Code der bereitgestellten Produkte, also ohne Kenntnis von und Abhängigkeit zu verfügbaren konkreten Produktklassen kompiliert werden kann.

Der alternativ in [KQ05] präsentierte Ansatz nutzt die statische Initialisierung von Subklassen aus, um ohne externe Hilfe alle geladenen Implementierungen der *Factory* bekannt zu machen. Dieser Ansatz eignet sich allerdings nicht für Plugin-Systeme, die verzögertes Laden (siehe Unterabschnitt 3.2.1.4) von spezialisierten Klassen anstreben.

**Blackboard** oder *Repository* [Sha96, S. 263] ist ein Entwurfsmuster, welches Kommunikation zwischen sich gegenseitig nicht kennenden Objekten ermöglicht. Eine zentrale Datenhalde, *Blackboard* oder *Repository* genannt, kann jederzeit Informationen oder Nachrichten entgegennehmen, bewahrt diese auf und stellt sie beliebigen Interessenten auf Nachfrage zur Verfügung.

Dieses Muster erlaubt anonyme Kommunikation zwischen beliebigen Objekten und gibt keine Struktur für die Beziehung zwischen diesen Objekten vor. Daher ist es für Plugin-Systeme nicht prägend. Trotzdem wird es gelegentlich als zusätzlicher Kommunikationsweg angeboten, so z. B. durch das zentrale virtuelle NetBeans-Dateisystem (siehe Unterabschnitt 3.3.2). Die folgenden zwei Muster *Whiteboard* und *Implicit Invocation* bauen auf einem ähnlichen Prinzip auf, verzichten jedoch auf das Persistieren der Informationen und beschränken sich auf die unmittelbare Zustellung von Nachrichten.

**Whiteboard** [OSG04] vermeidet die explizite Verwaltung der Menge von Objektreferenzen auf Benachrichtigungsinteressenten, die im *Observer*-Muster durch jede Quelle bewerkstelligt werden muss. Stattdessen wird eine Dienstbeziehung mit einem Nutzer und vielen Anbietern modelliert: Interessierte Nachrichtenempfänger melden ihre Empfangsbereitschaft als Dienst in einer zentralen *Registry* an. Eine Nachrichtenquelle agiert als Dienstnutzer und erfragt alle Anbieter, um Ihnen die Nachricht zu senden.

Dieses Entwurfsmuster wird im OSGi-Rahmenwerk (siehe Unterabschnitt 3.2.1.6 zur Verwendung anstelle des *Observer*-Musters empfohlen. Es stellt eine semi-

zentralisierte Variante des folgenden Musters *Implicit Invocation* dar, da nur das Dienstverzeichnis zentral verwaltet wird, nicht jedoch die Nachrichtenzustellung selber.

**Implicit Invocation** [Sha96, S. 262] lockert wie das *Whiteboard*-Muster die Beziehung zwischen Nachrichtenquellen und Nachrichtenempfängern. Direkte dynamische Referenzen zwischen Quellen und Empfängern werden vermieden, indem eine zentrale Klasse (üblicherweise ein *Singleton*) sämtliche Benachrichtigungen entgegennimmt und an alle interessierten Empfänger weiterleitet. Die Zuordnung von Nachrichten an ihre Empfänger geschieht anhand von Typen oder Bezeichnern, mit denen die Nachrichten versehen werden.

In Plugin-Systemen kann dieses Entwurfsmuster nützlich sein, um eine einheitliche Kommunikationsschnittstelle zwischen beliebigen, noch unbekannten Plugins zu ermöglichen. Diese Schnittstelle wird aufgrund ihres globalen Charakters aber nicht zur Implementierung von Erweiterungs- oder Dienstbeziehungen eingesetzt, sondern bietet einen zusätzlichen Kommunikationsweg. Ein Beispiel für die Verwendung dieses Musters ist der zentrale XPCOM-Nachrichtendienst `nsIObserverService`, der Ereignisse nach Themen gegliedert zustellt (siehe Unterabschnitt 3.1.2.1). Das virtuelle Dateisystem von NetBeans (siehe Unterabschnitt 3.3.2) kann ebenfalls im Sinne des *Implicit Invocation*-Musters verwendet werden.

### 2.4.3 Das Plugin-Konzept als Muster

Die Idee, ein Softwaresystem durch Plugins flexibel erweiterbar zu gestalten, kann selber als Entwurfsmuster[19] verstanden werden. Dieser Sichtweise folgen mehrere Autoren, deren Definitionen des *Plugin*-Musters im Folgenden vorgestellt werden.

**Rice und Foemmel** beschreiben in Fowlers Patternsammlung [Fow03, S. 499ff.] die Aufgabe des *Plugin*-Musters als Verlagerung des Zeitpunktes, zu dem Verbindungen zwischen Objekten hergestellt werden, von der Kompilation zur Laufzeitkonfiguration. Der Vorteil des Musters ist im Wesentlichen die Zentralisierung der Konfiguration variabler Teile des Systems. Es wird als Kombination mehrerer bekannter Muster umgesetzt. Sämtliche erweiterbaren Teile des Softwaresystems werden mittels des Musters *Separated Interface* entkoppelt, so dass Implementierungen austauschbar werden. Die Konfiguration der dynamischen Verbindungen zur Laufzeit wird durch eine Variante des *Factory*-Musters[20] bewerkstelligt, die

[19] Man kann hier statt von „Entwurfsmuster“ auch von „Architekturmuster“ sprechen. Die Unterscheidung von Entwurfs- und Architekturmustern ist bereits in Unterabschnitt 2.3.1 angesprochen worden. Die folgenden Muster illustrieren den fließenden Übergang von Entwurfs- zu Architekturmustern, da sie wahlweise als technischer Lösungsvorschlag oder als weitreichende Architekturentscheidung verstanden werden können.

[20] Rice und Foemmel betonen, dass die *Factory* erst zur Laufzeit die Referenzen zu Implementierungen aufbauen darf. Das widerspricht der strengen Auslegung der Muster *Factory Method* und *Abstract Factory*, welche beide die Wahl der Implementierung über Subklassen, d.h. bereits beim Kompilieren, festlegen. Hier eignen sich also eher einige Varianten des *Pluggable Factory*-Entwurfsmusters.

sämtliche Konfigurationsparameter aus einer einheitlichen, zentralen und externen Quelle bezieht.

Rice und Foemmel empfehlen den Einsatz des *Plugin*-Musters in Situationen, die abhängig von der Laufzeitumgebung alternative Implementierungen benötigen und nennen als Beispiel die Unterscheidung zwischen Test- und Produktivumgebung. Damit beschreiben sie einen deutlich weniger weit gefassten Einsatzzweck als ihn viele existierende Plugin-Systeme praktizieren. Insbesondere betrachten sie nicht das Einbinden von Implementierungen, die zum Entwurfszeitpunkt noch gar nicht bekannt waren. Zwar ist ihr Muster auch auf solche Situationen anwendbar, jedoch deckt das zugrunde liegende *Factory*-Muster lediglich die Erzeugung parametrisierter, alternativer Implementierungen ab – weiterer Kommunikationsbedarf zwischen den sich gegenseitig zunächst unbekannten Objekten wird außer Acht gelassen.

**Mayer, Melzer und Schweiggert** [MMS03] richten ihre Definition des Entwurfsmusters „Plugin“ am ursprünglichen Plugin-Begriff aus, der die Auslagerung spezieller Fähigkeiten aus einer Hauptanwendung beschreibt. Sie betonen, dass die Anwendung zum Zeitpunkt des Kompilierens keine Kenntnis von konkreten Plugins hat und dass daher auch das dynamische Laden von Plugin-Code zur Laufzeit nicht aus fest verdrahteten Bibliotheken erfolgen kann. Das Muster umfasst einen Plugin-Lader, über den eine Menge verfügbarer konkreter Plugins erfragt werden kann, welche alle dasselbe Plugin-Interface implementieren. Die Plugin-Schnittstelle kann Methoden vorsehen, über welche Plugins ihre Zuständigkeit für eine bestimmte Anfrage bekannt geben können. Die Bereitstellungsmethode des Plugin-Laders wird von den Autoren mit dem *Factory*-Entwurfsmuster verglichen, wobei diese Methode aber nicht nur eins, sondern mehrere erzeugte Objekte liefert, welche wiederum auf dynamisch geladenen Klassen basieren.

Beispiele für Anwendungen dieses Entwurfsmusters sind Anwendungen, die spezialisierte Funktionalität über eine oder mehrere öffentliche Plugin-Schnittstellen aus der Hauptanwendung auskoppeln wie z. B. die NPAPI (siehe Unterabschnitt 3.1.1). Mayer et al. erwähnen zwar auch die Eclipse-Plattform (siehe Abschnitt 3.2), gehen aber nicht auf deren Erhebung des Plugin-Konzepts zum grundlegenden Architekturmuster des Gesamtsystems ein. Vielmehr betonen sie, dass nicht alle Teile eines Systems als Plugin modellierbar seien, und nennen als Gegenbeispiele die Hauptanwendung selber oder Module mit direkten Code-Abhängigkeiten zu anderen Modulen.

**Marquardt** erläutert in [Mar06] einen ganzen Satz an Entwurfsmustern, die bei der Entwicklung von Plugin-Systemen zum Einsatz kommen können. Zentrales Element dieses Satzes ist das Muster des *Plug-In*s, welches eine erst zur Laufzeit zu aktivierende Komponente[21] beschreibt, die aus der Anwendung ausgelagerte Funktionalität enthält. Die Anwendung definiert einen oder mehrere Plugin-Verträge (*Plug-In Contract*). Jeder dieser Verträge umfasst eine Schnittstellenspezifikation, die von allen Plugins der jeweiligen Art implementiert werden muss,

[21] Zur Betrachtung von Plugins als Komponenten siehe Unterabschnitt 2.2.5.

und eine von der Anwendung bereitgestellte Schnittstelle (z. B.. ein Rahmenwerk, auch *Plug-In Context* genannt). Ein Plugin kann auch selber wieder eigene Plugin-Verträge (*Plug-In Subcontract*) anbieten.

Als Teil des Plugin-Vertrages wird üblicherweise das Registrierungsverfahren (*Plug-In Registration*) sowie die Lebenszyklen von Plugin-Typen und -Instanzen (*Plug-In Lifecycle*) festgelegt. Die Registrierung kann deklarativ (durch passive Bereitstellung von Plugin-Code und -Daten) oder aktiv (durch Methodenaufrufe vom Plugin an die Verwaltung) geschehen. Der Lebenszyklus eines Plugin-Typen umfasst laut Marquardt mindestens die Registrierung, der Lebenszyklus einer Plugin-Instanz zumindest Laden, Aktivieren, Deaktivieren und Entladen, wobei Plugins auf die Ereignisse reagieren können sollen. Im Falle dynamischer Plugin-Systeme verstärkt sich die Wichtigkeit der korrekten Einhaltung des Plugin-Vertrags, insbesondere der Lebenszyklen.

Um die Anwender nicht mit der individuellen Installation vieler kleiner Plugins zu belasten, bieten die Muster *Plugin Package* und *Plugin-based Product* die Möglichkeit, geeignete Zusammenstellungen auszuliefern. Während ein Paket in erster Linie Code und begleitende Daten (z. B. Icons) eines Plugins zwecks gemeinsamer Installation bündelt, umfasst ein *Plugin-based Product* eine Kombination mehrerer Plugins, die häufig in dieser Zusammenstellung genutzt werden oder zumindest einen sinnvollen Satz an Grundfunktionalität bereitstellen.

Marquardt nennt als Anwendungsbeispiele für seine Plugin-Entwurfsmusterfamilie eine Reihe von Softwaresystemen, die sowohl große Anwendungen mit spezifischen Plugin-Schnittstellen (wie z. B. die NPAPI, siehe Unterabschnitt 3.1.1) als auch vollständig Plugin-basierte Systeme wie z. B. Eclipse (siehe Abschnitt 3.2) umfassen. Die Beschreibung letzterer durch das Muster wird durch die Variante der Plugin-Subkontrakte ermöglicht. Darüber hinaus führt Marquardt Beispiele für Systeme an, die softwareseitige Plugin-Mechanismen nutzen, um hardwareseitige Erweiterungen in ein Gesamtsystem zu integrieren, darunter auch Schnittstellen für Gerätetreiber in Betriebssystemen (siehe hierzu auch Unterabschnitt 3.3.9).

Marquardt weist darauf hin, dass die Einhaltung von Plugin-Verträgen nicht vollständig durch technische Mittel erzwungen werden kann und empfiehlt daher den Anbietern von Plugin-Schnittstellen, Einfluss auf die Entwicklung von Plugins auszuüben und Richtlinien z. B. für die Oberflächengestaltung festzulegen.

### 2.4.4 Muster zur Dekomposition von Plugins

In der Literatur zu Plugin-Systemen, insbesondere zu Eclipse und NetBeans, finden sich Richtlinien zur Dekomposition von Funktionalität in Plugins. Einige dieser Richtlinien lassen sich als Entwurfs- oder Architekturmuster betrachten, auch wenn die Autoren der Richtlinien diese Sichtweise nicht eingenommen haben.[22] Diese Muster

[22] Die Namen der von den Autoren nicht als solche beschriebenen Entwurfsmuster sind vom Autor dieser Arbeit vergeben, lehnen sich aber an Begriffe an, die von den jeweiligen Autoren verwendet werden.

behandeln überwiegend nicht die Ebene des objektorientierten Entwurfs einer Klassenhierarchie, sondern die Ebene der Systemarchitektur, auf der die Komposition von Komponenten, Modulen oder Plugins beschrieben wird.

**Library Plugin** ist ein sehr einfaches Muster, welches in allen Plugin-Systemen auftritt, sobald Funktionalität von Drittanbietern, die nicht in Plugin-Form, sondern als Code- oder Klassenbibliothek ausgeliefert wird, eingebunden werden soll. In so einem Fall müssen Plugins, die so eine Bibliothek benötigen, die statische Abhängigkeit gegenüber dem Plugin-System deklarieren können. Das übliche Vorgehen ist, die Bibliothek mit einem funktionslosen Plugin-Mantel zu versehen, der lediglich der Kompatibilität zum Konfigurationsmechanismus des Plugin-Systems dient. Ein Nachteil dieses Verfahrens ist, dass die Bibliothek des Drittanbieters nicht unverändert an Anwender des Plugin-Systems ausgeliefert werden kann. Eclipse-Entwickler finden eine Beschreibung des Vorgehens in [GB03, S. 96].

**Bridge Plugin** [BTW07, Anhang A.5] ist ein Muster zur Verbindung zweier Plugins, die zwar unabhängig voneinander bereits eine eigenständige Funktionalität bieten, aber deren Kombination zusätzliche Funktionalität hervorbringt. Das Muster vermeidet eine Abhängigkeit zwischen den beiden Plugins, indem ein drittes Plugin erstellt wird. Dieses dritte Plugin erweitert beide Plugins über deren jeweilige Erweiterungsschnittstellen und verbindet so deren Funktionalität indirekt. Derart funktional abhängige Module werden in der deklarativen Konfiguration von NetBeans (siehe Unterabschnitt 3.3.2) explizit unterstützt, indem sie automatisch (*eager*) aktiviert werden, sobald beide Basis-Plugins aktiviert sind.

**Self-Extending Plugin** basiert auf der *Fair Play Rule* [GB03, S. 66]. Ein Plugin, welches erweiterbar ist, aber gleichzeitig eine Minimal- oder Standardfunktionalität anstelle der möglichen Erweiterungen bietet, soll diese Standardfunktionalität als Erweiterung für seinen eigenen Erweiterungspunkt implementieren (siehe Unterabschnitt 3.2.1.8). Dieses Muster vermeidet explizite Fallunterscheidungen, verbietet aber eine irreflexive Erweiterungsrelation. Viele zentrale Plugins der Eclipse-IDE folgen selbstverständlich dieser Regel.

**Separate API Plugin** beschreibt die Trennung von Schnittstelle und Implementierung in separate Plugins. So können Schnittstelle und Implementierung unterschiedliche Sichtbarkeitsgrade erhalten und getrennt versioniert werden. Es besteht nur eine statische Abhängigkeit von der Implementierung zur Schnittstelle, weitere Plugins brauchen ebenfalls nur auf die Schnittstelle zu verweisen. Um das Schnittstellenplugin funktionstüchtig zu halten, ist allerdings eine funktionale Abhängigkeit zur Implementierung notwendig, die zur Laufzeit konfiguriert werden muss.

Die Verwendung dieses Musters wird insbesondere in der NetBeans-Entwicklung [BTW07, Anhang A.3, S. 555f.] empfohlen (siehe Unterabschnitt 3.3.2). Eine diesbezügliche Empfehlung für Eclipse-Entwickler beschränkt sich darauf, individuelle Packages für Schnittstelle und Implementierung anzulegen [*Explicit*

*API Rule, Stability Rule, Defensive API Rule*, GB03, S. 266]. Die schwächere Trennung wird unter anderem durch die OSGi-Plattform nahegelegt, welche Sichtbarkeitsgrade und Versionierung auch auf Package-Ebene unterstützt (siehe Unterabschnitt 3.2.1.6).

**Layered Plugins** ist angeregt durch die *STRATA Rule* [GB03, S. 356]. Die Regel beschreibt ein Muster zur Trennung von inhaltlich zusammenhängender Funktionalität in mehrere Plugins, das speziell auf das Eclipse-System (siehe Unterabschnitt 3.2.1.8) zugeschnitten ist. Zum einen soll Kernfunktionalität von der zugehörigen GUI-Implementierung getrennt, zum anderen soll programmiersprachenunabhängige Funktionalität von programmiersprachenabhängiger Funktionalität getrennt werden. Der Vorteil der Aufteilung ist, dass die Abhängigkeiten der individuellen Plugins entsprechend gegliedert werden – jedes einzelne Plugin hat weniger Abhängigkeiten als die Kombination. So kann Kernfunktionalität auch ohne GUI oder sprachunabhängige Funktionalität auch ohne die spezielle Programmiersprache von anderen Plugins genutzt werden.

Das Muster beschreibt allgemein eine Zergliederung thematisch zusammenhängender Funktionalität entsprechend mehrerer Architekturschichten des Gesamtsystems (im zitierten Fall einer IDE). In der verallgemeinerten Form ist es auch auf andere Plugin-Systeme übertragbar, sofern dort eine Definition von Architekturschichten vorgenommen werden kann. Durch die Verteilung der Abhängigkeiten in kleineren Mengen auf mehrere Plugins unterstützt das Gesamtsystem insgesamt mehr mögliche Konfigurationen.

**Cooperating Plugins** [Mar06, S. 327] ist ein allgemeines Muster, welches die Aufteilung inhaltlich zusammenhängender, aber auf unterschiedlichen technischen Schichten oder Architekturschichten angesiedelter Funktionalität auf mehrere Plugins beschreibt. Der inhaltliche Zusammenhang der Plugins wird durch einen gemeinsamen Identifikator ausgedrückt, der es der Plugin-Verwaltung erlaubt, eine Gruppe von Plugins gleichzeitig zu aktivieren.

Dieses Muster kann als Verallgemeinerung der Muster *Bridge*, *Separate API Plugins* und *Layered Plugins* betrachtet werden. Als allgemeinere Variante gibt es allerdings auch im Vergleich zu den anderen Mustern weniger konkrete Strukturen zur Umsetzung des Musters vor. Insbesondere bietet die Verknüpfung über einen gemeinsamen Identifikator weniger Strukturierungsmerkmale als die in den anderen Mustern implizit geforderten, explizit zu deklarierenden statischen und dynamischen Abhängigkeiten zwischen den Plugins.

### 2.4.5 Diskussion

Alle Entwurfsmuster der vorangegangenen vier Gruppen eignen sich, die Intention und Struktur von Plugin-Systemen klarer zu beleuchten. Eine Präzisierung und Klärung der Muster im Sinne eines Musterkataloges ist nicht Ziel dieser Arbeit, daher soll diese Richtung nicht weiter verfolgt werden. Allerdings können einige der vorgenannten Muster bei der Identifikation (siehe 4.1) und Modellierung (siehe Kapitel 4) der zentralen Konzepte von Plugin-Systemen hilfreich sein.

Unter den Mustern zur Dekomposition von Plugins ist bemerkenswert, dass die Muster *Separate API Plugin*, *Layered Plugins* und *Cooperating Plugins* die Aufteilung von Funktionalität auf Plugins nach technischen zusätzlich zu inhaltlichen Aspekten vorschlagen. Damit laufen diese Muster genau einem der Merkmale entgegen, welche in Unterabschnitt 2.2.5 als kennzeichnend für Plugin-Architekturen im Gegensatz zu komponentenbasierten Systemen herausgestellt wurden, nämlich der inhaltlich-funktional motivierten Dekomposition. Die Ursache für den gegenläufigen Trend dürfte darin zu suchen sein, dass diese Vorschläge im Kontext so genannter „reiner" Plugin-Systeme ins Spiel gebracht werden, welche nicht nur die Erweiterbarkeit eines Softwaresystems, sondern auch dessen gesamte Architektur abbilden müssen.

Die in diesem Abschnitt vorgestellten Entwurfsmuster dienen speziell Software-Entwicklern, also Programmierern oder Software-Architekten, bei der Gestaltung der Aufgaben, Beziehungen und Abhängigkeiten von Klassen und Schnittstellen in der objektorientierten Softwareentwicklung. Meyer kritisiert in [Mey97, Abschnitt 4.2, S. 71f.], dass Entwurfsmuster als bloße Beschreibungen ähnlicher Klassen und Beziehungen, die in jedem Softwaresystem erneut angepasst implementiert würden, keine ausreichende Wiederverwendbarkeit brächten. Besser sei es, Entwurfsmuster als wiederverwendbare Softwarekomponenten basierend auf generischen abstrakten Klassen bereitzustellen.[23] Meyer verwendet den Begriff *aufgeschobene Klasse* (*deferred class*) für eine Klasse, welche nicht sämtliche ihrer Elemente selber implementiert, sondern einige zur Implementierung durch Unterklassen offen lässt [vgl. Mey97, S. 504ff.]. Eine aufgeschobene Klasse stellt eine Komponente dar, welche Freiräume zur Implementierung weiterer Funktionalität im Rahmen einer von ihr fest vorgegebenen Struktur oder Verhaltensweise lässt. In [Arn04] wurden die bekannten Entwurfsmuster aus [GHJV98] auf ihre Implementierbarkeit als Softwarekomponenten im Meyerschen Sinne untersucht und, sofern möglich, passende Rahmenwerke implementiert.

Die Eigenschaften aufgeschobener Klassen passen auf den ersten Blick hervorragend zu Plugin-Architekturen, welche ebenfalls Freiräume zur Erweiterung durch Plugins lassen. Die in Unterabschnitt 2.4.1 vorgestellten Entwurfsmuster *Factory* und *Separated Interface* verwenden eine ähnliche Technik. Allerdings muss hier angemerkt werden, dass eine aufgeschobene Klasse durch Spezialisierung (Vererbung) erweitert wird – eine Technik, die zu einer statischen Abhängigkeit zwischen konkreter und generischer Klasse führt und nur in Zusammenarbeit mit anderen Entwurfsmustern, welche die konkrete Klasse eines Plugins im erweiterbaren System bekannt machen, einsetzbar ist. Daher lassen sich aufgeschobene Klassen gut in Rahmenwerken einsetzen, welche sowohl zur Implementierung einer Plugin-Verwaltung als auch zur Implementierung von erweiterbarer Funktionalität innerhalb eines Plugin-Systems nötig sind. Das Muster *Pluggable Factories* ist eine offensichtliche Anwendung dieser Idee.

Es ist bei der Betrachtung von Mikroarchitekturen wie den Entwurfsmustern ein weit verbreitetes Phänomen, die nebenläufige Natur der modellierten Systeme zu vernachlässigen. Die Muster klären im Wesentlichen die Aufgabenverteilung und geben allenfalls gelegentlich noch eine Reihenfolge der Arbeitsschritte (Methodenaufrufe)

[23] Meyer verwendet einen diffusen Komponentenbegriff (*reusable component*), der zwischen Modulen, Objekten und Klassen angelegt ist. Er unterscheidet sich von den in Unterabschnitt 2.2.1 vorgestellten Definitionen, da er z. B. Rahmenwerke oder Bibliotheken als Sammlungen von Komponenten bezeichnet. [vgl. Mey97, S. 72f.]

vor. Effekte, die durch die Kombination mehrerer Entwurfsmuster innerhalb eines Systems oder auch nur durch nebenläufige Nutzung der Klassen eines einzelnen Entwurfsmusters auftreten können, werden nicht behandelt. Das in dieser Arbeit entwickelte petrinetzbasierte Plugin-Konzept bringt hier einen klaren Vorteil.

## 2.5 Agentenorientierte Softwareentwicklung

Das Paradigma der agentenorientierten Softwareentwicklung (*Agent-oriented Software Engineering, AOSE*) verbindet Konzepte aus der verteilten künstlichen Intelligenz, dem Gebiet der verteilten Systeme und der Softwareentwicklung, um eine neue Abstraktionsebene für Softwareentwickler zu schaffen. Das dem Paradigma zugrunde liegende architekturprägende Konzept ist das Multiagentensystem (*Multi-Agent System, MAS*). Durch dieses Konzept werden drei Ebenen der Systemarchitektur in den Fokus des Entwicklers gerückt: Die in einem MAS als elementare Systemkomponenten enthaltenen *Agenten* können Konzepte aus der KI nutzen, um flexibel und autonom Entscheidungen zu treffen (Mikro-Ebene). Die Kommunikation zwischen den Agenten in einem MAS benötigt zwecks Strukturierung und Interoperabilität Kommunikationsstandards, welche sich mit Syntax und Semantik von Nachrichten sowie deren konsistenter Verwendung in komplexen Interaktionsmustern zur Koordination der Agenten im MAS beschäftigen (Interaktionsebene). Das gewünschte konzertierte Zusammenspiel vieler Agenten in einem MAS erfordert darüber hinaus Konzepte der Soziologie zur Modellierung von Agenten-Organisationen oder -Gesellschaften (Makro-Ebene).

Wie in den meisten Softwareentwicklungsparadigmen gibt es auch in der AOSE viele voneinander abweichende Definitionen zentraler Begriffe wie z. B. *Agent* und *MAS*. Der uneinheitliche Stand wird durch den Umstand weiter gefördert, dass viele der unter dem Paradigma der AOSE zusammengefassten Techniken und Methoden ihre Schwerpunkte wahlweise in einer der drei Ebenen setzen und daher nur schwer direkt vergleichbar sind.

Im Folgenden werden zunächst in Unterabschnitt 2.5.1 die Grundbegriffe der AOSE entlang der drei Ebenen eingeführt. Anschließend wird in Unterabschnitt 2.5.2 auf die Verwendung von Petrinetzen in der AOSE eingegangen, insbesondere auf die referenznetzbasierte MAS-Referenzarchitektur Mulan. Da diverse Autoren einen Übergang von der CBSE zur AOSE vorschlagen, sind deren Thesen Diskussionsgegenstand des Unterabschnitts 2.5.3.

### 2.5.1 Grundbegriffe

Die Idee, die Begriffe der Agenten und Multiagentensysteme aus der (Verteilten) Künstlichen Intelligenz und der Robotik auf die Softwareentwicklung zu übertragen, kam Anfang der 90-er Jahre des vorigen Jahrhunderts auf. Die Entwicklung ging aus zwei Richtungen voran: Zum einen entwickelten Shoham und andere unter der Überschrift agentenorientierte Programmierung (*Agent-oriented Programming, AOP*) wissensorientierte, regelbasierte und mit Kommunikationsprimitiven versehene Programmiersprachen [siehe Sho93]. Ansätze aus dieser Richtung sind durch starke Betonung der Mikro-Ebene geprägt. Für die allgemeine Betrachtung von Plugin-Systemen sind

die Konzepte dieser Ebene nur wenig hilfreich, da die Entscheidungsfindung innerhalb von Plugins eine anwendungsspezifische Problematik darstellt. Die Konzepte der Mikro-Ebene werden daher nur kurz in Unterabschnitt 2.5.1.1 angerissen.

Zum anderen entstanden unter dem Oberbegriff der agentenbasierten Softwareentwicklung (*Agent-based Software Engineering*) die Definitionen von (Software-)Agenten und Agentenkommunikationssprachen. Treibender Faktor war die Erkenntnis, dass isolierte Einzelsysteme den künftigen Anforderungen nicht mehr gerecht werden könnten, und dass stattdessen zunehmend Interoperabilität zwischen Softwaresystemen ermöglicht werden muss. Ansätze aus dieser Richtung betonen die Interaktionsebene. Es entstand zunächst unter dem Dach des „Knowledge Sharing Effort“ (KSE) – einem Projekt der amerikanischen „Advanced Research Projects Agency“ (ARPA) – die „Knowledge and Query Manipulation Language“ (KQML) [FW93] sowie das „Knowledge Interchange Format“ (KIF) [Gen95]. Die im Wesentlichen von Forschern aus dem Bereich der künstlichen Intelligenz besetzten Arbeitsgruppen wandten sich dann verstärkt dem Thema der Multiagentensysteme zu [siehe GK94]. 1996 wurde die „Foundation for Intelligent Physical Agents“ (FIPA) gegründet, welche sich um die Standardisierung von Agententechnologie und um die Behebung der Schwächen des KQML-Ansatzes bemüht. Die von der FIPA erarbeiteten Standards haben ebenfalls die Interoperabilität von Software-Agenten zum Ziel [Vgl. Pos05]. Die zentralen Konzepte der Interaktionsebene werden auf Basis der FIPA-Spezifikationen in Unterabschnitt 2.5.1.2 dieser Arbeit vorgestellt.

Der Begriff der Agentenorientierten Softwareentwicklung (*Agent-oriented Software Engineering, AOSE*) festigte sich spätestens im Jahr 2000 mit der Gründung der gleichnamigen Workshop-Reihe. Zu dieser Zeit ist auch die Makro-Ebene als nützliche Ebene in der AOSE erkannt worden.

Für die Zwecke dieser Arbeit sind die aus der KI stammenden Techniken und Methoden zur Bearbeitung von Fragestellungen innerhalb der Mikro-Ebene der Agenten weniger relevant. Die Techniken und Methoden der Makro-Ebene zur Gestaltung von Organisationen haben dagegen einen starken strukturierenden Effekt auf die resultierende Softwarearchitektur. Aus dem Bereich der Kommunikationsstandards ist vor allem das Konzept der Interaktionsprotokolle für die Softwaretechnik interessant, da es – ähnlich den Verbindern in einer ADL (siehe Abschnitt 2.3) oder vielen Entwurfsmustern – Grundbausteine zur losen Kopplung von Systemkomponenten in einem Gesamtsystem bietet.

#### 2.5.1.1 Agenten und Multiagentensysteme (Mikro-Ebene)

Der Agentenbegriff existiert schon in der klassischen Künstlichen Intelligenz (KI) und bezeichnet dort ein einzelnes System zur Lösung von Problemen. Aus der verteilten Künstlichen Intelligenz (VKI) und den verteilten Systemen ist dann der Agentenbegriff entstanden, der im Zusammenhang mit Multiagentensystemen verwendet wird. Bei der Definition dieses Agentenbegriffs fällt immer auch der Begriff „Umwelt“.

Häufig zitiert wird die mehrfach von Jennings und Wooldridge verwendete und überarbeitete Definition, die aus der ausführlichen Diskussion des Agentenbegriffs in [WJ95] hervorgegangen ist:

> „An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.“ [Jen00, S. 280]

Dabei bedürfen eigentlich alle Begriffe weiterer Erklärung:

- Agenten sind *gekapselt*, d.h. sie haben klar umrissene Grenzen und Schnittstellen.
- Agenten sind in einer *Umwelt* eingebettet, die sie durch Sensoren wahrnehmen und durch Effektoren beeinflussen können.
- Agenten sind zu *flexiblen Aktionen* fähig, d.h. sie können sowohl zeitnah auf Ereignisse *re*agieren als auch von selbst agieren (d.h. *pro*aktiv handeln).
- Agenten sind *autonom*, d.h. sie können ihren Zustand und ihr Verhalten selbst bestimmen.
- Agenten sind für einen bestimmten *Zweck* geschaffen und verfolgen entsprechende Ziele.

Diese Definition, die von Jennings und Wooldridge selber als *schwache* Definition des Begriffs gesehen wird, kann in etwa als kleinster gemeinsamer Nenner für die meisten Agentensysteme gesehen werden.

Für die *starke* Definition des Agenten führen Jennings und Wooldridge in [WJ95] zusätzliche Eigenschaften an. Diese Eigenschaften betonen die Verwurzelung des Agentenbegriffs in der Künstlichen Intelligenz. Sie laufen darauf hinaus, dass Agenten mit menschenähnlichen Mechanismen und Begriffen modelliert werden, wie z. B. Wissen, Glauben, Zielen und Verpflichtungen.

Die von Ferber in [Fer01] gegebene Agenten-Definition geht in eine ähnliche Richtung. Zunächst zeichnen sich Agenten durch das selbständige, zielbestimmte Agieren in einer Umwelt aus. Darüber hinaus führt Ferber explizit die Kommunikation zwischen Agenten, die Möglichkeit, Dienste zu offerieren, und den Besitz eigener Ressourcen an. Er betont, dass Agenten die Umwelt nur unvollständig wahrnehmen und repräsentieren können. Jeder Agent muss sein Verhalten darauf ausrichten, mit seinen beschränkten Informationen, Ressourcen, Fähigkeiten, Kommunikationsmöglichkeiten und kognitiven Modellen seine jeweiligen Ziele zu erreichen.

Ferber unterscheidet zwischen rein kommunikativen Agenten (so genannten Software-Agenten) und rein situierten (physischen) Agenten. Für einen Software-Agenten, der sich in einem offenen Rechnernetz befindet, besteht die Umwelt ausschließlich aus anderen Agenten, mit denen er kommunizieren kann. Seine Effektoren und Sensoren sind somit auf den Nachrichtenempfang und -versand reduziert. Anders bei rein situierten Agenten, für die die Umwelt in der Regel ein metrischer, physikalischer Raum ist, der nur über Sensoren wahrgenommen und durch Aktionen verändert werden kann, d.h. nur indirekte Kommunikation zwischen den Agenten zulässt.

Auch der Begriff des Multiagentensystems (*Multi-Agent System, MAS*) wird in mehreren Variationen definiert. Die Interpretationen verschiedener Autoren unterscheiden sich danach, ob der jeweilige Autor mehr Wert auf die physische Realisierung des Gesamtsystems oder auf die logische Struktur der Agentengemeinschaft legt.

Ferber definiert in [Fer01] ein Multiagentensystem operational:

„Der Begriff Multiagentensystem (MAS) bezeichnet ein System, das aus folgenden Elementen besteht:

1. Eine Umwelt $\mathsf{E}$. $\mathsf{E}$ ist ein Raum, der im Allgemeinen ein Volumen hat.
2. Eine Menge von Objekten, $\mathsf{O}$. Diese sind situiert, das bedeutet, dass zu einem beliebigen Zeitpunkt jedem Objekt eine Position in $\mathsf{E}$ zugewiesen werden kann. Objekte können von den Agenten wahrgenommen, erzeugt, modifiziert und gelöscht werden.
3. Eine Menge von Agenten, $\mathsf{A}$. Diese repräsentieren die aktiven Objekte ($\mathsf{A} \subseteq \mathsf{O}$) des Systems.
4. Eine Menge von Beziehungen, $\mathsf{R}$, die Objekte miteinander verbinden.
5. Eine Menge von Operationen, $\mathsf{Op}$, damit Agenten Objekte empfangen, erzeugen, konsumieren, verändern und löschen können.
6. Operatoren mit der Aufgabe, die Anwendung dieser Operationen und die Reaktion der Umwelt auf die entsprechenden Veränderungsversuche darzustellen. Wir bezeichnen diese Operatoren als die Gesetze des Universums.“ [Fer01, S. 31]

Weiß nimmt im Prolog zu [Wei00] eine gröbere Sicht ein und zitiert folgende Zusammenstellung der Eigenschaften von Multiagentensystemen aus [JSW98]:

„The characteristics of MAS are:

- each agent has incomplete information, or capabilities for solving the problem, thus each agent has a limited viewpoint;
- there is no global system control;
- data is decentralized; and
- computation is asynchronous.“ [Wei00]

Darüber hinaus führt er in Anlehnung an die Diskussion in [HS98] eine Reihe von Eigenschaften an, in denen Multiagentensysteme variieren können. Dazu gehören Fragen von der bloßen Anzahl der Agenten über die Unterschiede in ihren technischen Fähigkeiten bis zur Komplexität ihres „Denkapparats“ oder den Wahrnehmungs- und Interaktionsmöglichkeiten mit ihrer Umgebung.

Die eben vorgestellten Definitionen der Begriffe Agent und Multiagentensystem beschreiben „Systeme aus Agenten“. Modelliert und implementiert werden Wissen und Verhalten der individuellen Agenten auf der Mikro-Ebene. Dementsprechend wird in Arbeiten aus dieser Forschungsrichtung diskutiert, welches kognitive Modell seinem Verhalten zugrunde liegt. Verschiedene „Agenten-Architekturen“ unterscheiden sich nach dieser Sichtweise darin, ob und wie der Agent Ziele und Wissen repräsentiert oder auf welche Weise seine Aktionen und Reaktionen zu Stande kommen, also welchem Organisationsprinzip der Entscheidungsprozess im Agenten folgt. Einen Überblick über verschiedene reaktive, deliberative und interaktive Architekturen gibt Müller in [Mül97]. Besonders prominent ist beispielsweise die BDI-Architektur (*Beliefs, Desires, Intentions*), welche menschliches Planen im Hinblick auf zu erreichende Ziele imitiert.

Das Verhalten des Gesamtsystems ergibt sich erst in zweiter Linie aus dem Zusammenspiel der Verhaltensweisen der einzelnen Agenten. Für einen systemübergreifenden Entwurf fehlen zunächst die Ausdrucksmittel, welche auf den beiden folgenden Ebenen nachgereicht werden.[24]

#### 2.5.1.2 Kommunikation (Interaktionsebene)

Im Rahmen der Softwareentwicklung sind in erster Linie Multiagentensysteme aus reinen Software-Agenten von Interesse. Wie bereits angesprochen, zeichnen sich diese dadurch aus, dass sie in einer Umgebung situiert sind, die aus anderen Agenten besteht, und dass sie diese Umgebung ausschließlich durch Nachrichtenkommunikation wahrnehmen und beeinflussen können. Daraus ergibt sich, dass die Implementierung und Organisation der Nachrichtenkommunikation einen weiteren Schwerpunkt im Rahmen der Forschung an Multiagentensystemen darstellt.

Kennzeichnend für diesen Aspekt von Multiagentensystemen sind zwei zentrale Begriffe: Plattformen und Sprechakte. Eine Agenten-*Plattform* stellt das softwaretechnische Analogon zum Aufenthaltsort physischer Agenten dar. Unter einer Plattform ist die Gesamtheit aus physischer Hardware des Rechners, dem Betriebssystem und weiterer Software zur Ausführung der Agenten zu verstehen. Sie bietet die grundlegende Infrastruktur, welche erst eine Nachrichtenkommunikation ermöglicht. Nachrichten sind *Sprechakte* (in Anlehnung an eine Betrachtungsweise der menschlichen Kommunikation, siehe [Sea69]). Ein Sprechakt umfasst vier Ebenen: die eigentliche Laut- oder Zeichenfolge, eine logisch formulierbare Aussage über Objekte und deren Beziehungen, eine Handlung des Sprechenden (z. B. Auffordern, Fragen, Bestätigen etc.) und eine Wirkung beim Empfänger. Da bei den meisten Handlungsarten eine Reaktion vom Empfänger erwartet wird, werden Abfolgen kausal verbundener Nachrichten als Konversationen betrachtet, welche in der Regel entlang bekannter Interaktionsprotokolle verlaufen.

Von den Ergebnissen der eingangs in diesem Abschnitt erwähnten Diskussionsgruppen und Standardisierungsgremien haben sich die Arbeiten der FIPA (*Foundation for Intelligent Physical Agents*) weitgehend durchgesetzt. Ziel der FIPA ist das Erarbeiten von „Software-Standards für heterogene und interagierende Agenten und agentenbasierte Systeme“ [Vgl. FIP, `/about/index.html`]. Der Schwerpunkt der Arbeit liegt auf der Interoperabilität zwischen Agenten, die in unterschiedlichen Programmiersprachen und/oder auf unterschiedlichen Agentenplattformen implementiert sind. Ein allgemein gültiger und weit verbreiteter Standard soll es allen Agenten ermöglichen, zumindest

[24] Um eine Analogie zu anderen Softwareentwicklungsparadigmen zu ziehen: In der Objektorientierung entspräche die Mikro-Ebene der Betrachtung des Innenlebens von Objekten. Ausgehend von der Definition von Objekten als Kombination aus Daten und Methoden läge der Diskussionsschwerpunkt auf den verschiedenen Techniken, wie Objekte Methoden implementieren können. Eine Diskussion auf dieser Ebene ist interessanterweise in der Objektorientierung nur eine Randerscheinung. In dieser Arbeit findet eine Diskussion auf dieser Ebene beim Vergleich der Semantik synchroner Kanäle mit imperativen Methodenaufrufen in Unterabschnitt 2.1.2.1 und Unterabschnitt 4.2.5.1 statt.

Auf der Mikro-Ebene fehlen übergreifende Konzepte zur Beschreibung von Beziehungen wie sie z. B. in der Objektorientierung mit Klassen, Vererbung und Polymorphie zur Verfügung stehen. Objektorientierte Softwareentwicklung zeichnet sich aber gerade dadurch aus, dass Analyse und Entwurf auf der Ebene der Klassen und ihrer Beziehungen stattfinden.

auf einer technischen Ebene erfolgreich zu kommunizieren. Auf semantischer Ebene kann dann zumindest die Tatsache kommuniziert werden, dass zwei Agenten sich eventuell doch nicht verstehen, weil sie über ein zu stark abweichendes Hintergrundwissen verfügen.

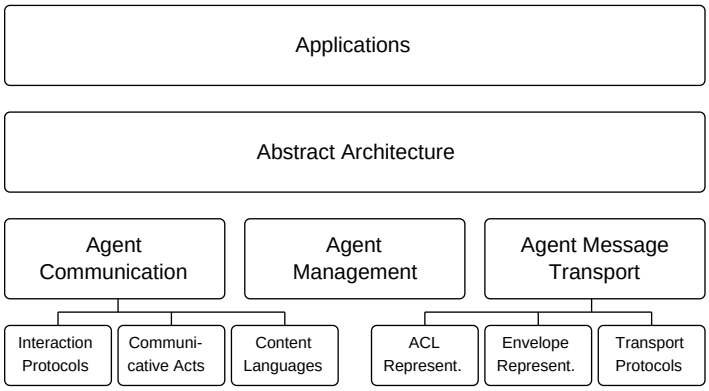



Abbildung 2.12: Thematische Einteilung der FIPA-Spezifikationen (nach [FIP, `/repository/bysubject.html`])

Die plattformübergreifende Kommunikation wird in verschiedenen Aspekten standardisiert, die durch die FIPA wie in Abbildung 2.12 dargestellt unterteilt werden. Die in der mittleren Zeile genannte „Abstrakte Architektur“ bezeichnet eine Spezifikation [FIP01a], die auf einer allgemeinen Ebene die notwendigen Elemente einer FIPA-konformen Standardisierung der Kommunikationsinfrastruktur definiert. Damit stellen alle Spezifikationen der FIPA, welche die unteren beiden Zeilen thematisieren, eine „konkrete Architektur“ dar, die den Anforderungen der abstrakten Architektur genügt. Die Implementierung konkreter Agentenplattformen ist nicht Teil der Spezifikationen, da auch die „konkrete Architektur“ nur Rahmenbedingungen für Interoperabilität festlegt.

Im Rahmen der konkreten Spezifikationen werden drei Teilbereiche behandelt: Nachrichten benötigen ein Transportmedium, also eine Kommunikationsinfrastruktur mit allen Beteiligten bekannter Schnittstelle. Der „Agent Message Transport“ genannte Block beschäftigt sich mit diesem Aspekt. Das „Agent Management“ stellt sicher, dass sich die Kommunikationspartner gegenseitig kontaktieren können, indem Adressen und Vermittlungsdienste definiert werden. Zugleich sollen Agenten trotz der heterogenen Plattformen einem einheitlichen Lebenszyklus unterliegen. Im Block „Agent Communication“ werden Details zu den verfügbaren Sprechakten, den formulierbaren Aussagen und ihrer Verwendung im Rahmen von Interaktionsprotokollen geregelt. Die Spezifikationen dieses Blocks definieren eine einheitliche *Agent Communication Language* (ACL) mit bekannter Syntax, Semantik und Pragmatik. Die Spezifikationen dieser drei Bereiche haben zu großen Teilen eine eher technische Natur. Die grundlegende Festlegung auf Sprechakte, die formulierbaren Aussagen und Handlungen und ihre

Verbindung zu Interaktionsprotokollen beschreiben aber zugleich die elementaren Ausdrucksmittel des agentenorientierten Systementwurfs.

Für die Betrachtung im Rahmen dieser Arbeit nicht relevant sind FIPA-Spezifikationen, die konkrete Gebiete für die Anwendungen („Applications“) von Agentensystemen im Blickfeld haben. Sie definieren, aufsetzend auf den Kommunikationsstandards der anderen FIPA-Spezifikationen, konkrete Dienste und Ontologien zu Themen wie audio-visueller Unterhaltung, persönlichen Assistenten, Netzwerkmanagement usw. Diese Spezifikationen modellieren in erster Linie Agenten und fachliche Domänen, sie sind daher nicht prägend für die AOSE.

**Agentenkommunikation** Die FIPA-ACL definiert eine Nachricht als einen kommunikativen Akt. Ein kommunikativer Akt besteht aus zwei Teilen, einer inhaltlichen Aussage und dem Performativ (der Handlung des Sprechakts). Die FIPA bietet in der „Communicative Act Library“ [FIP01b] einige häufig gebrauchte, mit Bezeichner und Semantik definierte Performative an, die zur Kommunikation verwendet werden können. Für den inhaltlichen Teil – die logisch formulierbare Aussage – des kommunikativen Aktes werden zum einem mehrere Sprachen (*content languages*) angeboten, von denen der Agent eine wählen kann. Ferner verwenden die Agenten fachspezifische Ontologien, um die Objekte und Beziehungen zu definieren, über welche Aussagen gemacht werden können. In jeder Nachricht ist sowohl das verwendete Performativ, die verwendete Inhaltssprache und die verwendete Ontologie explizit anzugeben, um Missverständnisse zu vermeiden. Ebenfalls anzugeben ist das Interaktionsprotokoll, dem die Konversation folgt, zu der die Nachricht gehört. Auch in diesem Gebiet definiert die FIPA einige häufig benötigte Interaktionsprotokolle, die Auswahl ist allerdings stark eingeschränkt.

Entwickler von FIPA-konformen Multiagentensystemen dürfen eigene Performative, Inhaltssprachen und auch Interaktionsprotokolle definieren und verwenden. Während jedoch das Angebot an Performativen und Inhaltssprachen für die meisten Anwendungen ausreichend ist, müssen Interaktionsprotokolle in der Regel anwendungsspezifisch definiert werden. Da Ontologien immer fachspezifisch sind, wird der Versuch eines einheitlichen Ontologie-Kataloges gar nicht erst unternommen.[25]

**Nachrichtentransport** Zur Kommunikation über ein Netzwerk greift die FIPA auf gängige Internet-Protokolle wie z. B. HTTP oder WAP zurück und definiert lediglich, wie Agentennachrichten über diese Protokolle verschickt werden können. Zwecks Transport werden Nachrichten ähnlich einer E-Mail mit einem Transportumschlag versehen, der es den Agentenplattformen erlaubt, die Nachricht ohne Rückgriff auf den eigentlichen Nachrichteninhalt über mehrere Plattformen bis zur Plattform der adressierten Agenten weiterzuleiten. Zur Kodierung der Nachrichten und Umschläge stehen zwei bis drei Alternativen zur Auswahl, die wiederum durch eigene Alternativen ergänzt werden dürfen.

[25] Die FIPA definiert allerdings ein paar fachspezifische Ontologien im Rahmen der Spezifikationen, welche sich mit konkreten Anwendungsgebieten beschäftigen.

**Agentenverwaltung** Agenten als elementare Systemkomponenten werden auf der durch eine Agentenplattform (*Agent Platform*) bereitgestellten physischen Infrastruktur eingesetzt. In jeder Plattform werden gemäß den FIPA-Spezifikationen mindestens drei Dienste angeboten: Neben dem Nachrichtentransportdienst (*Message Transport Service, MTS*) existieren ein Agentenverwaltungsdienst (*Agent Management Service, AMS*) und ein Dienstanbieterverzeichnis (*Directory Facilitator, DF*). Die beiden letzteren erlauben es Agenten, Adressen möglicher Kommunikationspartner zu erfahren, indem ein globales Verzeichnis nach gewünschten Kriterien durchsucht wird.

Der Agentenverwaltungsdienst stellt zudem die Schnittstelle zwischen den Agenten und der Funktionalität der Plattform bereit: Erzeugung und Terminierung sowie andere elementare Zustandswechsel eines Agenten können über das AMS angefordert werden. Die möglichen Zustandswechsel werden durch das in Abbildung 2.13 wiedergegebene Lebenszyklus-Diagramm vorgegeben.

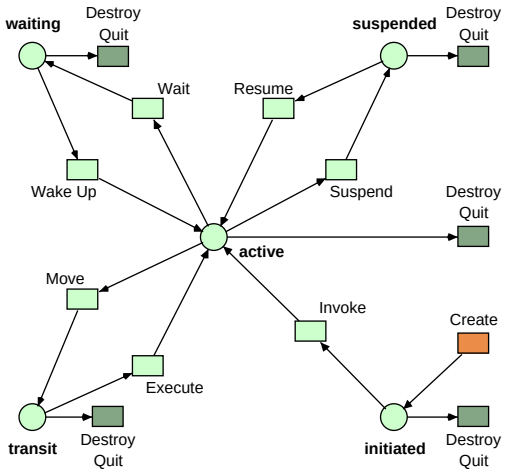



Abbildung 2.13: FIPA-Agenten-Lebenszyklus als Petrinetz [nach FIP00a, S. 9]

Der als Petrinetz[26] dargestellte Lebenszyklus eines Agenten beginnt mit seiner Erzeugung, die in zwei Schritten („Create“ und „Invoke“) geschieht. Danach befindet sich der Agent im Zustand „Active“, in dem er ganz normal arbeiten und Nachrichten versenden bzw. empfangen kann. Die Zustände „Waiting“ und „Suspended“ symbolisieren Schlafzustände, in denen der Agent keine Nachrichten empfangen kann. Der Zustand „Transit“ wird für die Migration von mobilen Agenten benötigt, wie sie in der „FIPA Agent Management Support for Mobility Specification“ [FIP00b] grob umrissen wird. Das Beenden eines Agenten ist sowohl mit Zustimmung des Agenten („Quit“) als auch ohne („Destroy“) möglich. Die FIPA definiert Auswirkungen dieser Zustände nur bezüglich des Nachrichtentransports, Plattformimplementierungen sind ansonsten frei in der technischen Umsetzung der Zustände.

[26] Zur Modellierung von Lebenszyklen als Petrinetz siehe Unterabschnitt 4.4.2.1.

Für die Softwareentwicklung auf der Basis von Multiagentensystemen sind die technischen Details der eben angesprochenen Gebiete allenfalls zur Implementierung von Ausführungsumgebungen, also Agentenplattformen relevant. Zudem handelt es sich bei Thematiken des Nachrichtentransports, der Kodierung und der Adressierung um Probleme aus der Disziplin der verteilten Systeme, welche keineswegs spezifisch für Multiagentensysteme sind. Die den Lösungen zugrundeliegenden Konzepte haben jedoch einen starken Einfluss auf die agentenorientierte Softwareentwicklung. Die Festlegung auf *asynchrone Nachrichtenkommunikation* stellt klar, dass Multiagentensysteme grundsätzlich nebenläufige Systeme sind. Die Kommunikation mit *Sprechakten* verdeutlicht, dass Agenten die Autonomie haben, auf Anforderungen anders als erwartet zu reagieren. Der Begriff des *Interaktionsprotokolls* betont zudem die Bedeutung von Prozessen im System.

In der Betonung dieser eben genannten Begriffe liegt der Unterschied der agentenorientierten Softwareentwicklung zu den Paradigmen der objektorientierten oder komponentenbasierten Softwareentwicklung begründet. Das Konzept des Multiagentensystems bietet wenig Anhaltspunkte zur Definition einer statischen Struktur mit klar definierten Schnittstellen. Stattdessen wird der dynamische Aspekt durch Nachrichten und Interaktionen betont. Die Spezifikation von Interaktionsprotokollen tritt an die Stelle der Spezifikation von Schnittstellen. Ein Analogon zu Klassen oder Komponenten wird in der Regel erst auf der Makro-Ebene definiert.

Ein weiterer Unterschied ergibt sich in der Art der Kapselung. Der Zustand und die Implementierung eines Agenten sind aufgrund der nicht deklarierten Schnittstellen von außen noch weniger wahrnehmbar als es bei Objekten oder Komponenten der Fall ist. Ein Software-Agent kann beliebige Nachrichten empfangen, seine Reaktion darauf ist nicht vorhersagbar. Die Kapselung wird durch Autonomie gestärkt. Um den Entwurf einer sinnvollen, übersichtlichen Software-Architektur zu ermöglichen, muss die Autonomie der Agenten wieder beschränkt werden. Ein erster Schritt in diese Richtung ist die Deklaration von Interaktionsprotokollen in Nachrichten – ein Agent gibt damit zu erkennen, dass er sich im Hinblick auf dieses Interaktionsprotokoll erwartungskonform verhalten wird. Die im folgenden Unterabschnitt erläuterte Makro-Ebene bietet hierzu darüber hinausgehende Konzepte zur sinnvollen Beschränkung der Autonomie.

Ein dritter Unterschied besteht in der grundsätzlichen Existenz globaler Verzeichnisse in einem FIPA-konformen Agentensystem. Zwar kann jeder Agent in seinem lokalen Wissen auch Adressen von Kommunikationspartnern halten, aber die Kontaktaufnahme muss in einem dynamisch konfigurierten, heterogenen System zwangsweise über Verzeichnisse oder Vermittler erfolgen. Zudem stellt jedes Multiagentensystem (zumindest aus dem Blickwinkel der FIPA-Spezifikationen heraus) eine dienstorientierte Architektur dar. Die Deklaration von Dienstangeboten stellt eine weitere (freiwillige) Einschränkung der Autonomie eines Agenten dar, welche schon der Makro-Ebene zugeordnet werden kann.

#### 2.5.1.3 Organisation und Rollen (Makro-Ebene)

Das Zusammenspiel mehrerer Agenten durch die Struktur ihrer Beziehungen und Interaktionen stellt die am stärksten zur Strukturierung einer Softwarearchitektur geeignete

Modellierungsebene dar. Die im vorigen Unterabschnitt erwähnten Interaktionsprotokolle stellen auf dieser Ebene lediglich ein primitives Element zur Koordination von Agenten dar. Manche Autoren bezeichnen diese Ebene als *Makro-Architektur* [vgl. WJ04, S. 19f.] oder auch als *MAS-Architektur* [vgl. ZO04, S. 261].

Die Makro-Ebene wird in der Regel erst im Rahmen der verschiedenen alternativ erarbeiteten Methoden betrachtet. Beispiele für solche Methoden sind Gaia [WJK00, ZJW03], MaSE [WD01, DeL05] oder Paose [CDD+07]. Beim Entwurf von Multiagentensystemanwendungen stehen neben der aus anderen Paradigmen bekannten funktionalen und strukturellen Dekomposition der Domäne insbesondere Rollen und Interaktionen im Fokus des Systemarchitekten. Fast alle publizierten AOSE-Vorgehensmodelle zu Systemanalyse und -entwurf sehen das Konzept der *Rolle* vor [vgl. Untersuchung in WJ04]. Rollen können zum einen Positionen und Verantwortlichkeiten in einer Organisation identifizieren, um das Multiagentensystem im Groben zu strukturieren. Zum anderen ist der Rollenbegriff eng mit der Technik der Interaktionsprotokolle verknüpft: Alle an einer Interaktion beteiligten Agenten nehmen eine Rolle im Rahmen dieser Interaktion ein. Anders herum lässt sich eine Rolle durch die Auflistung sämtlicher Interaktionen definieren, welche von der Rolle benötigt werden.

Das Konzept der Rolle stellt ein Analogon zu den strukturierenden Konzepten der objektorientierten oder komponentenbasierten Softwareentwicklung dar. Hierbei ist aber wiederum als Unterschied festzuhalten, dass die AOSE ein deutlich dynamischeres Konzept bietet. Ein Objekt ändert seine Klasse während seiner Lebenszeit nie, ebensowenig ändert eine Komponente während ihres Einsatzes ihre Schnittstelle. Ein Agent hingegen ist keine „Instanz“ einer Rolle. Er kann jederzeit Rollen einnehmen und wieder aufgeben, wenn seine Ziele oder Abmachungen mit Kooperationspartnern dies erfordern. Viele der bisher im Rahmen der AOSE vorgeschlagenen Methoden zum Systementwurf gehen allerdings vereinfachend davon aus, dass auch ein Agent in einem konkreten Multiagentensystem seine Rolle *nicht* wechselt.

Die gegenseitige Verzahnung von Rollen und Interaktionen führt dazu, dass bei Analyse und Entwurf agentenorientierter Softwaresysteme eine Matrix-artige Verflechtung von Struktur- und Prozesssicht stattfindet. Die gleichwertige Betrachtung von (Interaktions-)Prozessen und (Organisations-)Strukturen wird in den alternativen Methoden der AOSE in unterschiedlichen Graden ausgenutzt. Sofern die Gleichberechtigung in einer Methode durchgehalten wird, bietet die AOSE eine gut integrierte mehrfache Sicht auf das Softwaresystem. Eine ähnlich ausdrucksstarke Integration erreichen von den anderen in diesem Kapitel besprochenen Techniken und Paradigmen zur Softwareentwicklung lediglich einige Architekturbeschreibungssprachen.

Das Konzept der Rolle wird in einigen Methoden der AOSE durch weitere organisatorische Konzepte ergänzt. Fast noch auf der Interaktionsebene ist das Konzept des Dienstes angesiedelt, da auch die FIPA-Verwaltungsagenten durch ihr Dienstangebot definiert werden. Ein Agent, der einen Dienst anbietet, nimmt im Rahmen dieses Angebots die Rolle des Dienstleisters ein und verzichtet zugunsten eines verlässlichen Dienstangebots auf einen Teil seiner Autonomie. Er verpflichtet sich, sich konform zur Dienstanbieter-Rolle zu verhalten. Im Agent/Group/Role (AGR)-Modell [FG98, FMB05] werden Rollen im Kontext von Gruppen definiert. Das Gruppen-Konzept ersetzt das Konzept der physischen Plattform, wobei nun ein Agent Mitglied mehrerer Gruppen sein und in jeder Gruppe andere Rollen einnehmen kann.

Die Unterteilung in Mikro-, und Makro-Ebene verbindet die Multiagentensysteme mit der Soziologie, welche ähnliche Betrachtungen über menschliche Gesellschaften und Individuen vornimmt, und der Biologie, aus der beispielsweise Ameisenschwärme als Vorbild für emergentes Verhalten übernommen werden können. Die Fragestellungen der Koordination von Agenten lassen sich gut mit Konzepten aus der Organisationstheorie, Biologie und Soziologie bearbeiten, so dass aus diesen Forschungsgebieten neue softwaretechnische Konzepte gewonnen werden können.

Zur grafischen Notation greifen die meisten AOSE-Methoden auf angepasste Varianten der Unified Modeling Language (UML) zurück. Beispielsweise führt die „Agent-UML“ [AUML, siehe OPB00] Ergänzungen im Bereich der Modellierung von Interaktionen ein. In den Spezifikationen der FIPA werden AUML-Diagramme zur Illustration von Interaktionsprotokollen eingesetzt.

### 2.5.2 Agenten und Petrinetze

Petrinetze werden auf verschiedenste Weisen zur Modellierung von Agenten oder Multiagentensystemen eingesetzt. Aufgrund der guten Eignung von Petrinetzen zur Spezifikation von Prozessen ist der Einsatz zur Definition und zum Validieren von Interaktionsprotokollen naheliegend. Ein Ansatz dieser Art findet sich beispielsweise in [MSH02] oder [PPW02]. Rölke listet einige weitere Ansätze in [Röl04, Abschnitt 5.2.2] auf.

Über die reine Interaktionsmodellierung hinaus werden Petrinetze auch von verschiedenen Autoren zur Modellierung von Agenten eingesetzt. Rölke gibt in [Röl04, Abschnitt 5.2.1] einen Überblick über einige der verfolgten Ansätze. Viele der Ansätze führen Techniken aus dem Bereich der objektorientierten Petrinetze fort und ergänzen agentenorientierte Aspekte. Nur wenige Ansätze strukturieren ein ganzes Multiagentensystem mit Petrinetzen. Im Kontext dieser Arbeit ist vor allem der in [Röl04] vorgestellte Ansatz von Heiko Rölke von Interesse, da seine Modelle ebenso wie die Modelle in dieser Arbeit auf Referenznetzen (siehe Unterabschnitt 2.1.3) beruht. Das von ihm entwickelte Modell mit dem Namen Mulan (**Mul**ti**a**genten**n**etze) stellt eine Referenzarchitektur für Multiagentensysteme dar. Es gibt eine einheitliche Struktur für beliebige MAS vor, deren zentrale Elemente, der Agent und die Agentenplattform, in allen auf Mulan basierenden Systemen gleich aufgebaut sind.

Die Wahl von Referenznetzen für die Modellierung eines Multiagentensystems begründet sich zum einen darin, dass viele für Multiagentensysteme interessante Konzepte wie Nebenläufigkeit und Synchronisation, Lokalität und Mobilität, Objektorientierung und dynamisches Verhalten von diesem Petrinetzformalismus abgedeckt werden. Zum anderen erlauben die Referenznetze Dank ihrer operationalen Semantik die Ausführung des Modells. Damit kann das zunächst abstrakt spezifizierte System unter Beibehaltung der Repräsentierungssprache (d.h. weiterhin mit Netzen) ausführbar gemacht werden.

**Mulan** modelliert ein Multiagentensystem für rein kommunizierende Agenten, also Software-Agenten, die keine andere Umwelt haben als ihre Kommunikationspartner (siehe Diskussion in Unterabschnitt 2.5.1.1). Dementsprechend werden Plattformen zur Modellierung von Orten herangezogen (siehe Unterabschnitt 2.5.1.2).

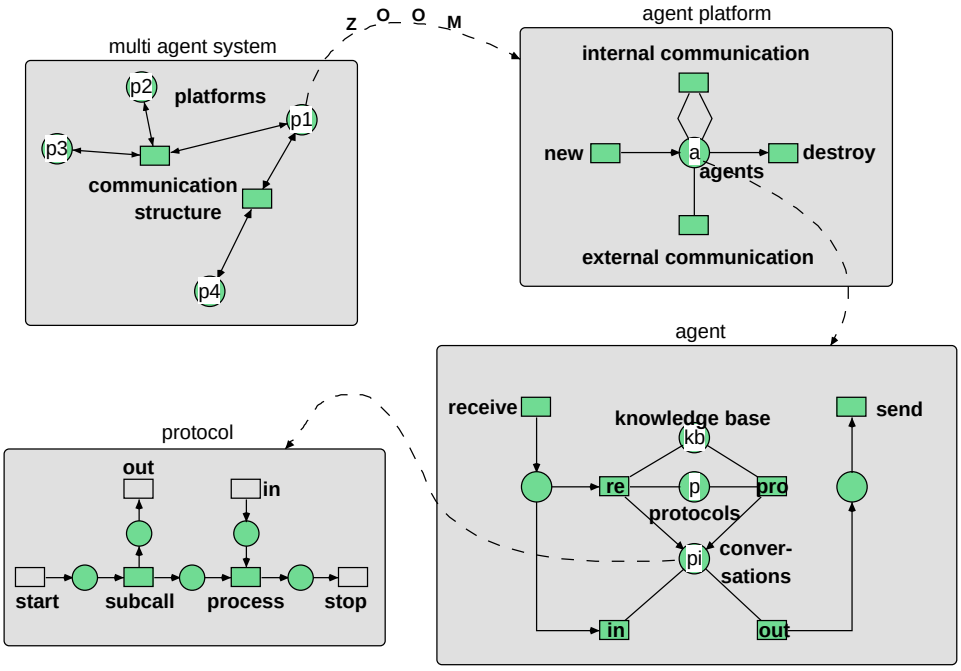



Abbildung 2.14: Überblick über die MULAN-Architektur

In Abbildung 2.14 sind die vier Ebenen der MULAN-Architektur zu sehen. Die oberste Ebene (oben links im Bild) stellt einen groben Blick auf die gesamte Struktur des *Multiagentensystems* dar, wobei die Plattformen und die Kommunikationsstrukturen zwischen den Plattformen modelliert werden. Die Struktur in diesem Teil der Abbildung ist exemplarisch, da die MULAN-Referenzarchitektur auf dieser Ebene eine Konkretisierung abhängig vom abzubildenden System vorsieht. Je nach der in einem konkreten MAS geplanten oder vorhandenen Infrastruktur sind andere Plattformstellen und Kommunikationsverbindungen zu modellieren.

Jede Plattform wird durch eine Marke auf einer Stelle des Systemnetzes repräsentiert, wobei die Marke wiederum eine Netzinstanz ist. Die Kommunikationsstruktur zwischen den Plattformen wird durch Transitionen modelliert, die über synchrone Kanäle (die entsprechenden Anschriften werden in Abbildung 2.14 der Übersichtlichkeit halber ausgeblendet) zwei Plattformen kurzfristig miteinander koppeln, um Nachrichten auszutauschen oder Agenten zu bewegen.

Alle Netzinstanzen, die die Plattformen repräsentieren, basieren auf einem einheitlichen Netzmuster, das oben rechts im Abbildung 2.14 zu sehen ist. Der mit ZOOM beschriftete Pfeil soll die Vergrößerung der Marke symbolisieren (was nicht mit einer Vergröberung der die Marke enthaltenden Stelle verwechselt werden darf).

Das *Plattformnetz* enthält eine zentrale Stelle, in der alle Agenten (wiederum durch Marken repräsentiert, die Netzinstanzen sind) gelagert werden, die sich auf dieser Plattform befinden. Die mit der Stelle verbundenen Transitionen beschreiben die primitiven Dienste, die eine Plattform den Agenten zur Verfügung stellen muss. Dazu gehören Kommunikationsmöglichkeiten und die Möglichkeit, Agenten zu erzeugen und zu terminieren. Zu den individuellen Agenten wird seitens der Plattform eine unveränderliche Schnittstelle definiert, welche genau zwei synchrone Kanäle zum Senden und Empfangen von Nachrichten umfasst. Diese kommunikativen Grundprimitive reichen aus, um beliebige Agentenkommunikation zu ermöglichen. Die Plattform implementiert ihren Anteil der Kommunikationsschnittstelle durch die alternativ aktivierbaren Transitionen „internal communication“ und „external communication“. Während erstere eine Nachricht direkt zwischen zwei Agenten der Plattform vermittelt, leitet letztere die Nachricht über die Transitionen des übergreifenden Systemnetzes an andere Plattformen weiter.

Die Agentenmarken im Plattformnetz sind wiederum Instanzen eines *Agentennetzmusters* (in Abbildung 2.14 rechts unten). Alle Agenten haben konzeptionell den gleichen Aufbau, so dass dieses Netz vom Entwickler eines anwendungsspezifischen Multiagentensystems nicht verändert zu werden braucht. Allerdings legt dieses Netz die Entscheidungsarchitektur des Agenten fest. Das MULAN-Modell bietet hier die Möglichkeit, für andere kognitive Modelle andere Agentennetze zu verwenden [siehe Röl04, Abschnitt 2.5, S. 28–44], für die dann die folgenden Aussagen über Wissen, Protokolle und Konversationen nicht mehr zutreffen. Letztendlich lassen sich aber alle kognitiven Modelle auch mit dem Standard-Agentennetz umsetzen, da dieses eine allgemeinste mögliche konzeptionelle Beschreibung von Agenten darstellt.

Der Zustand eines Agenten setzt sich aus zwei Teilen zusammen: dem Wissen, welches der Agent längerfristig explizit ansammelt und bewahrt, sowie den dynamisch anfallenden Informationen bei der Ausführung des Agenten (also der Markierung der Agentennetze). Das längerfristige Wissen wird von einem MULAN-Agenten in einer Wissensbasis abgelegt. Deren Modellierung lässt die Architektur mit Rücksicht auf die unterschiedlichen Bedürfnisse der Anwendungsgebiete offen.

Das Verhalten von Agenten wird in der MULAN-Architektur durch *Protokolle* modelliert. Protokolle sind wiederum Netze, die dem Agenten zur Verfügung stehen. Wenn der Agent irgendetwas tun oder denken möchte – sei es als Reaktion auf eine ankommende Nachricht, oder auch proaktiv, d.h. von sich aus – wählt er ein passendes Protokollnetz aus und erzeugt eine Netzinstanz davon. Ein Beispiel für ein instantiiertes Protokoll, in der MULAN-Terminologie Konversation genannt, ist in Abbildung 2.14 links unten zu sehen.

Mittels der Protokollnetze und der Wissensbasen der Agenten wird der dynamische Aspekt des Multiagentensystems modelliert. Zum einen beschreibt der Entwickler eines Multiagentensystems durch die Spezifikation von Protokollnetzen das Agentenverhalten und die von den Agenten zu versendenden und zu erwartenden Nachrichten. Zum anderen kann sich das Wissen der Agenten und damit auch die Menge der einem Agenten bekannten Protokolle zur Laufzeit ändern, so dass adaptive oder lernfähige Agenten modellierbar sind.

Die vier Ebenen der MULAN-Architektur – System, Plattform, Agent und Protokoll – beschreiben ein Multiagentensystem von der äußersten bis zur innersten Sichtwei-

se. Diese „sichtbar-in“-Beziehung wird durch das Netze-in-Netzen-Paradigma intuitiv wiedergegeben: Die Plattformen liegen als Marken im System, die Agenten wiederum liegen in der Plattform, und die Konversationen laufen im Agenten ab.

Die Kommunikation zwischen den jeweils benachbarten Ebenen wird durch synchrone Kanäle hergestellt, wobei auch eine „Weitervermittlung“ über mehrere Kanäle, z. B. vom Agenten über die Plattform zur Kommunikationsstruktur des Systems und zurück zu einem anderen Agenten auf einer anderen Plattform, möglich ist.

Die MULAN-Architektur ist durch und durch auf Nebenläufigkeit und Verteilung ausgelegt. Die Verteilung wird nur in der obersten Ebene, dem Systemnetz, explizit beschrieben, Nebenläufigkeit ist aber auf allen Ebenen vorhanden. Innerhalb einer Plattform können beliebig viele Agenten nebeneinander existieren und unabhängig voneinander arbeiten und kommunizieren. Auch innerhalb des Agenten ist das nebenläufige Führen mehrerer Konversationen möglich, auch die Initiierung von Konversationen kann nebenläufig geschehen. Synchronisation ist nur dann nötig, wenn innerhalb eines Agenten zwischen verschiedenen Handlungssträngen Informationen ausgetauscht werden müssen. Der Synchronisationspunkt ist dann die Wissensbasis des Agenten, die – je nach Aufbau – unterschiedlich feine Synchronisationsgrade beim Modifizieren und Abfragen von Informationen bieten kann.

Eine Weiterentwicklung von MULAN führt zu hierarchischen Multiagentensystemen. Das Innenleben eines MULAN-Agenten lässt sich wiederum als MAS betrachten, in dem alle Protokoll- und Wissensbasis-Netzinstanzen als Agenten gemeinsam das Verhalten der äußeren Agentenhülle bewirken. Die Verwaltungs- und Kommunikationsdienste, die ein Agent seinen Sub-Agenten bietet, ähneln denen der Plattform so stark, dass die beiden Konzepte Agent und Plattform zu einem Plattformagenten zusammengefasst werden können. Dieser Kniff führt zur Modellierung rekursiv geschachtelter Plattformen. Das hierarchische MULAN-Modell stellt dann ein holonisches MAS dar.

Da die MULAN-Referenzarchitektur mit Java-Referenznetzen (eingeführt in Unterabschnitt 2.1.3) modelliert ist, ist sie auch innerhalb des Werkzeugs RENEW ausführbar. MULAN kann daher zur Implementierung konkreter Multiagentensysteme eingesetzt werden. Eine alternative, FIPA-konforme Plattformimplementierung namens CAPA für MULAN hat der Autor in [Duv02, DMR03] vorgestellt. Beim Einsatz von CAPA ersetzt die Plattformimplementierung gleichzeitig das Systemnetz, da nun „echte“ Netzwerkkommunikation zwischen Plattformen stattfinden kann. In [Sch07] ist zudem die Anpassung von CAPA an das Konzept des hierarchischen MULAN beschrieben.

MULAN gibt als Referenzarchitektur klare Anhaltspunkte, welche Artefakte im Rahmen der agentenorientierten Softwareentwicklung zu erschaffen sind. Dabei handelt es sich gerade um die Wissensbasisinhalte und die Protokollnetze. Letztere beschreiben, wenn passende Protokollnetze verschiedener Agenten zusammengeführt werden, Interaktionsprotokolle. Die meisten Protokollnetze können genau einem Kreuzungspunkt aus Rolle und Interaktion in der Matrix, welche die MAS-Struktur übergreifend beschreibt, zugeordnet werden. Die Modellierung der MAS-Struktur auf Makro-Ebene wird somit von MULAN nicht eingeschränkt.

### 2.5.3 AOSE und CBSE

Fast alle existierenden Implementierungen von Plattformen für Multiagentensysteme setzen auf einer objektorientierten oder komponentenbasierten Grundlage auf, da die Funktionalität schlichtweg in einer existierenden Programmierumgebung stattfinden muss. Die Eigenschaften des zugrundeliegenden Paradigmas sind in den meisten Fällen nur für Programmierer von Multiagentensystemen wahrnehmbar, da sie keinen expliziten Einfluss auf die Modellierung der Systeme haben.

Einige Spezifikationen oder Rahmenwerke integrieren sich allerdings in ein existierendes Komponentenmodell und ergänzen es um agentenorientierte Konzepte. Beispielsweise beschreibt die ungefähr zeitgleich zu den FIPA-Spezifikationen entstandene „Multi-Agent System Interoperability Facilites Specification“ [MASIF, siehe OMG97] CORBA-Dienste zur Implementierung eines Rahmenwerkes für Multiagentensysteme. Das „Agent Deployment Model“ [ADM, siehe MCC+04] setzt auf dem „CORBA Component Model“ (CCM, siehe Unterabschnitt 2.2.2) auf. Im ADM wird zum einen ein Agent als Verbund einer Kernkomponente mit weiteren Komponenten für Pläne und zum anderen eine Organisation als Verbund einer Kernkomponente mit weiteren Agenten-Komponenten definiert. Beide CORBA-basierten Ansätze profitieren von der existierenden CORBA-Infrastruktur zur Verwaltung von Komponenten und zur Kommunikation über ein Netzwerk. Sie setzen daher eine gesonderte Kommunikationssprache wie die FIPA-ACL allenfalls in zweiter Linie ein. Allerdings lässt diese Vorgehensweise die in der Agentenkommunikation relevante Interaktionsprotokollsicht außen vor.

Gervais und Muscutariu beschreiben in [MGM02] ein UML-Profil namens „MASIF-DESIGN“, das im Wesentlichen auf den Deployment-Diagrammen der UML aufbaut. Das Profil dient als ADL für agentenbasierte Systeme und eignet sich insbesondere zur Beschreibung der Migrationswege von Agenten. Die technische Grundlage der beschriebenen Systeme orientiert sich an den MASIF-Spezifikationen.

Da ein Deployment-Diagramm lediglich einen Schnappschuss des Systemzustandes beschreibt, wird die Dynamik in der Organisation agentenbasierter Systeme nicht wirklich erfasst. Zudem erwähnen Gervais und Muscutariu in dem genannten Papier weder Ausdrucksmittel für ADL-typische Elemente wie z. B. Verbinder noch solche für agententypische Elemente wie z. B. Interaktionsmuster. Somit stehen dem Entwickler neben dem durch das Profil definierten Ortsstruktur lediglich die standardmäßig von der UML bereitgestellten Konzepte zur Modellierung von Komponentenschnittstellen und -hierarchien zur Verfügung.

Diverse Arbeiten betrachten die Beziehung zwischen CBSE und AOSE aus einer anderen Richtung, indem sie Ähnlichkeiten erkennen und die Einführung agentenorientierter Konzepte in die komponentenbasierte Softwareentwicklung vorschlagen. Griffel benennt beispielsweise in [Gri98, Kap. 6] folgende Zusammenhänge und Ähnlichkeiten zwischen Komponenten und Agenten. Beiden Konzepten ist gemein, dass zwei Sprachebenen in der Programmierung zum Einsatz kommen – eine zur Implementierung einzelner Komponenten bzw. Agenten und eine zur Komposition bzw. Koordination der Einzelteile [vgl. Gri98, S. 537]. Der Anpassbarkeit von Komponenten steht bei Agenten die Zuteilung konkreter Aufträge im Rahmen einer Anwendungsdomäne gegenüber [vgl. Gri98, S. 538f.]. Der Autonomie der Agenten entspricht die Eigenständigkeit von Komponenten, um in möglichst vielen Situationen einsetzbar zu

sein – wobei letztere gezielt eingeschränkt wird, da Komponenten zuverlässige und validierbare Dienstleister sein müssen [vgl. Gri98, S. 539f.]. Die Mobilität von Agenten wird in Komponentensystemen nicht unmittelbar benötigt, die damit einhergehenden Konzepte lassen sich aber einsetzen, um plattformunabhängige, fehlertolerante oder austauschbare Komponenten zu beschreiben [vgl. Gri98, S. 542]. Die komplexen Kommunikations- und Koordinationsfähigkeiten der Agenten können eingesetzt werden, um die Kopplung von Komponenten auf einem ausdrucksstarken Niveau zu beschreiben und Komponentenschnittstellen flexibler zu gestalten [vgl. Gri98, S. 575f.].

Griss stellt in [GP01, zusammen mit Pour] und [Gri01b] der CBSE-Gemeinde die Möglichkeit vor, die komponentenbasierte Softwareentwicklung in Richtung Agentenorientierung weiter zu entwickeln. Komponenten werden in der Regel als unabhängig einsetzbare Einheiten gesehen, die mit ihrer Umgebung durch wohldefinierte Schnittstellen interagieren, während sie ihre Implementierung einkapseln [vgl. GP01, S. 37]. Diese Definition ist nicht weit von den bekannten Agentendefinitionen (siehe Unterabschnitt 2.5.1.1) entfernt, so dass eine Komponente als Agent betrachtet werden kann, sobald sie eine Kombination der charakteristischen Eigenschaften von Agenten wie Adaptivität, Autonomie, Kollaborationsfähigkeit, Weisheit (*knowledgeability*), Mobilität oder Persistenz aufweist [vgl. Gri01b, S. 645f.]. Während die Agentenplattformimplementierung die Rolle der Komponentenmodellimplementierung bzw. Middleware übernimmt, kann die Komponenten-Infrastruktur (Definition siehe Unterabschnitt 2.2.1) durch spezielle Dienstagenten bereitgestellt werden. [vgl. Gri01b, S. 647]. Die Komplexität der Komponentenschnittstellen wird dabei reduziert und durch komplexe Nachrichtenstrukturen ersetzt. Der Vorteil der Verwendung einer ACL im Vergleich zu einer IDL besteht nach Griss darin, dass die semantische Bedeutung einer Schnittstelle nicht mehr nur dem Entwickler aus der Dokumentation, sondern auch den Agenten selber während der Kommunikation zugänglich ist. Ergänzungen an standardisierten Schnittstellen lassen sich schwieriger vornehmen als Erweiterungen der ACL-Nachrichtenstruktur [vgl. Gri01b, S. 648f.].

Insbesondere unter dem Schlagwort der üblicherweise komponentenbasiert implementierten WebServices (siehe Unterabschnitt 2.2.3) finden sich sowohl konzeptionelle als auch praktische Arbeiten, die Brücken zwischen komponentenbasierten und Multiagentensystemen schlagen. Sven Offermann hat einen Gateway-Agenten in Capa implementiert, der zwischen der WebService-typischen Kommunikation auf Basis von HTTP, SOAP etc. und der FIPA-ACL übersetzen kann. Der Ansatz wurde in [MOO04] und [MOO05] vorgestellt. Jan Ortmann hat in [Ort03, MO04] eine Abbildung zwischen textuellen Prozessbeschreibungssprachen und Petrinetzen definiert und später in den Capa-WebService-Gateway auf der Basis von Mulan-Protokollnetzen integriert. Einen ähnlichen Ansatz für die Agentenplattformimplementierung JADE präsentieren Nguyen et al. in [NKCG05] in Form eines WebService-Rahmenwerkes. Varga et al. stellen in [VHW05] ein Werkzeug vor, dass die *WebService Definition Language* WSDL in eine Agenten-taugliche Ontologie sowie Code-Gerüste für JADE-Agenten überführt.

Die Grenzen der AOSE zu anderen Softwareentwicklungsparadigmen sind keineswegs scharf, zumal auch in anderen Paradigmen verstärkt die Fragestellungen komplexer, verteilter und dynamischer Systeme angegangen werden. Beispielsweise werden auch in der Objekt- oder Komponentenorientierung aktive Objekte bzw. Komponenten

diskutiert. Die von Architekturbeschreibungssprachen und Komponentenmodellimplementierungen bereitgestellten Verbinder bzw. Interaktionsmuster werden zunehmend komplexer und offener [vgl. ZO04, S. 257]. Weiß und Jakob vertreten daher in [WJ04, S. 12f.] den Standpunkt, dass die agentenorientierte Sichtweise als Ergänzung zu den anderen Paradigmen zu verstehen ist. Wie eben erläutert, wird die AOSE speziell in der komponentenbasierten Softwareentwicklung (CBSE) von einigen Forschern als zukünftige Weiterentwicklungsrichtung erkannt.

Die AOSE verspricht, sich besonders für die Anforderungen komplexer Softwaresysteme zu eignen, da die abzubildenden realen Systeme inhärent verteilt sind und keine zentrale Steuerung besitzen [vgl. Jen00, S. 283f.]. Die AOSE verspricht ferner, sich gut zur Integration bereits existierender Softwaresysteme in ein heterogenes, verteiltes Gesamtsystem zu eignen, da auch hier wieder die Konzepte der Autonomie, Situiertheit und losen Koordination greifen [vgl. ZO04, S. 258f.]. Die mit diesen realen Systemeigenschaften einhergehende Unvorhersagbarkeit des vollständigen Systemverhaltens lässt sich durch Multiagentensysteme gut nachbilden. Die Unvorhersagbarkeit des Verhaltens eines agentenorientierten Softwaresystems ist allerdings auch ein vielzitierter Kritikpunkt der AOSE [vgl. Jen00, S. 289f.]. Dieses Problem lässt sich allerdings durch die Beschränkung einiger Agenteneigenschaften umgehen, wodurch natürlich die Distanz zum abgebildeten realen System wieder vergrößert wird.

## 2.6 Diskussion

Softwareentwicklungsparadigmen, Architekturbeschreibungssprachen und Entwurfsmuster zerlegen die Komplexität eines umfassenden Softwaresystems durch Dekomposition, Abstraktion, Organisation und Wiederverwendung in handhabbare Päckchen. Die in diesem Kapitel vorgestellten Ansätze orientieren sich dabei an verschiedenen architekturprägenden Konzepten. Auf Basis dieser Konzepte werden Systeme in den einzelnen Softwareentwicklungsparadigmen unterschiedlich betrachtet, namentlich als System aus Objekten (Netzexemplaren), als System aus Komponenten oder als Multiagentensystem. Daneben unterstützen Architekturbeschreibungssprachen eine Sicht auf die strukturelle Komposition eines Systems im Überblick, während Entwurfsmuster objektorientierte Mikroarchitekturen innerhalb des Systems identifizieren. Aus den architekturprägenden Konzepten der Ansätze ergeben sich jeweils die Namen und Eigenschaften der Dekompositions-, Abstraktions- und Wiederverwendungselemente sowie die Art und Weise und der Zeitpunkt der Integration der Elemente zu einem Gesamtsystem. Außerdem bestimmt der jeweilige Ansatz, ob der Schwerpunkt der Systembetrachtung auf Funktionen, Prozessen oder Daten liegt und welche Organisationsformen beschrieben werden können.

In den folgenden Absätzen werden die Eigenschaften der Softwareentwicklungsparadigmen und -techniken noch einmal zusammenfassend diskutiert und verglichen. Tabelle 2.3 stellt die Ergebnisse des Vergleichs im Überblick dar.

Die Modellierung und Implementierung von Systemen mit objektbasierten Referenznetzen folgt dem Softwareentwicklungsparadigma der objektorientierten Softwareentwicklung. Während die möglichen Beziehungen zwischen Objekten bereits beim Systementwurf von Typen, Klassen und Schnittstellen spezifiziert werden, findet das

| Ansatz<br>*s. Abschnitt* | Referenznetze<br>2.1 | CBSE<br>2.2 | ADL<br>2.3 | Entwurfsmuster<br>2.4 | AOSE<br>2.5 |
|---|---|---|---|---|---|
| Abstraktionskonzept | Objekte | Komponenten | Komponenten und Verbinder | Klassen und Beziehungen | Agentenrollen und Interaktionen |
| Integration | Spezifikation durch Modellierer (Instantiierung zur Laufzeit) | Montage durch „Ingenieur“ | Spezifikation durch Modellierer | Implementierung durch Entwickler | Selbstorganisation zur Laufzeit |
| Dekomposition | funktional, daten- oder prozessorientiert | funktional | funktional und datenorientiert | technisch-funktional | funktional und prozessorientiert |
| Organisation | beliebig; dynamisch | flach oder hierarchisch; statisch | hierarchisch; rekonfigurierbar | beliebig; statisch | beliebig; dynamisch |
| Verhaltensspezifikation | Teil der Petrinetze | nur in Schnittstellenverträgen | Attribut von Komponenten oder Verbindern | nur als Konvention | Interaktionen |

Tabelle 2.3: Gegenüberstellung der diskutierten Ansätze zur Softwarearchitektur

tatsächliche Knüpfen von Beziehungen erst dynamisch, also während der Laufzeit des Systems, statt. In einen Objekt können Daten und zugehörige Funktionalität gekapselt werden, so dass eine Dekomposition des Systems nach funktionalen[27] und datenorientierten Aspekten unterstützt wird. Mit Petrinetzen ist zudem in jeder Klasse (Netzmuster) eine visuelle und formale Verhaltensbeschreibung angelegt, auf Basis derer auch eine prozessorientierte Dekomposition erfolgen kann. Referenzen erlauben ein beliebig strukturiertes Geflecht an Beziehungen zwischen Objekten. Das Beziehungsgeflecht kann sich während der Laufzeit des Systems jederzeit dynamisch verändern.

Das komponentenbasierte Softwareentwicklungsparadigma (CBSE) zerlegt ein System nach funktionalen Aspekten in wiederverwendbare, anpassbare Komponenten. Ein „Ingenieur“ (*engineer*) montiert aus Komponenten ein System für den Endkunden – er bestimmt dabei die Organisation des Systems, so dass sie aus Sicht des Kunden statisch unveränderlich ist. Während in akademischen Modellen und auch in den Komponentendiagrammen der UML eine hierarchische Organisation von Komponenten durch Aggregation von Einzelkomponenten zu komplexen Komponenten möglich ist, bieten die in der Praxis verwendeten Komponentenmodellimplementierungen de facto nur eine flache Organisation aus gleichberechtigten Komponenten an. Dekomposition und Abstraktion beschreiben lediglich die Struktur des Systems. Eine umfassende Spezifikation des Verhaltens des Gesamtsystems oder von Teilsystemen ist kaum möglich, da Komponentenschnittstellen oftmals nur einen Teil des Verhaltens von Komponenten in Form von Vor- und Nachbedingungen oder Nutzungsverträgen spezifizieren, sofern solche Techniken überhaupt eingesetzt werden.

Auch den Architekturbeschreibungssprachen (ADL) liegt die Idee einer funktionalen Dekomposition zugrunde. Allerdings werden nicht nur Komponenten, sondern auch die sie verbindenden Elemente (Verbinder) als vollwertige Elemente betrachtet. Im Vordergrund steht die Modellierung oder Spezifikation der Architektur eines Softwaresystems. Manche ADLs können aufbauend auf dem Konzept der Rekonfigurierbarkeit auch gewisse dynamische Veränderungen der Architektur modellieren. Da die Beschreibung der (De-)Komposition eines Systems das zentrale Ziel von ADLs ist, erlauben sie natürlich die Beschreibung einer hierarchischen Organisation, so dass jede Hierarchieebene auch eine Abstraktionsebene ist. Das Verhalten eines Systems bzw. seiner Einzelteile lässt sich in Attributen der Komponenten und Verbinder spezifizieren, so dass insbesondere die Analyse bzw. Verifikation von Systemeeigenschaften unterstützt wird. Eine umfassende Modellierung von Dynamik und Prozessen im System scheint dennoch schwierig zu sein, da dieser Bereich seit längerem erforscht wird und die bisherigen Ergebnisse nicht für den praktischen Einsatz taugen.

Die Aufgabe von Entwurfsmustern ist, häufig wiederkehrende Beziehungen zwischen Klassen in objektorientierten Systemen abstrakt zu erkennen und zur Wiederverwendung aufzubereiten. Als Mikroarchitekturen betrachten Entwurfsmuster Klassen und deren Beziehungen, meist dienen sie der Flexibilisierung dieser Beziehungen. Verwendet werden Entwurfsmuster beim objektorientierten Entwurf bzw. während der Implementierung. Die durch ein Entwurfsmuster vorgegebene Dekomposition ist zwar funktional orientiert, findet aber meist auf einer technischen Ebene statt – es wird

[27] Das Adjektiv „*funktional*“ wird in weiten Teilen dieser Arbeit nicht im mathematischen Sinne, sondern in der Bedeutung „*funktionalitätsorientiert*“ verwendet.

keine anwendungsspezifische Funktionalität betrachtet. Gleiches gilt für die durch ein Entwurfsmuster beschriebene Organisation. Die Beziehungen zwischen den beteiligten Klassen können zwar beliebig organisiert sein, aber sie sind, da sie im konkreten Entwurf bzw. Code des Systems realisiert werden, nicht mehr variabel. Zur Definition eines Entwurfsmusters gehört eine Beschreibung der gegenseitigen Nutzung der beteiligten Klassen in Form von Interaktionen. Diese Beschreibung wird aber lediglich dem Softwareentwickler als Konvention auferlegt, die beteiligten Objekte können diese Ebene nicht wahrnehmen.

Im Paradigma der agentenorientierten Softwareentwicklung (AOSE) werden auf mehreren Ebenen Metaphern definiert, welche das Erreichen von Eigenschaften wie Adaptivität oder Flexibilität des Gesamtsystems erleichtern. Zwar beeinflussen die unterschiedlichen Techniken zur Implementierung von Adaptivität oder Autonomie individueller Agenten auf der Mikro-Ebene die übergreifende Softwarearchitektur nur geringfügig. Die Betrachtung von Interaktionsprotokollen in einem Multiagentensystem bietet aber ebenso wie die Definition von Organisationen und Rollen auf der Makro-Ebene Begriffe, welche in der objektorientierten oder komponentenbasierten Softwareentwicklung keine unmittelbare, gleichermaßen flexible Entsprechung haben. Rollen beschreiben Verantwortlichkeiten und Zuständigkeiten in einem System, welche von wechselnden Agenten eingenommen werden können. Interaktionen als gleichberechtigte Architekturelemente neben Rollen ergänzen die funktionale um eine prozessorientierte Dekompositionsrichtung. Beide Dekompositionsrichtungen sind eng verzahnt, da Interaktionsmuster zum einen die funktionalen Rollen der Beteiligten festlegen und zum anderen dem Zweck der Organisation des Multiagentensystems dienen. Die Spezifikation hierarchischer Multiagentensysteme – Agenten in Agenten – ist in der AOSE allerdings noch nicht so weit gediehen wie in den anderen betrachteten Softwareentwicklungsparadigmen. Stattdessen können auf der Makro-Ebene Gruppen oder Organisationen hierarchisch organisiert werden, so dass die Agenten (ohne eigene Hierarchie) Positionen in der Organisationshierarchie einnehmen können.

In einem reinen, offenen, aus der Mikro-Ebene heraus modellierten Multiagentensystem benötigt man keine Möglichkeit, die Systemorganisation im Vorwege durch Modellierer oder Ingenieure festzulegen, da sich die Agenten während der Laufzeit des Systems selbständig organisieren, dabei beliebige Organisationsstrukturen aufbauen und diese Strukturen jederzeit dynamisch anpassen können. Die Eigenschaften der Autonomie von Agenten bzw. der Selbstorganisation von Multiagentensystemen führen allerdings bei einer Anwendung im Sinne der reinen Lehre der (V)KI zu einer stark eingeschränkten Vorhersagbarkeit des Systemverhaltens. Um für den praktischen Einsatz in der Softwareentwicklung ein deterministisches Systemverhalten (soweit im Rahmen der Anwendungsdomäne wünschenswert und erreichbar) zu forcieren, wird in der AOSE meist eine feste Systemorganisation sowie ein wohldefinierter Satz an Rollen und Interaktionsmustern vorgegeben, die nur begrenzte Flexibilität zulassen.

In Tabelle 2.3 nicht aufgeführt ist der Ursprung oder Hintergrund der Ansätze. Der Hintergrund ist zwar eine recht unscharfe Eigenschaft, er verdeutlicht aber in der Regel die mit der Wahl des architekturprägenden Konzepts verbundenen Intentionen.

Referenznetze verbinden einen Petrinetzformalismus mit dem Paradigma der objektorientierten Softwareentwicklung. Dem objektorientierten Paradigma liegt die Idee der gemeinsamen Kapselung von Daten und Funktionalität zugrunde, welche die Wart-

barkeit und Anpassbarkeit eines Programms verbessern soll. Die objektorientierte Perspektive ist daher tendenziell kleinteilig angelegt. Petrinetze dienen hingegen ursprünglich der formalen und grafischen Spezifikation von Zuständen, Aktivitäten und Prozessen in einem nebenläufigen System. Erst durch die Einführung objektbasierter Konzepte – wie im Paradigma der *Netze-in-Netzen* – entstehen Mittel zur Modellierung oder Implementierung von Softwarearchitekturen. Diese ergänzen allerdings im Unterschied zu imperativen oder funktionalen objektorientierten Sprachen intuitive und formal fundierte Konzepte zum Umgang mit Prozessen inklusive Nebenläufigkeit und Synchronisation.

Die komponentenbasierte Softwareentwicklung ist offensichtlich stark aus dem praktischen Einsatz in der industriellen Fertigung von Software heraus motiviert, wie die konkurrierenden, in der Praxis verwendeten, dafür aber recht einfach gehaltenen Komponentenmodelle belegen. Die Nähe der CBSE zur etablierten objektorientierten Softwareentwicklung ist unverkennbar, so dass zwar für Entwickler beim Paradigmenwechsel nur eine flache Lernkurve zu bewältigen ist, aber auch der Zugewinn an Abstraktionsmöglichkeiten gering ausfällt.

Die Architekturbeschreibungssprachen sehen hingegen ihren Schwerpunkt in der Spezifikation und Analyse ganzer Systeme mit Struktur und Verhalten. Durch den hohen, theoretisch ausgerichteten Ansatz wird zum einen die Ausdrucksmächtigkeit existierender Sprachen den Ansprüchen des eigenen Paradigmas nicht gerecht, zum anderen ist die Erzeugung praktisch einsetzbarer Software aus Architekturbeschreibungen heraus noch nicht ausgereift.

Entwurfsmuster befassen sich mit wiederverwendbaren Mikroarchitekturen, welche Struktur und Verhalten eines kleinen Systemausschnitts mittels einer Lösungsschablone für wiederkehrende Problemstellungen prägen. Da Entwurfsmuster aus der rauhen Praxis der objektorientierten Softwareentwicklung heraus entstehen und mit dem konfliktträchtigen Anspruch möglichst allgemein verwendbarer, aber trotzdem hilfreich konkreter Lösungsschablonen entwickelt werden, ist ihnen eine gewisse Unschärfe der Geltungsbereiche der einzelnen Entwurfsmuster zu eigen. Dies führt zur häufigen Definition ähnlicher oder sich überlappender Muster. Der Vorteil des flexiblen Einsatzes wird durch einen Mangel an Präzision bei der Definition einer Softwarearchitektur erkauft.

Die agentenorientierte Softwareentwicklung führt im Vergleich zu den anderen beiden Paradigmen sehr viele aus dem Bereich der (verteilten) künstlichen Intelligenz entlehnte neue Konzepte und Begriffe ein, so dass die Wiederverwendbarkeit von Konzepten und Techniken aus etablierten Softwareentwicklungsparadigmen nicht sofort sichtbar wird. Lockert man den Umgang mit den neuen Begriffen aus der (V)KI, so dass sie als flexiblere Varianten etablierter Konzepte erscheinen, so erhält man im Wesentlichen zusätzliche Abstraktionsebenen für Prozesse und Organisationen, deren Umwandlung in praktisch einsetzbare Software allerdings noch in Arbeit ist.

Ein zentraler Begriff dieses Kapitels ist der Begriff der Softwarearchitektur, dessen Definitionen in Unterabschnitt 2.3.1 besprochen wurden. Die in den Abschnitten dieses Kapitels vorgestellten Softwareentwicklungsparadigmen stehen in unterschiedlicher Nähe und Ausrichtung zu dem Konzept einer Softwarearchitektur. Die Modellierung von Objekten oder Klassen und ihren Beziehungen in einer komplexen Anwendung ist in der Regel zu kleinteilig, um als „übergreifende Struktur“ gesehen zu werden.

Gleiches gilt daher für die Modellierung mit dem Paradigma der *Netze-in-Netzen*. Allerdings lassen sich die Konzepte der Klassen oder Netzmuster auch auf einer abstrakteren Ebene einsetzen, so dass die Modellierung von Softwarearchitekturen mit diesen Konzepten möglich ist. Die Betrachtung eines Softwaresystems als Komposition aus Komponenten – wie in der CBSE dominant – entspricht ziemlich genau dem Softwarearchitektur-Begriff, wie er auch Architekturbeschreibungssprachen zu Grunde liegt. In der agentenorientierten Softwareentwicklung kann von einer systemumfassenden Softwarearchitektur eigentlich nur auf der Makro-Ebene gesprochen werden. Eine kleinere Fassung des Systembegriffs auf der Agentenebene erlaubt zwar auch die Betrachtung der inneren Architektur eines Agenten auf der Mikro-Ebene als Softwarearchitektur. Diese Verwendung bezieht sich dann allerdings nicht auf die übergreifende Architektur der gesamten Anwendung als Multiagentensystem.

Wie die fünf in diesem Kapitel besprochenen Ansätze sich für die konzeptionelle Beschreibung von Plugin-Systemen eignen, wird nach der Untersuchung existierender Plugin-Systeme zu klären sein. Im Anschluss an den zusammenfassenden Vergleich existierender Plugin-Systeme werden daher in Unterabschnitt 3.4.3 Ähnlichkeiten und Unterschiede herausgearbeitet. Die verschiedenen architekturprägenden Konzepte – Objekt, Komponente, Agent – werden in Unterabschnitt 4.1.9.1 mit dem dann definierten Konzept des Plugins verglichen werden. In Unterabschnitt 4.5.3 wird das modellierte Plugin-Konzept als Softwareentwicklungstechnik in die hier diskutierten Ansätze eingeordnet werden.

# 3 Plugin-Systeme

Der Begriff des Plugins (*plug-in*) ist seit ca. 20 Jahren im Zusammenhang mit Software in Verwendung. Allerdings gab es auch davor schon Software- und Hardware-Systeme, deren Funktionalität durch externe Module erweitert werden konnte, die über eine Schnittstelle in das System eingesteckt (*to plug in*) werden konnten. Alternativ zur Bezeichnung Plugin treten auch die Bezeichnungen Erweiterung (*extension*) oder Ergänzung (*add-on/ add-in*) auf. Versuche, die Begriffe anhand von Eigenschaften wie der Eigenständigkeit des Moduls [z. B. Wik07b], der Beschränkung der Erweiterungsmöglichkeiten [z. B. Wik07c] oder der Trennung nach Funktionalitätsbereichen [z. B. MDC] zu unterscheiden, lassen keine einheitliche Richtung erkennen, zumal oft einige der Begriffe gleichgesetzt werden [z. B. GB03, S. 5]. Die bereits in Unterabschnitt 2.4.3 vorgestellten Plugin-Entwurfsmuster betonen wiederum andere Eigenschaften, z. B. die zeitliche Verlagerung der Systemkonfiguration von der Entwicklung zum Einsatz, das Auslagern spezieller Fähigkeiten einer Anwendung oder das Zusammenspiel mehrerer Architekturelemente in einem Plugin-System. Daher sollen in diesem Kapitel – bis eine exakte Definition des Plugin-Begriffs in Unterabschnitt 4.1.9 erfolgt – alle genannten Begriffe (*plug-in, add-on, add-in, extension*) einheitlich als Plugin verstanden werden.

Um eine Überblick über die Vielfalt der Plugin-Systeme zu bekommen, werden in diesem Kapitel einige prominente, aktuell weit verbreitete Softwaresysteme vorgestellt. Die Auswahl der untersuchten Systeme orientiert sich im Vorgriff auf die spätere Definition des Plugin-Begriffs bereits an einer essentiellen Eigenschaft von Plugin-Systemen. Entscheidend ist, dass die untersuchten Systeme technisch und fachlich darauf ausgerichtet sind, in einem oder mehreren Bereichen auf einfache Weise während ihres Einsatzes beim Anwender um zusätzliche Funktionalität erweitert werden zu können. Da die Zugänglichkeit des Quellcodes und der Dokumentation des zu untersuchenden Softwaresystems eine notwendige Eigenschaft für eine erfolgreiche Untersuchung ist, findet sich in der Auswahl ausschließlich Open-Source-Software. Das Hauptaugenmerk dieses Kapitels liegt auf zwei prominenten Anwendungen, die bereits jeweils mehrere Plugin-Systeme in sich vereinen:

- Die Netscape-Plugin-Programmierschnittstelle (*Netscape Plugin API, NPAPI*) der Mozilla-Anwendungssuite, welche die Integration von multimedialen Inhalten in die Präsentation von Webbrowsern unterstützt, steht stellvertretend für die vielen Plugin-Systeme mit eng begrenzter Schnittstellenfunktionalität. Eine Besonderheit ist, dass sich die NPAPI zu einem de-facto-Standard entwickelt hat, da die Schnittstelle von mehreren Softwaresystemen konkurrierender Hersteller bereitgestellt wird.

  Die *Mozilla*-Suite bietet gleichzeitig mit den *Extensions* eine extrem flexible Möglichkeit zur Integration zusätzlicher Funktionalität an. Die Softwarearchitektur umfasst mehrere Schichten, auf denen unterschiedliche Modularisierungs-

techniken eingesetzt werden. Dabei bestehen keine Unterschiede zwischen den technischen Möglichkeiten von Extensions und Kernanwendungen.

- Die *Eclipse*-Plattform basiert auf einer Softwarearchitektur, die von Grund auf als Plugin-System konzipiert wurde, so dass sämtliche Funktionalität von Plugins bereitgestellt wird. Die zugrunde liegende Plugin-Verwaltung wurde zudem vor ein paar Jahren ausgewechselt, so dass neben den klassischen Eclipse-Plugins auch Bundles entsprechend der Architektur der *Open Service Gateway Initiative (OSGi)* eingesetzt werden können.

Ergänzt wird die Betrachtung dieser beiden Anwendungen durch die Vorstellung weiterer Softwaresysteme, welche interessante Ideen zur Definition von Konzepten oder zur Lösung von Problemen im Bereich der Plugin-Systeme einbringen. Da der Übergang zwischen reinen Plugin-Systemen und anderen flexiblen, erweiterbaren oder dynamischen Softwarearchitekturen ein fließender ist, werden einige spezielle Fragestellungen, die Plugin-Systeme mit benachbarten Forschungsgebieten gemeinsam haben, aufgegriffen – soweit sie nicht bereits in Kapitel 2 Gegenstand der Diskussion waren.

Bei der Betrachtung der Plugin-Systeme ist vor allem die Softwarearchitektur von Interesse – welche Schnittstellen werden definiert, wie werden Plugins gekapselt und dann in das System integriert, und welches Systemverhalten wird dadurch beschrieben. Daneben ist die Abhängigkeit oder Abstraktion von Betriebssystem- oder anderen Plattformspezifika von Interesse, da sie die Wiederverwendbarkeit der Plugins betrifft.

Ein immer wieder bei der Untersuchung von Plugin-Systemen benötigter Begriff ist der des Lebenszyklus (*life-cycle*): Er bezeichnet die Zustände, in denen sich ein Plugin zwischen Installation, Ausführung und Deinstallation befinden kann, sowie die zwischen diesen Zuständen möglichen Übergänge. Im Gegensatz zu klassischen, monolithischen Softwaresystemen, die einmal installiert und anschließend bis zu ihrem Nutzungsende nur noch gewartet werden, wechseln Plugins ihren Lebenszyklus-Zustand deutlich häufiger. Daher wird dem Lebenszyklus in den folgenden Untersuchungen eine vergleichsweise hohe Aufmerksamkeit zuteil.

In den folgenden beiden Abschnitten werden die erwähnten zwei Softwaresysteme (die Netscape/Mozilla-Anwendungsfamilie sowie Eclipse) detailliert vorgestellt, wobei die Begrifflichkeiten der jeweiligen Softwarearchitektur übernommen werden.[1] Im Abschnitt 3.3 werden weitere, teilweise deutlich weniger prominente Plugin-Systeme vorgestellt, die im Vergleich zu den beiden zuvor diskutierten Anwendungen mit schärferen Konzeptdefinitionen oder interessanten Ideen zur Umsetzung von Plugin-Architekturen aufwarten. Ergänzt wird diese Sammlung durch eine kurze Benennung spezieller, im Rahmen der Entwicklung dynamisch erweiterbarer Systeme auftauchender Fragestellungen, die ebenfalls für die Entwicklung von Plugin-Systemen relevant sind.

[1] Leser, die sich in einem der vorgestellten Softwaresysteme bereits gut auskennen, können die detaillierte Vorstellung des Systems überspringen und direkt zur jeweiligen Zusammenfassung (Unterabschnitte 3.1.3, S. 134 bzw. 3.2.2, S. 155) vorblättern. Es empfiehlt sich allerdings, dennoch einen Blick auf die in die Vorstellung der Softwaresysteme eingearbeiteten Petrinetz-Darstellungen der Plugin-Lebenszyklen (Netscape-Plugins in Abbildung 3.2, S. 119, Mozilla-Extensions in den Abbildungen 3.3, S. 125, 3.4, S. 128 und 3.6, S. 133, klassische Eclipse-Plugins in Abbildung 3.7, S. 141, OSGi-Bundles in Abbildung 3.10, S. 145 und OSGi-Services in den Abbildungen 3.11, S. 147 und 3.13, S. 150 zu werfen, da diese in Unterabschnitt 4.4.2.2 miteinander verglichen werden.

Abschnitt 3.4 setzt die in den vorangegangenen Abschnitten vorgestellten Softwarearchitekturen zueinander in Beziehung. Dabei werden zunächst Gemeinsamkeiten und Unterschiede der Plugin-Systeme erarbeitet und mit den Eigenschaften der im vorigen Kapitel untersuchten Softwareentwicklungsparadigmen und -techniken verglichen.

## 3.1 Netscape/Mozilla

Die ursprünglich von der Firma Netscape produzierte Webbrowser-Software namens „Netscape Navigator“ ist ein Paradebeispiel für eine Anwendung, die im Laufe der Zeit immer mehr Funktionalität angesammelt hat und dabei mit mehreren Plugin-Schnittstellen versehen wurde. Die ursprüngliche Funktionalität der Darstellung statischer Hypertextdokumente im HTML-Format mit eingebetteten Bildern wurde zum einen um die Darstellung dynamischer und multimedialer Inhalte mittels neuer Internet-Technologien erweitert. Zum anderen wurden Programmteile zur Integration populärer Internet-Protokolle wie E-Mail oder Newsgroups ergänzt. Aus der einzelnen Anwendung von 1994 entstand zunächst eine Anwendungssuite namens „Netscape Communicator“. 1998 wurde der Quellcode der Anwendung an ein Open-Source-Projekt übergeben, das den ehemaligen Arbeitstitel „Mozilla“ zum offiziellen Namen des Projekts erkor. Das Mozilla-Projekt vertreibt seit 2005 anstelle einer komplexen Anwendungssuite getrennte Anwendungen für die Darstellung des World Wide Web und die Bearbeitung von E-Mail. Die Einzelanwendungen teilen sich aber weiterhin ein gemeinsames komponentenbasiertes Rahmenwerk.

Im Rahmen dieser Arbeit soll zunächst eine seit 1996 bestehende, stark zweckorientierte Schnittstelle namens *Netscape Plugin Application Programming Interface (NPAPI)* betrachtet werden. Ab der Version 2.0 des Netscape Navigators bestand mittels der NPAPI die Möglichkeit, beliebige andere Inhalte als bebilderten Hypertext direkt im Webbrowser zu präsentieren, sofern ein geeignetes *Plugin* installiert ist. Da die NPAPI inzwischen nicht nur von der Netscape- oder Mozilla-Anwendungssuite, sondern auch von diversen Webbrowsern anderer Hersteller verwendet wird, hat sie sich zu einem De-Facto-Standard für die Integration von multimedialen Inhalten in das World Wide Web entwickelt.

Neben der auf die spezielle Funktionalität der Präsentation von Inhalten festgelegten Netscape-Plugin-Schnittstelle bietet die Mozilla-Anwendungssuite seit der Einführung mehrerer modularisierender Architekturschichten die Möglichkeit, auf einem Rahmenwerk aufbauend weitere Funktionalität zu implementieren. Die Funktionalität kann wahlweise in eigenen Anwendungsfenstern bereitgestellt oder nahtlos in die Benutzungsoberfläche der Mozilla-Anwendung integriert werden. Für solche Funktionalitätserweiterungen hat sich seit 2004 die Bezeichnung „*Extension*“ etabliert.

Der folgende Unterabschnitt stellt die NPAPI vor. Zunächst werden die Definition, Eigenschaften und Bestandteile eines Netscape-Plugins aus der NPAPI-Spezifikation herausgearbeitet. Anschließend werden der Lebenszyklus und die Installation von Plugins genauer untersucht, da gerade dieser Bereich für die konzeptionelle Aufarbeitung von Plugin-Systemen relevant ist. Der Unterabschnitt 3.1.2 ist den Mozilla-Extensions gewidmet. Auf einen groben Überblick über die in Mozilla-Anwendungen und -Extensions enthaltenen Architekturschichten folgt eine genauere Betrachtung der

Mechanismen und Schnittstellen der einzelnen Schichten. Die wesentlichen Eigenschaften von Plugins und Extensions werden in Unterabschnitt 3.1.3 zusammengefasst.

## 3.1.1 Plugins

Vor der Einführung von Plugins griff der Netscape Navigator zur Präsentation von Inhalten, die nicht Text oder einfache Bilder waren, auf so genannte Hilfsanwendungen (*helper applications*) zurück. Hilfsanwendungen sind eigenständige Anwendungen – externe Programme, die beim Aufruf eine heruntergeladene Datei mitgegeben bekommen, und sich weder mit ihrer Darstellung noch ihren Interaktionsmöglichkeiten in den Webbrowser integrieren. Nach dem Start einer Hilfsanwendung findet keinerlei Informationsfluss mehr zwischen Webbrowser und Hilfsanwendung statt, so dass der Anwender mehrere Anwendungen zu bedienen hat.

Die *Netscape-Plugin-Programmierschnittstelle* (*Netscape Plugin Application Programming Interface, NPAPI*) ermöglicht im Gegensatz dazu die direkte Präsentation multimedialer Inhalte innerhalb des Webbrowsers. Sofern eine von Drittanbietern bereitgestellte, auf die Präsentation der jeweiligen Inhalte spezialisierte Softwarebibliothek, *Plugin* genannt, auf dem System des Anwenders installiert ist, kann der Webbrowser über die in der NPAPI deklarierten Funktionen mit der Plugin-Software direkt kommunizieren. Dem Anwender werden die multimedialen Daten so präsentiert und zur Interaktion angeboten als wäre dies Teil der Funktionalität des Webbrowsers. Prominente Beispiele für durch Plugins integrierbare Inhalte sind komplexe Dokumente im Portable Document Format (PDF), animierte interaktive Inhalte im Flash-Format, Audio- und Videodaten in diversen Formaten sowie beliebige Programme in diversen Programmiersprachen wie z.B. Java. [Vgl. Wik07a]

### 3.1.1.1 Definition und Eigenschaften

Der Begriff des Plugins wird im 1998 mit der Version 4.0 des Navigators veröffentlichten „Plug-in Guide“ wie folgt definiert:

> „A plug-in is a separate code module that behaves as though it is part of the Netscape Communicator browser. You can use the Plug-in API to create plug-ins that extend Communicator with a wide range of interactive and multimedia capabilities, and that handle one or more data (MIME) types.“ [Net98, Plug-in Basics]

In dieser Definition wird ein Plugin zunächst auf seinen technischen Aspekt reduziert, als extern bereitgestellte Softwarebibliothek ein Teil der Anwendung sein zu können. Erst in der Erläuterung der Verwendungsmöglichkeiten geht der Text darauf ein, dass die Funktionalität des Webbrowsers erweitert werden kann, indem Fähigkeiten zur Verarbeitung von Inhalten mit weiteren, dem Webbrowser sonst unbekannten Datentypen ergänzt werden.

Die von der Plugin-Schnittstelle angebotenen Möglichkeiten zur Präsentation und Integration beliebiger Inhalte in die Darstellung des World Wide Web werden im genannten „Plug-in Guide“ wie folgt zusammengefasst:

„With the Plug-in API, you can create dynamically loaded plug-ins that can:

- register one or more MIME types
- draw into a part of a Communicator window
- receive keyboard and mouse events
- obtain data from the network using URLs
- post data to URLs
- add hyperlinks or hotspots to link to new URLs
- draw into sections on an HTML page“

[Net98, Plugin-in Basics: How You Can Use Plug-ins]

Nach dieser Liste lässt sich sowohl die visuelle Darstellung im Webbrowser integrieren, als auch die interaktive Steuerung über den Webbrowser an das Plugin weiterreichen. Zudem dürfen Plugins über die Mechanismen des Webbrowsers Daten aus dem Internet empfangen bzw. dorthin versenden, die Navigationsmöglichkeiten des Anwenders durch das World Wide Web erweitern sowie Einfluss auf den in der Umgebung des multimedialen Inhaltes dargestellten Hypertext nehmen. Zusammengenommen ergeben die genannten Fähigkeiten die Möglichkeit, einen mit dem passenden MIME-Typ versehenen multimedialen Inhalt durch das Plugin direkt im Webbrowser zu präsentieren, so dass dem Anwender das Zusammenspiel mehrerer Programme bei seiner Navigation durch das World Wide Web nicht bewusst wird. Somit stellt die NPAPI eine auf den Einsatzzweck der Webbrowser-integrierten Präsentation multimedialer Inhalte spezialisierte Schnittstelle dar.

Diese Liste an Fähigkeiten wurde seitdem nicht verändert, sie ist in der aktuellen Version der „Plugin API Reference“ lediglich um eine Fähigkeit vervollständigt worden, die eigentlich auch 1998 schon in ähnlicher Form zur Verfügung stand:

- „communicate with JavaScript/DOM from native code“

[MDC07f, Plugin-in Basics: How Plug-ins are Used]

Hierbei geht es um eine weitergehende Integration der durch Plugins dargestellten Inhalte in dynamisch veränderliche Webseiten, die mittels Skriptsprachen auf Interaktionen des Anwenders oder Anweisungen eines Webservers reagieren können. Die hierzu erforderliche Erweiterung der Plugin-Schnittstelle wurde im Laufe der Jahre über verschiedene Umwege (z. B. auf Basis der Programmiersprache Java, des Microsoft-Komponentenmodells COM oder auch des in Unterabschnitt 3.1.2.1 behandelten Mozilla-Komponentenmodells XPCOM) versucht. Die aktuelle, im Zusammenspiel mehrerer großer Firmen entwickelte und seit einigen Jahren im Einsatz befindliche Technik trägt den Namen „NPRuntime“ [Ste04] und erweitert die NPAPI, so dass Plugins Objekte in der Skriptsprache bereitstellen und auch selber Skripte ausführen lassen können.

#### 3.1.1.2 Dynamische Bibliotheken

Ein Plugin besteht aus einer dynamischen Softwarebibliothek (*Dynamic Library*), die prozessorarchitekturabhängigen Maschinencode enthält. Um den Maschinencode mehrerer Bibliotheken dynamisch in den Maschinencode von Anwendungen integrieren zu

können, geben die Prozessorarchitektur, das Betriebssystem und der verwendete Compiler eine Binärschnittstelle (*Application Binary Interface, ABI*) vor, die vom Code jeder Bibliothek eingehalten werden muss. Die NPAPI ist in der Programmiersprache C spezifiziert und implementiert. Diese Sprache ermöglicht zwar die umfangreiche Wiederverwendung von Quellcode auf mehreren Betriebssystemen und Prozessorarchitekturen (Portabilität), aber es muss jeweils ein spezifisches Kompilat erzeugt werden, das der Prozessorarchitektur und der Binärschnittstelle entspricht.[2] [Vgl. MDC07f, Plug-in Basics: Plug-ins and Platform Independence]

Das auf allen gängigen Betriebssystemen angebotene Konzept der dynamischen Bibliothek stellt die Basis für die Implementierung des Netscape-Plugin-Systems dar. Damit übernimmt das Betriebssystem die grundlegende Aufgabe, zur Laufzeit Code aus verschiedenen Quellen (eben Bibliotheken) in die Anwendung zu integrieren. Obwohl alle Betriebssysteme ein ähnliches Grundkonzept verwenden, unterscheiden sich die Dateiformate und Integrationstechniken der dynamischen Bibliotheken mehr oder weniger stark. Beispielsweise wird unter der Betriebssystemfamilie „Microsoft Windows“ eine Datei im Format einer Dynamic Link Library (DLL), unter Betriebssystemen der „Unix“-Familie im Format eines (Dynamic) Shared Object (SO oder DSO), sowie unter der Betriebssystemreihe „Mac OS“ im Format eines Mach-O-Bundles benötigt.

Zur Integration einer dynamischen Bibliothek in eine Anwendung übernimmt das Betriebssystem zwei Aufgaben: das Laden (*loading*) und das Verknüpfen (*linking*). Einen Überblick über verschiedenen Techniken des Ladens und Verknüpfens sowie die Entwicklung von statisch verknüpften Programmen zu dynamischen Bibliotheken gibt Levine in [Lev00].

Beim *Laden* wird der Inhalt der dynamischen Bibliothek in den Speicher des Systems geholt, um dort als Code ausgeführt werden zu können. Bei diesem Vorgang werden Anpassungen am Code vorgenommen (z. B. Korrekturen von Sprungadressen), falls der Code von seiner Position im Speicher abhängig ist. Je nach Technik kann der Inhalt einer Bibliothek nur einer Anwendung oder auch mehreren zur Verfügung gestellt werden. Zudem ist es möglich, andere Inhalte als Maschinencode wie z. B. Texte oder Grafiken in einer dynamischen Bibliothek unterzubringen – insbesondere unter den Betriebssystemfamilien „Microsoft Windows“ und „Mac OS“ wird auf solche Ressourcen zurückgegriffen, um Anwendungen in die grafische Benutzungsoberfläche zu integrieren.

Um den Maschinencode einer Bibliothek mit dem einer Anwendung verknüpfen zu können, werden in der Regel gemeinsame Bezeichner für Variablen und Funktionen vereinbart. Diese werden im Quellcode der Bibliothek explizit als zu *exportierende* Symbole und im Quellcode der Anwendung als zu *importierende* Symbole deklariert. Der Maschinencode der Anwendung enthält eine Tabelle mit den genutzten Bezeich-

[2] Die Vorgabe einer ABI durch Prozessorarchitektur, Betriebssystem und Compiler führt dazu, dass die Änderung nur eines der drei Beteiligten eine Inkompatibilität der Kompilate zur Folge haben kann. Dies wird z. B. bei der im Gange befindlichen Umstellung der Registerbreite der x86-Prozessorarchitektur von 32-Bit auf 64-Bit deutlich. Selbst wenn auf Prozessor-, Betriebssystem- und Compilerebene jeweils Mechanismen zum Umgang mit beiden Registerbreiten vorgesehen sind, lassen sich 32-Bit-Plugins nicht ohne Adapter in 64-Bit-Webbrowsern ausführen [vgl. Lee08]. Mitunter erzeugen verschiedene C-Compiler (oder gar verschiedene Versionen desselben Compilers, wie z. B. einige Versionen des `gcc`) unter demselben Betriebssystem auf derselben Prozessorarchitektur Kompilate mit abweichenden Binärschnittstellen.

nern und verwendet deren Adressen als Platzhalter für den Code der Bibliothek. Beim *Verknüpfen* ersetzt das Betriebssystem die Platzhalter durch Verweise auf den tatsächlichen Code der Bibliothek.

Das dynamische Laden und Verknüpfen von Anwendungs- und Bibliothekscode erlaubt das separate Entwickeln, Bereitstellen und Austauschen der dynamischen Bibliotheken. Es erfordert allerdings eine im Vorwege bekannte, feste Menge von Bezeichnern mit fest vereinbarter Semantik. Je nach Technik wird dabei ein globaler Namensraum für alle innerhalb einer Anwendung verwendeten Bibliotheken oder ein eigener Namensraum je Bibliothek verwendet [vgl. Lev00, Abschnitt 10.10, S. 225f.]. Zudem muss die Anwendung jede einzubindende dynamische Bibliothek benennen können, was bei beliebig erweiterbaren Plugin-Systemen nicht möglich ist. Zwar bieten die meisten Betriebssysteme einen Suchpfad-Parameter oder Standardverzeichnisse, in denen nach Bibliotheken gesucht wird, aber der Name der Bibliothek muss von der Anwendung explizit genannt werden.

Bei Plugin-Systemen sind aber die individuellen Plugins zur Übersetzungszeit nicht bekannt. Diese Eigenschaft unterscheidet Plugin-Systeme wesentlich von konventionellen modularisierten Anwendungen, welche auf dem Mechanismus dynamischer Bibliotheken aufbauen [vgl. MMS03, S. 88]. Daher muss die NPAPI – ebenso wie alle anderen Plugin-Systeme, die auf dynamischen Bibliotheken aufsetzen – einen eigenen Mechanismus zum Auffinden der Plugins definieren, dieser wird im Rahmen der Installation von Plugins im Unterabschnitt 3.1.1.5 erläutert. Außerdem muss die Vereinbarung der Bezeichner (also die Programmierschnittstelle) gleichzeitig konkret und flexibel genug sein, um dem Einsatzzweck des Plugin-Systems zu genügen.

#### 3.1.1.3 Programmierschnittstelle

Die dynamische Bibliothek des Plugins muss zum einen ein Mindestmaß der in der NPAPI spezifizierten Funktionen implementieren, zum anderen eine Beschreibung der Eigenschaften des Plugins bereitstellen. Zu den Eigenschaften gehört insbesondere die Liste der MIME-Bezeichner der vom jeweiligen Plugin behandelten Datentypen, anhand derer der Webbrowser später die Zuständigkeit des Plugins für einen im World Wide Web angetroffenen Inhalt ermitteln kann. Die Eigenschaften werden ebenfalls mit betriebssystemspezifischen Mechanismen bereitgestellt. Während die Bibliotheksformate unter Windows und Mac OS jeweils einen speziellen Bereich zur Beschreibung von Bibliothekseigenschaften bereitstellen, muss unter Unix eine von der NPAPI vorgegebene zusätzliche Bibliotheksfunktion implementiert werden, welche die Eigenschaften liefert. [Vgl. MDC07f, Plug-in Development Overview: Registering Plug-ins]

Die von der NPAPI spezifizierten Funktionen betreffen beide Seiten der Plugin-Schnittstelle – ein Teil wird vom Webbrowser bereitgestellt, der andere Teil kann vom Plugin implementiert werden.[3] Auf diese Weise lassen sich Informationen in beide Richtungen austauschen. Die spezifizierten Funktionen decken mehrere Funktionalitätsbereiche ab. Tabelle 3.1 gibt einen Überblick. Die individuellen Funktionen sind für die Untersuchung im Rahmen dieser Arbeit nicht relevant. Allerdings erlaubt es

[3] Einige der vom Plugin zu implementierenden Funktionen sind optional – sie müssen nicht implementiert werden, wenn das Plugin den zugehörigen Funktionalitätsbereich nicht nutzt.

die kompakte und vollständige Schnittstellendefinition, die Aufgabenverteilung einer Plugin-Schnittstelle exemplarisch zu betrachten.

| Funktionalitätsbereich | Vom Plugin zu implementieren | Vom Webbrowser zu implementieren |
|---|---|---|
| Initialisierung & Termination | `NP_Initialize`<br>`NP_Shutdown`<br>`NPP_New`<br>`NPP_Destroy` | `NPN_ReloadPlugins` |
| Parameter & Konfiguration | `NPP_GetValue`<br>`NPP_SetValue`<br>`NP_GetValue` | `NPN_GetValue`<br>`NPN_SetValue`<br>`NPN_UserAgent`<br>`NPN_Version` |
| Darstellung | `NPP_HandleEvent`<br>`NPP_SetWindow`<br>`NPP_Print` | `NPN_ForceRedraw`<br>`NPN_InvalidateRect`<br>`NPN_InvalidateRegion` |
| Empfang von Datenströmen | `NPP_NewStream`<br>`NPP_WriteReady`<br>`NPP_Write`<br>`NPP_DestroyStream`<br>`NPP_StreamAsFile` | `NPN_RequestRead` |
| Versand von Datenströmen | | `NPN_NewStream`<br>`NPN_Write`<br>`NPN_DestroyStream` |
| Aufruf von URLs (Post & Get) | `NPP_URLNotify` | `NPN_GetURL`<br>`NPN_GetURLNotify`<br>`NPN_PostURL`<br>`NPN_PostURLNotify` |
| Speicherverwaltung | | `NPN_MemAlloc`<br>`NPN_MemFlush`<br>`NPN_MemFree` |
| Statuszeile | | `NPN_Status` |

Tabelle 3.1: Sämtliche Funktionen der NPAPI [basierend auf MDC07f, eigene tabellarische Aufstellung]

Ein essentieller Block aus fünf Funktionen dient der Initialisierung und Termination von Plugins und Plugin-Instanzen. Der dadurch definierte Lebenszyklus von Plugins wird im Unterabschnitt 3.1.1.4 näher erläutert. Sieben weitere Funktionen erlauben das Abfragen der ausgeführten Version von Webbrowser und Plugin sowie das Abfragen und Setzen von Konfigurationsparametern beider Seiten. Zum einen können so Inkompatibilitäten zwischen den ausgeführten Versionen von Webbrowser und Plugin erkannt werden, zum anderen geben insbesondere die vom Webbrowser bereitgestell-

ten Konfigurationsparameter diverse betriebssystemspezifische Implementierungsdetails wie z. B. das zur grafischen Darstellung zur Verfügung stehende Fenster bekannt. [Vgl. MDC07f, Plug-in Development Overview]

Der interaktiven, grafischen Darstellung dient ein Block von weiteren sechs Funktionen. Insgesamt zwölf Funktionen – davon acht durch den Webbrowser bereitgestellte und sechs auf Seiten des Plugins optional zu implementierende – ermöglichen dem Plugin den Empfang und Versand von Daten auf Basis der im Webbrowser enthaltenen Internet-Funktionalität. Mehrere dieser Funktionen bedingen einander, da sie erst im Wechselspiel von Anfrage und Benachrichtigung sinnvoll nutzbar sind. Mit drei Funktionen, die dem Plugin die Anforderung und Freigabe von Speicherbereichen erlauben, die unter der Verwaltung des Prozesses des Webbrowsers stehen, sowie einer Funktion, die dem Anwender kurze Nachrichten in der Statuszeile des Webbrowsers präsentieren kann, ist die NPAPI vollständig beschrieben. [Vgl. MDC07f]

Die aktuelle Erweiterung der NPAPI namens „NPRuntime“ [Ste04] definiert 18 weitere, vom Webbrowser zu implementierende Funktionen, die im Wesentlichen der Erzeugung, Manipulation und Speicherverwaltung von Objekten der angebundenen Skriptsprache dienen. Ein Plugin, dass die Erweiterung nutzen möchte, muss lediglich einen neuen Aufrufparameter der Funktion `NPP_GetValue` erkennen, um solche Objekte liefern zu können. Die Aufrufbarkeit von nativen Methoden dieser Objekte aus Skripten heraus wird direkt durch den Webbrowser ermöglicht. Da diese Erweiterung die für die Untersuchung im Rahmen dieser Arbeit relevanten Eigenschaften des Plugin-Systems nicht wesentlich beeinflusst, soll sie im Folgenden nicht weiter betrachtet werden.

Für die Untersuchung im Rahmen dieser Arbeit sind aus dieser vollständigen Auflistung der Schnittstellendefinition folgende Erkenntnisse zu entnehmen: Erstens gibt es eine klare Aufgabenteilung zwischen Browser und Plugin. Die Schnittstelle des Webbrowsers stellt dem Plugin einen umfangreichen Kontext zur Verfügung, der Funktionalität zum Netzwerk-Zugriff, zum Betrieb der grafischen Benutzungsoberfläche und sogar betriebssystemähnliche Funktionalität wie eine Speicherverwaltung beinhaltet. Die Funktionalität des Plugins kann sich in diesem Kontext auf die spezielle Interpretation der übertragenen Daten beschränken. Zweitens steuert der Webbrowser den Plugin-Lebenszyklus und gibt damit die Zeitpunkte zum Laden, zur Initialisierung und zur Nutzung der Plugin-Funktionalität vor. Drittens sind Callback-Mechanismen vorhanden, um die Funktionalität beider Seiten effizient einander integrieren zu können. Und viertens erfordert die automatisierte einfache Konfiguration eines flexibel im Einsatz komponierbaren Systems offensichtlich einen gewissen Aufwand in der Schnittstellendefinition.

#### 3.1.1.4 Lebenszyklus

Der Lebenszyklus eines Netscape-Plugins wird in [MDC07f] durch vier Funktionen zur Initialisierung und Termination des Plugins sowie durch eine Beschreibung des Webbrowser-Verhaltens definiert. In Abbildung 3.1 sind die einzelnen Zustände und die durch die Funktionen definierten Übergänge im Überblick dargestellt. Modelliert wird der Lebenszyklus als gefärbtes Petrinetz, wie es im Unterabschnitt 4.4.2.1 erläutert

wird. Jedes Plugin lässt sich in diesem Netz als eine Marke repräsentieren, die zwischen den beiden hell eingefärbten Stellen wechseln kann.

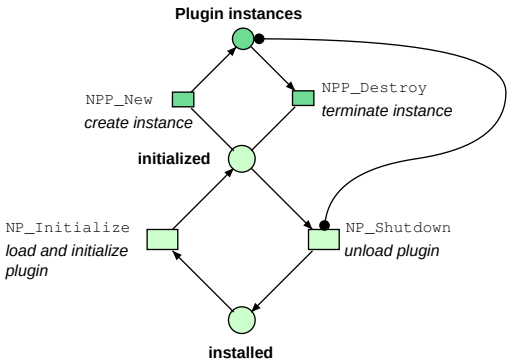



Abbildung 3.1: Der Lebenszyklus von Netscape-Plugins als Petrinetz

Die drei Stellen des Netzes repräsentieren dabei drei verschiedene Existenzebenen der Pluginfunktionalität. Zum ersten besteht ein Plugin aus einer *Datei* im Dateisystem (die je nach Betriebssystem durch externe Eigenschaftslisten ergänzt wird). Die Stelle „installed“ repräsentiert Plugins, die momentan nur in dieser Ebene existieren. Zum zweiten muss sich der in der Datei enthaltene ausführbare *Code* des Plugins während seiner Verwendung im Speicher des Webbrowser-Prozesses befinden. Plugins, die sowohl als Datei als auch als initialisierter Code existieren, werden in der Stelle „initialized“ abgelegt. Zum dritten wird vom Webbrowser je Objekt, dass im Webbrowser zu präsentieren ist, eine *Instanz* des Plugins erzeugt. Enthält die aufgesuchte Webseite mehrere Objekte, deren Typ auf das Plugin passt, werden auch mehrere Instanzen des Plugins parallel eingesetzt.

Strenggenommen betrachtet man auf der letzten Existenzebene nicht mehr den Lebenszyklus des Plugins, sondern seiner Instanzen. Da aber die Funktionalität eines Plugins nicht ohne eine Instanz genutzt werden kann, ist diese Ebene in das Petrinetz integriert. Die zugehörigen Stellen und Transitionen sind zur Kennzeichnung dunkler eingefärbt.

Die Transitionen des Netzes repräsentieren die vier vom Plugin zu implementierenden Funktionen der NPAPI, die die Übergänge zwischen den Existenzebenen definieren. Die Steuerung der Übergänge liegt beim Webbrowser, das Plugin reagiert auf die Funktionsaufrufe. Wird die Funktionalität eines Plugins erstmalig benötigt, lädt der Webbrowser den Code aus der Datei und ruft `NP_Initialize` auf. So wird das Plugin darüber informiert, dass es geladen wurde, und es kann alle für seine Funktionalität benötigten Ressourcen anfordern. Anschließend fordert der Webbrowser je darzustellendem Objekt das Plugin per `NPP_New` auf, eine Instanz zur Präsentation des Objekts zu erzeugen.

Verlässt der Anwender die Seite mit dem darzustellenden Objekt, oder schließt er das Fenster, so wird das Plugin vom Webbrowser per `NPP_Destroy` aufgefordert, die

Instanz zu terminieren. Falls alle Instanzen des Plugins terminiert wurden, kann der Webbrowser das Plugin mittels `NP_Shutdown` auffordern, alle Ressourcen freizugeben, damit er den Code des Plugins wieder aus dem Speicher entfernen kann. Die Inhibitorkante zur Transition „unload plugin“ stellt im Netz sicher, dass beim Entladen des Plugins keine Instanzen des Plugins mehr existieren.

Das Netz aus Abbildung 3.1 beschreibt nur die Zustände, die ein Plugin einnehmen kann, während der Webbrowser ausgeführt wird. Zur vollständigen Betrachtung eines Plugin-Lebenszyklus gehört aber auch die Persistenz von installierten Plugins, während die Webbrowser-Anwendung nicht ausgeführt wird. Eine erweiterte Version des Petrinetzes, die auch die persistente Installation oder Deinstallation von Plugins berücksichtigt, ist in Abbildung 3.2 dargestellt.

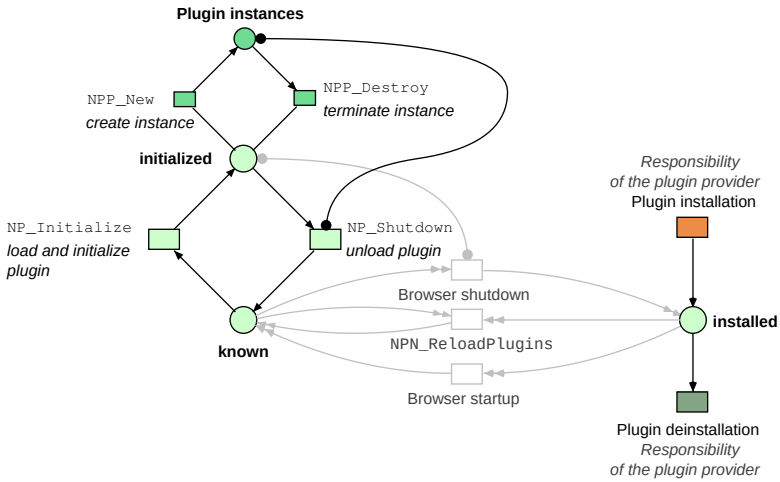



Abbildung 3.2: Der vervollständigte Lebenszyklus von Netscape-Plugins

Da die Installation und die Deinstallation von Netscape-Plugins in der Regel außerhalb der Kontrolle des Webbrowsers stattfinden, muss der Zustand „installed“ feiner unterschieden werden: Im vervollständigten Netz bezeichnet „installed“ die Existenz der Datei im Dateisystem, während der Webbrowser nicht ausgeführt wird. Die Stelle „known“ beschreibt hingegen den Umstand, dass die Webbrowser-Anwendung ausgeführt wird und von der Existenz des Plugins weiß. Die Transitionen „Plugin installation“ und „Plugin deinstallation“ repräsentieren das Einbringen der Datei in bzw. das Entfernen der Datei aus dem System.[4]

Die Übergänge zwischen „installed“ und „known“ hängen nicht vom Plugin oder seinen aktiven Existenzebenen ab, sondern ausschließlich vom Webbrowser. Beim Start der Webbrowser-Software („Browser startup“) sucht diese nach installierten Plugin-Dateien und liest sämtliche Eigenschaften der Plugins ein. Das Plugin selber bekommt

[4] Die besonderen Probleme einer betriebssystemabhängigen Installation von Plugins, die zudem noch außerhalb der Kontrolle des Browsers stattfindet, werden in Unterabschnitt 3.1.1.5 diskutiert.

hiervon nichts mit. Die Kenntnis der installierten Plugins bleibt bis zum Beenden der Webbrowser-Software („Browser shutdown“) unverändert erhalten. In neueren Versionen der NPAPI bietet der Webbrowser mit der Funktion `NPN_ReloadPlugins` die Möglichkeit, erneut (wie beim Programmstart) die Eigenschaften aller installierten Plugins neu einzulesen.[5] Dies erlaubt die Nachinstallation oder ein Update von Plugin-Software. Um zu verdeutlichen, dass a) sämtliche Plugins auf einmal betroffen sind und b) die Übergänge nicht vom Plugin selber wahrgenommen werden, sind die Übergänge zwischen „installed“ und „known“ durch blassgraue Transitionen mit flexiblen Kanten dargestellt.

De facto leert die Transition „Browser shutdown“ nicht nur die Stelle „known“, sondern alle drei Stellen auf der linken Seite des Netzes. Die Dokumentation der NPAPI deutet an, dass der Webbrowser vor dem Beenden alle Plugin-Instanzen beendet [vgl. MDC07f, NPP_Destroy], also sämtliche Marken aus den Stellen „Plugin instances“ und „initialized“ durch Schalten der geeigneten Transitionen abzieht. Dieses Verhalten wird im Petrinetz durch die Inhibitorkante zwischen „Browser shutdown“ und „initialized“ erzwungen. Ob die Funktion `NPN_ReloadPlugins` ein ähnliches Verhalten hat, war aus der Dokumentation nicht zu entnehmen.

Das vervollständigte Lebenszyklus-Netz für Netscape-Plugins besteht somit aus zwei Teilen, die lediglich durch Zustandsübergänge der Webbrowser-Software miteinander verbunden sind. Aus Sicht des ausgeführten Plugins sind nur die Zustände des linken Teils der Abbildung 3.2 wahrnehmbar. Der rechte Teil sowie die verbindenden Übergänge sind aber wichtige Bestandteile der Softwarearchitektur des Netscape-Plugin-Systems, da sonst die für Plugin-Systeme elementare Rekonfigurierbarkeit durch Installation von Plugins nicht beschrieben wäre.

#### 3.1.1.5 Installation

Netscape-Plugins werden, wie bereits in Unterabschnitt 3.1.1.2 erläutert, als betriebssystem- und prozessorarchitekturabhängige Softwarebibliothek ausgeliefert. Die Installation von Plugins läuft jedoch auf allen Betriebssystemen weitgehend ähnlich ab: Im Dateisystem wird in der Nähe der Webbrowser-Anwendung ein spezielles Verzeichnis ausgewiesen, dass der Webbrowser beim Programmstart nach Plugins durchsucht. Alle in diesem Verzeichnis liegenden Dateien mit zum Betriebssystem passenden Softwarebibliotheken gelten als installierte Plugins.

Im ursprünglichen Entwurf der Netscape-Plugin-Architektur ist vorgesehen, dass die Installation von Plugins außerhalb der Laufzeit des Webbrowsers durch den Plugin-Hersteller erfolgt. Neuere Versionen einiger Webbrowser bringen ergänzend eine eigene Plugin-Installations- und Aktualisierungsroutine mit (z. B. das in Unterabschnitt 3.1.2.4 beschriebene *XPInstall*), ändern dadurch aber nichts an der Betriebssystemabhängigkeit der Plugin-Dateien. Die Installation durch die Webbrowser-Software während der Laufzeit vereinfacht vor allem für die Anwender das Finden geeigneter Plugins beim Antreffen multimedialer Inhalte, für die noch kein passendes Plugin installiert ist.

[5] Die Funktion `NPN_ReloadPlugins` wird nicht von allen Webbrowsern unterstützt, obwohl sie in der NPAPI definiert ist. So ist z. B. in der mit dem Netscape Communicator 4.0 veröffentlichen Dokumentation der NPAPI [Net98] die Funktion von vorneherein als *not supported* vermerkt.

Die Installation durch den Plugin-Hersteller ist dennoch weit verbreitet, vereinfacht sie es doch Herstellern komplexer Anwendungen, die eigentliche Anwendungssoftware gleichzeitig als NPAPI-konformes Plugin auszuliefern. Ein prominentes Beispiel ist das PDF-Anzeigeprogramm Adobe Reader, das sowohl als eigenständige Anwendung als auch als Netscape-Plugin arbeiten kann. Die Installationsroutine der Plugin-fähigen Anwendung muss also lediglich eine Softwarebibliothek im Plugin-Installationsverzeichnis des Webbrowsers ablegen, die als Brücke zwischen den Funktionen der NPAPI und der Anwendungsfunktionalität fungiert.

Dabei entsteht allerdings für die Installationsroutine des Plugin-Herstellers ein komplexes Problem: Jeder Webbrowser, der die NPAPI implementiert, hat eine eigene Konvention, in welchem Verzeichnis er nach Plugins sucht. Meist liegt dieses Verzeichnis innerhalb des Installationsverzeichnisses des Webbrowsers, welches wiederum vom Anwender beliebig gewählt worden sein kann. Die Installationsroutine des Plugins muss also die auf dem Rechner installierten Webbrowser finden und die jeweilige Konvention des Webbrowsers kennen, um je eine Kopie des Plugins in das jeweils korrekte Verzeichnis installieren zu können.

Naturgemäß ist das Finden des Webbrowser-Installationsverzeichnisses nur möglich, wenn der Webbrowser zeitlich *vor* dem Plugin installiert wurde. Wählt der Anwender eine Reihenfolge, in der der Webbrowser *nach* der Plugin-fähigen Anwendung installiert wird, so spricht man vom *First Install Problem* [vgl. MDC07h]. Zur Lösung dieses Problems werden je nach Betriebssystem verschiedene Wege gegangen: Betriebssysteme der „Mac OS“-Familie sehen ein anwendungsunabhängiges zentrales Plugin-Verzeichnis vor, das allen Webbrowsern und Plugin-Herstellern bekannt ist. Für Systeme der „Microsoft Windows“-Familie wird in dem Artikel [MDC07h] vorgeschlagen, dass alle Plugin-Installationsroutinen in der systemweiten Konfigurationsdatenbank (*Registry*) an bekannter Stelle jeweils einen Verweis auf den Speicherort ihrer Plugin-Datei hinterlassen. Für Systeme der „Unix“-Familie deutet der Artikel an, dass es vermutlich keine zentrale Lösung geben wird.

Im Zusammenhang mit der Lösung des First-Install-Problems wird die Einführung global eindeutiger Plugin-Identifikatoren (*Plugin identifier*) [Moz02] vorgeschlagen, die mit Angaben zu Herstelleradresse, unterstütztem MIME-Typ und der vorliegenden Version des Plugins angereichert sind. Diese Identifikatoren erlauben auch ein gezieltes Auflösen des Konflikts zwischen mehreren Plugins, die denselben MIME-Typ verarbeiten können, wenn das zu präsentierende multimediale Objekt mit einem zu präferierenden Plugin-Identifikator versehen ist.

### 3.1.2 Extensions

Die Familie der Anwendungen des Mozilla-Projekts ist zwar vom Quellcode her ein Nachfolger des Netscape Communicators, verfügt aber über eine grundlegend überarbeitete Softwarearchitektur. Durch die Aufteilung des Codes in einzelne Komponenten und die Integration neuer Technologien zur Beschreibung grafischer Benutzungsoberflächen ist eine flexible Softwarearchitektur entstanden, so dass die Anwendung von Programmieren und Anwendern in vielfältiger Weise angepasst oder ergänzt werden kann. Insbesondere die jüngeren Anwendungen des Projekts (z. B. der Webbrowser „Firefox“ oder der E-Mail-Client „Thunderbird“) nutzen die Vorteile dieser Architektur –

sie basieren auf einer breiten, gemeinsamen Basis an grundlegenden Softwarekomponenten und bieten darauf aufbauend spezialisierte Funktionalität.

Zudem erlaubt die Softwarearchitektur Dritten, Erweiterungen an den Anwendungen des Mozilla-Projekts vorzunehmen und diese auf einfache Weise den Anwendern zugänglich zu machen. Installierfertig bereitgestellte Ergänzungen werden als *Add-on* bezeichnet. Der Begriff der Plugin-Architektur drängt sich in diesem Zusammenhang zwar auf, wird aber nur selten verwendet [z. B. in Bak03] – vermutlich, da der Begriff „*Plugin*“ im Kontext der Mozilla-Anwendungsfamilie durch die ältere Definition der NPAPI schon belegt ist.

Auf der Webpräsenz unter `http://addons.mozilla.org/` steht eine Vielzahl von *Add-ons* zur Installation bereit. Dabei handelt es sich zum einen um so genannte Erweiterungen (*Extensions*), die zusätzliche Funktionalität in die Anwendung integrieren. Zum anderen werden neben Extensions auch Themen (*Themes*), Wörterbücher (*Dictionaries*), Suchmaschinen (*Search engines*) oder Plugins (im Sinne der NPAPI) angeboten.[6] Alle diese Ergänzungen werden unter dem Begriff „*Add-on*“ zusammengefasst und mit einer einheitlichen Technik namens *XPInstall* (*Cross(**X**)-**P**latform Install*) auf dem System des Anwenders installiert. Bei Themen, Wörterbüchern und Suchmaschinen handelt es sich ebenso wie bei der bereits in Unterabschnitt 3.1.1 ausführlich behandelten NPAPI um Plugin-Schnittstellen mit spezialisiertem Funktionsumfang. Extensions hingegen stellen eine wesentlich flexiblere Ergänzungsmöglichkeit dar und werden daher im Folgenden näher betrachtet.

Eine Mozilla-Extension ist – ebenso wie die Mozilla-Anwendung selbst – ein so genanntes Paket (*package*), in dem sowohl die grafische Benutzungsoberfläche als auch die dahinterstehende Funktionalität definiert werden. Die Mozilla-Softwarearchitektur ermöglicht dabei eine Trennung der Oberflächengestaltung von der Implementierung der Funktionalität. Die Oberfläche wird in Form von XML-basierten Beschreibungssprachen spezifiziert, während die Funktionalität mittels klassischer Programmiersprachen implementiert wird. Die Interpretation der Oberflächenbeschreibung übernimmt die eigentliche Webbrowser-Engine namens *Gecko*, so dass dieselbe Technik zur Darstellung von Inhalten des World Wide Web und zur Darstellung der Anwendungsoberfläche verwendet wird. Komplexe Funktionalität kann in Form von Komponenten des Mozilla-eigenen, plattformübergreifenden[7] Komponentenmodells namens *XPCOM* (*Cross(**X**)-**P**latform **C**omponent **O**bject **M**odel*) realisiert werden. Beispielsweise ist

[6] Themen passen die Gestaltung und Benutzung der grafischen Oberfläche der Anwendung an. Wörterbücher dienen der Rechtschreibprüfung, wenn der Anwender auf einer Webseite oder in einer E-Mail längere Texte tippt. Bei Suchmaschinen handelt es sich um eine von externen Webservern angebotene Dienstleistung, deren Aufruf allerdings in einer speziellen Eingabezeile des Webbrowsers vorkonfiguriert werden kann.

[7] Die in den Akronymen des Mozilla-Projekts häufig auftauchende Kombination *„XP“* steht für „*cross-platform*“, also *plattformübergreifend*. McFarlane interpretiert diesen Begriff in [McF03, S. 7] als *Portabilität*, also Kompilierbarkeit des Quellcodes auf unterschiedlichen Betriebssystemen und Plattformen. Betrachtet man die im Mozilla-Projekt verwendeten Techniken, könnte man auf den höheren Abstraktionsebenen der Mozilla-Softwarearchitektur sogar von *Plattformunabhängigkeit* sprechen, da XML-Beschreibungen und Skriptsprachen nicht plattformspezifisch kompiliert werden müssen. Dennoch finden sich auch in diesen Beschreibungen und Programmen häufig Definitionen, die nur auf einzelnen Betriebssystemen einen Sinn haben und der besseren Integration der Funktionalität in die Oberflächengestaltung des jeweiligen Systems dienen.

die Webbrowser-Engine Gecko selber in Form von XPCOM-Komponenten implementiert.

Ein Mozilla-Paket beinhaltet also ein oder mehrere Elemente, die je nach Aufgabe des Pakets in verschiedenen Sprachen verfasst sein können: Die Bedienelemente (*Widgets*) der Oberfläche werden in der XML-basierten Benutzungsoberflächensprache namens *XUL* (***X**ML **U**ser Interface **L**anguage*) spezifiziert. Die grafische Gestaltung der Oberfläche wird mittels der im World Wide Web verwendeten Technik der separaten Stilvorlagen (***C**ascading **S**tyle **S**heets, CSS*) beschrieben. Einfache funktionale[8] Anteile der Anwendungsoberfläche werden ebenso wie die Ablaufkontrolllogik mittels der ebenfalls im Web verwendeten Skriptsprache *JavaScript* implementiert. Dabei können die Skripte auf die Dienste sämtlicher XPCOM-Komponenten der Anwendung zurückgreifen. Zudem kann ein Paket auch eigene Komponenten bereitstellen, die in beliebigen Programmiersprachen mit XPCOM-Anbindung implementiert sein dürfen.[9] Somit hat eine Extension die Möglichkeit, auf allen Ebenen der Mozilla-Softwarearchitektur in der jeweils geeigneten Sprache (C++, JavaScript, XUL oder andere) Funktionalität zu ergänzen. [Vgl. Tin05, MO05]

Im Folgenden sollen vier Techniken kurz vorgestellt werden, die den Plugin-Charakter der Mozilla-Extensions definieren: *XPCOM* stellt das grundlegende Komponentenmodell bereit, durch dessen Implementierung die Komponenten zu einer Anwendung integrierbar werden. Die Struktur eines *Pakets* wird von der Mozilla-Architektur vorgegeben und gliedert die verschiedenen Anknüpfungspunkte für Erweiterungen. Die Technik der *XUL-Overlays* erlaubt das Überlagern bestehender Elemente der Oberfläche der erweiterten Anwendung und ermöglicht so das Ersetzen oder den Wegfall von Funktionalität durch eine Erweiterung. *XPInstall* ist die einheitliche Technik zur Installation von Paketen, die als Teil des *Mozilla-Toolkits* in eine anwenderfreundliche Software-Verwaltung integriert ist.

#### 3.1.2.1 Komponenten (XPCOM Components)

Das *Cross(**X**)-**P**latform **C**omponent **O**bject **M**odel* ist ein vom Mozilla-Projekt definiertes, einfaches Komponentenmodell, das auf den Einsatzzweck einer portablen und flexiblen Objektverwaltung innerhalb einer einzigen Anwendung spezialisiert ist. Daher wird die direkte Interaktion von Komponenten über Prozessgrenzen oder gar Rechnergrenzen hinweg nicht unterstützt. Im Vergleich zu den in Unterabschnitt 2.2.2 vorgestellten prominenten Komponentenmodellen orientiert sich XPCOM stark am Vorbild COM (***C**omponent **O**bject **M**odel*) – einer Vorgängerversion von COM+, die nur einfache Fähigkeiten zur prozessübergreifenden Verwaltung von Objekten (*common object management*) beinhaltet. In Ergänzung zum Vorbild COM werden die Schnittstellen der Komponenten in einer Schnittstellenbeschreibungssprache namens *XPIDL* (*Cross(**X**)-**P**latform **I**nterface **D**efinition **L**anguage*) spezifiziert, die der CORBA-IDL ähnelt. [Vgl. McF03, Abschnitt 1.3.1.2, S. 13f.]

[8] Das Adjektiv „*funktional*“ wird in weiten Teilen dieser Arbeit nicht im mathematischen Sinne, sondern in der Bedeutung „*funktionalitätsorientiert*“ verwendet.

[9] Die hauptsächlich verwendete Programmiersprache für Komponenten ist C bzw. C++, wobei (wie bei Netscape-Plugins) auf die damit einhergehende Plattformabhängigkeit geachtet werden muss. Zwar ist der Quellcode einer XPCOM-Komponente idealerweise portabel, aber die kompilierten Artefakte im auslieferbaren Paket müssen auf die jeweiligen Zielplattformen abgestimmt sein.

Zu den Aufgaben der XPCOM-Implementierung gehört zunächst eine Objektverwaltung, welche eindeutige Namen für Komponentenverträge (*Contract ID*), Komponentenimplementierungen (*Class ID, CID*) und deren Schnittstellen (*Interface ID, IID*) verlangt sowie Mechanismen zum Instantiieren von Objekten gemäß dem Factory-Pattern, zum Zählen der Referenzen auf ein Objekt und zur Behandlung von Ausnahmesituationen (*Exceptions*) definiert. Jede XPCOM-Komponente muss sich an den Vertrag der Objektverwaltungsschnittstelle halten.[10] Die zusätzliche Beschreibung der Schnittstellen in der XPIDL erlaubt die Implementierung oder Verwendung von Komponenten in anderen Sprachen wie beispielsweise JavaScript oder Python. Aus einer XPIDL-Beschreibung lassen sich sowohl C++-Deklarationen als auch binäre Typbibliothek-Dateien (*Typelibs, XPT*) zur Verarbeitung in Komponentenladern für andere Sprachen erzeugen. [Vgl. TO03, Kapitel 1]

XPCOM-Komponenten werden in Form dynamischer Bibliotheken in die Anwendung eingebunden, die spezifisch für jedes Betriebssystem, jede Prozessorarchitektur und jede Binärschnittstelle kompiliert werden müssen (ebenso wie NPAPI-Plugins, siehe Unterabschnitt 3.1.1.2). Um die verschiedenen Kompilate einer Plattform zuordnen zu können, werden die Binärschnittstellen nach einer einheitlichen Nomenklatur differenziert [MDC06e]. In C++ verfasste Komponenten sind direkt in einer Bibliothek enthalten, in anderen Programmiersprachen verfasste Komponenten greifen auf einen in C++ implementierten generischen Komponentenlader für die jeweilige Sprache zurück.

Der Code der Bibliothek muss vier vorgegebene Schnittstellen realisieren: Die Programmierschnittstelle der *dynamischen Bibliothek* umfasst nur einen einzigen Funktionsbezeichner, über den die XPCOM-Laufzeitumgebung ein so genanntes Modul erfragt. Die *Modul-Schnittstelle* ist verantwortlich für das Bereitstellen von Fabriken für XPCOM-Objekte sowie für die Bekanntmachung dieser Fabriken im XPCOM-System. Jede Implementierung der *Fabrik-Schnittstelle* ist für die Instantiierung von XPCOM-Objekten auf Basis einer bestimmten Klasse zuständig – wobei der Begriff „Klasse“ in der XPCOM-Architektur synonym mit „Komponente“ gebraucht wird.[11] Jede Klasse muss die *Schnittstelle* `nsISupports` implementieren, wodurch die Klasse (und alle ihre Instanzen) selber für die Referenzzählung und die Konvertierung von Objektreferenzen zwischen den unterstützten Interface-IDs zuständig ist. [Vgl. TO03, S. 46–48]

XPCOM pflegt zwei persistente Datenbanken (*Registries*) der installierten Komponenten und Typbibliotheken. Die Komponenten-Registry ordnet Contract IDs, Class IDs und die im Dateisystem installierten dynamischen Bibliotheken einander zu. Die Typ-Registry bildet Interface IDs auf installierte Typbibliotheken ab. Die Registrierung aller Komponenten in der Datenbank kann zwar beim Start der Anwendung automatisch durchgeführt werden, in den ausgelieferten Anwendungen der Mozilla-

[10] Die Objektverwaltungsfunktionalität wäre bei vielen objektorientierten Programmiersprachen bereits im Standardumfang enthalten, so auch (zumindest teilweise) bei der XPCOM zugrundeliegenden Sprache C++. Das Mozilla-Projekt hat sich aber aus Gründen der Performance und der Portabilität [vgl. TO03, S. 17] für eine eigene Implementierung auf Basis einfacher C-Mechanismen entschieden. Der Preis dafür ist, dass die Implementierung jeder einzelnen Komponente eine mögliche Fehlerquelle in der Objektverwaltung darstellen kann.

[11] McFarlane reißt in [McF03, S. 621f.] kurz die Vielfältigkeit der in den verschiedenen Schichten der Mozilla-Architektur verwendeten Modularisierungsbegriffe an.

Familie wird diese Möglichkeit aber nicht genutzt. Stattdessen nimmt ein Entwicklerwerkzeug oder das XPInstall-Werkzeug (siehe weiter unten) eine explizite Aktualisierung der Datenbank vor. Die Registrierung der einzelnen Komponenten eines Moduls wird nicht anhand externer Beschreibungen, sondern direkt vom Code der Bibliothek selber vorgenommen – der XPCOM-Registrierprozess fordert jedes Modul dazu auf. Ebenso wird beim De-Registrieren die aktive Mitarbeit des Moduls benötigt. [Vgl. TO03, S. 42–45]

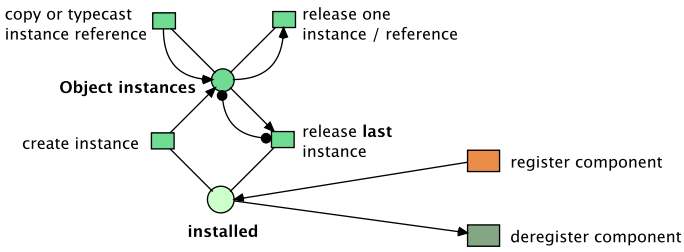



Abbildung 3.3: Das Lebenszyklus-Netz von XPCOM-Komponenten

Aus den eben beschriebenen Mechanismen lässt sich ein sehr einfacher „Lebenszyklus“ wie in Abbildung 3.3 ableiten. Aus Sicht des XPCOM-Systems beginnt die Existenz einer Komponente mit ihrer Registrierung – vorher sind auch dynamische Bibliotheken in geeigneten Verzeichnissen dem System nicht zugänglich. Da die Registrierung über die Laufzeit der XPCOM-Umgebung hinaus persistent ist, entfällt die Aufteilung des Zustandes „installed“ in zwei von der Anwendung unterschiedene Zustände wie bei NPAPI-Plugins (vgl. Abbildung 3.2). Zudem wird in der XPCOM-Architektur (anders als z. B. bei der NPAPI) keine offizielle Unterscheidung zwischen registrierten (aber noch nicht geladenen) und im Speicher befindlichen Code-Bibliotheken vorgenommen. Es steht der Komponentenimplementierung frei, auf die erste Instantiierung eines Objektes speziell zu reagieren. Falls eine Komponente über Start oder Ende des gesamten XPCOM-Systems informiert werden möchte, so kann sie das über einen zentralen, nach Themen gegliederten Nachrichtendienst (Interface `nsIObserverService`) erfahren.

Da die XPCOM-Architektur den Komponenten auferlegt, über ihre instantiierten Objekte selber zu wachen, sind die vier kleineren, dunkler eingefärbten Transitionen zum Zählen der Referenzen in Abbildung 3.3 mit eingezeichnet.[12] Die Komponente darf bzw. soll auf die Freigabe ihrer letzten Objektinstanz mit einer Freigabe aller ihrer Ressourcen reagieren. Das De-Registrieren von Komponenten ist in der Architektur zwar vorgesehen und muss von den Komponenten unterstützt werden. Anders als zur

[12] Alle vier Transitionen verwenden Testkanten, um sich auf ein der erzeugten oder freigegebenen Instanz zugrundeliegendes Objekt (Komponente oder eine andere Instanz) zu beziehen. Die Transition zum Freigeben der letzten Objektinstanz wird durch eine Inhibitorkante mit dem Kantengewicht 1 implementiert. Diese Kante aktiviert die Transition nur, wenn eine oder weniger Marken auf der Stelle „Object instances“ liegen. Da zusätzlich eine Eingangskante von derselben Stelle existiert, muss *exakt* eine Marke auf der Stelle liegen, um die Transition zu aktivieren.

Registrierung finden sich aber kaum Hinweise in den Quellen, wann eine De-Registrierung genau stattfindet. Ebenso unklar ist die Verwendung der `canUnload`-Funktion der Modul-Schnittstelle `nsIModule`: Laut der aktuellen Referenzdokumentation kann die XPCOM-Umgebung diese Funktion nutzen, um die Zustimmung der Komponente zum Entladen ihrer dynamischen Bibliothek zu erhalten – allerdings findet dieses Entladen (zumindest in der ca. 2004 aktuellen Mozilla-Version 1.7) nicht statt.[13]

Der Zugriff auf ein XPCOM-Objekt erfolgt über den zentralen `ComponentManager`, der eine ContractID oder eine ClassID mittels der Komponenten-Registry zur passenden Fabrik hin auflösen kann. Die Funktion `CreateInstance` der Fabrik (oder auch des Component Managers) erzeugt ein Objekt der Klasse, welches dann verwendet und später wieder freigegeben werden kann. Die XPCOM-Entwickler tragen daneben einem Bedarf an langlebigen Objekten, die bestimmte Dienstleistungen (*Services*) erbringen, Rechnung (der Component Manager ist selbst ein Beispiel dafür). Als einfache Implementierung ist vorgesehen, dass eine Fabrik den Dienstleistungscharakter der Klasse kennt und dementsprechend immer dasselbe Objekt liefert, statt neue Instanzen zu erzeugen (Singleton-Pattern). Daneben bietet ein Dienstverwalter-Objekt (*Service Manager*) die Möglichkeit, Referenzen auf dienstleistende Singleton-Objekte global zu erhalten. Die erste Referenz auf die Komponenten- oder den Dienstverwaltung kann über globale Funktionen des XPCOM-Kerns erhalten werden. [Vgl. TO03, S. 23, 200ff. u. 253ff.]

Abhängigkeiten zwischen Komponenten werden von XPCOM nicht explizit verwaltet. Auf Code-Ebene verlässt sich das Modell offensichtlich auf die von der Programmiersprache C++ verwalteten Abhängigkeiten und den Mechanismus des Verknüpfens des Codes dynamischer Bibliotheken mit dem Code der XPCOM-Laufzeitumgebung. Wenn eine Komponente die Funktionen einer anderen XPCOM-Komponente benötigt, so muss sie jeden Versuch, eine Instanz der anderen Komponente über den Component Manager zu erhalten, explizit auf seinen Erfolg hin überprüfen.

#### 3.1.2.2 Pakete (Chrome Packages)

Die Mozilla-Anwendungssuite führt oberhalb der generischen XPCOM-Architektur eine weitere Komponentenschicht zur Modularisierung der grafischen Benutzungsoberfläche ein. Diese Schicht, die darin enthaltenen Komponenten als auch die so gestalteten Oberflächen werden unscharf als „*Chrome*“ bezeichnet. Die Komponenten dieser Schicht werden Pakete (*Packages*) genannt. Eine Anwendung kann mehrere Chrome-Pakete umfassen, welche in ihrer Kombination die Struktur, die Gestaltung und das Verhalten der Anwendung bestimmen. Die Pakete werden in einer Datenbank namens „*Chrome-Registry*“ verwaltet.[14] Die XPCOM-Komponenten der Mozilla-Softwarear-

[13] Originaltext vom 1. Juli 2009:

> „If the component module is native (i.e., as part of a DLL), then this method may be called to determine whether or not the DLL may be unloaded from memory.
>
> Note that as of Mozilla 1.7, native component modules are never unloaded.“
> (`http://developer.mozilla.org/en/NsIModule/canUnload`)

[14] Die Bezeichnung „*Chrome*“ wird in der Mozilla-Entwicklergemeinde in verschiedenen Ausprägungen verwendet. Häufig wird der Begriff „*Chrome*“ zur Bezeichnung von Paketen allgemein oder zur Benennung des einen, übergeordneten Paketes verwendet, welches den Rahmen für die gesam-

chitektur (vor allem die Gecko-Implementierung) können so als Basis für verschiedene Anwendungen wiederverwendet werden, indem ein alternativer Satz an Paketen eine eigenes Chrome mit eigener Funktionalität implementiert. Mozilla-*Extensions* sind von der Konvention her Pakete, deren Struktur fest am Chrome der Mozilla-Anwendungsfamilie ausgerichtet ist, um in diesem Rahmen Ergänzungen an Oberfläche und Funktionalität vorzunehmen.

Ein Paket besteht aus mehreren Dateien, die nach einem vorgegebenen Schema in einem Verzeichnis oder einer Archivdatei (`.jar`) organisiert sind. Typischerweise enthält ein Paket XUL-Oberflächenbeschreibungen („Content“), CSS-Layoutbeschreibungen und Grafiken („Skin“), und/oder Übersetzungstabellen zur Internationalisierung der Oberfläche („Locale“) – diese drei Elemente werden auch als „*Chrome-Provider*“ bezeichnet [vgl. MDC07d]. Zusätzlich können XPCOM-Komponenten („Components“) enthalten sein. Die in Klammern genannten Begriffe entsprechen jeweils einem Unterverzeichnis des Pakets. In jedem dieser Unterverzeichnisse können alternative Implementierungen nebeneinander stehen, die z. B. die Unterbringung plattformspezifischer Bibliotheken, unterschiedlicher Gestaltungsstile oder verschiedener Landessprachen ermöglichen. [Vgl. Oes00]

Die Chrome-Registry dient im Wesentlichen der Zuordnung von logischen Paketnamen zu den entsprechenden Dateien, die überall im Dateisystem verstreut liegen dürfen.[15] Ein *Chrome-Manifest* – eine Textdatei, die in jedem Paket vorhanden sein muss – enthält Information über die zu dem Paket gehörigen Dateien, aber auch Anweisungen zur Integration der Teile des Paketes (Content, Skin oder Locale) in die gesamte Anwendung. Um Extensions in mehrere Anwendungen integrierbar oder zu verschiedenen Binärschnittstellen kompatibel, aber dennoch in einem einzigen Paket gebündelt, ausliefern zu können, können einzelne Einträge des Manifests als anwendungs-, versions-, plattform- oder betriebssystemspezifisch gekennzeichnet werden. Bei der Installation des Paketes werden die Informationen des Manifestes zur Registrierung der Paketteile in der zentralen, persistenten Datenbank der Anwendung – der Chrome-Registry – genutzt. [Vgl. MDC07d]

Die feste Struktur der Mozilla-Pakete in Kombination mit den Angaben im Chrome-Manifest erlauben bereits einen einfachen Plugin-Mechanismus: Beispielsweise kann ein Paket eine zusätzliche Sprachtabelle (Locale) oder eine alternative Layoutvorlage (Skin) für eine durch ein anderes Paket definierte Oberfläche bereitstellen [vgl. Hya99, Overview of a Package]. Sofern das ursprüngliche Paket die Integration alternativer Layouts vorgesehen hat, kann der Anwender dann z. B. in Konfigurationsdialogen be-

te Anwendung definiert [z. B. Oes00, Tin05]. Alternativ kann die Gesamtheit aller zusammengenommenen Pakete oder auch die Verzeichnisstruktur, in der die Pakete untergebracht sind, als *Chrome* bezeichnet werden [z. B. McF03, S. 412]. Zur Verwirrung trägt auch bei, dass die Zeichenfolge “`chrome`“ auch in URLs, Dateinamen oder Kommandozeilenparametern verwendet wird. Einen Überblick über die verschiedenen Verwendungen und Bedeutungen des Begriffs gibt (seit Juli 2007) die Seite [MDC07c].

Im Rahmen dieser Arbeit soll [analog zu MDC07d] *Chrome* die Gesamtheit der Oberflächenbeschreibung bezeichnen, während *Pakete* die einzelnen Komponenten darstellen.

[15] Meist sind die Dateien der Pakete trotzdem an einer von zwei bevorzugten Stellen untergebracht. Neben dem zentralen Installationsverzeichnis der Anwendung als Installationsort bieten sich auch die Profilverzeichnisse an, von denen jeder Benutzer über eins oder mehrere verfügt, so dass dort individuelle Konfigurationsdaten abgelegt werden können.

liebig zwischen den installierten Alternativen wählen.[16] Die *Add-on*-Untergruppe der „*Themes*“ ist ein Beispiel für die Nutzung dieser Technik, „*Themes*“ bestehen nur aus alternativen Layout-Vorlagen für eines der Haupt-Anwendungspakete der Mozilla-Familie.

Die Zuordnung von Ergänzungen zum ergänzten Paket geschieht anhand einfacher, sprechender Paketnamen (wie z. B. „`browser`“), das ergänzende Paket verwendet schlichtweg denselben Namen in den entsprechenden Zeilen des Chrome-Manifests. Daneben verfügt jedes Anwendungspaket und auch jede Extension über eine global eindeutige Identifikationszeichenfolge. Diese dient im Wesentlichen der Vermeidung von Namenskollisionen in den Standardverzeichnissen zur Installation von Paketen. Ist ein Paket in einem Standardverzeichnis installiert, finden es die neueren Anwendungen der Mozilla-Suite beim nächsten Start automatisch und nehmen es in die Chrome-Registry auf [vgl. MDC07a]. Obwohl die Chrome-Registry vom Entwurf her darauf angelegt ist, zur Laufzeit Änderungen an der Menge der installierten Pakete integrieren zu können, erfordert die Darstellung der umgestalteten Benutzungsoberfläche in der Regel einen Neustart der Anwendung [vgl. McF03, S. 425].

Pakete als Teil des Chrome haben keinen eigenen Kontrollfluss, sondern werden von der Laufzeitumgebung ausgewertet (von XPCOM-Komponenten einmal abgesehen). Die JavaScript-Anteile sind reaktiv ausgelegt, sie verarbeiten hauptsächlich Ereignisse, die durch die Interaktion des Anwenders mit dem System hervorgerufen werden. Daher benötigen Pakete keinen eigenen Lebenszyklus, dennoch lässt sich das Verhalten des Systems wie in Abbildung 3.4 darstellen. In der Stelle „installed“ werden alle

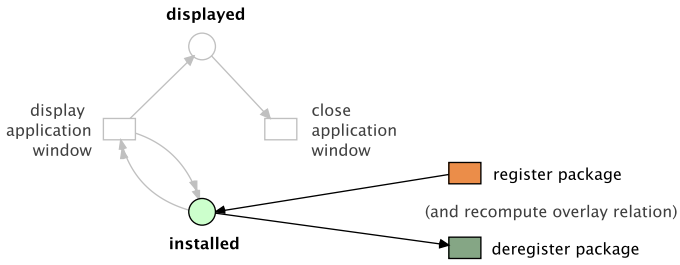



Abbildung 3.4: Das Lebenszyklus-Netz von Chrome-Paketen

installierten und registrierten Pakete abgelegt. Die (im folgenden Unterabschnitt beschriebenen) Querbezüge zwischen den Paketen werden bei der Registrierung bzw. De-Registrierung neu berechnet. Wenn ein Anwendungsfenster dargestellt werden soll, liest die Laufzeitumgebung die notwendigen Daten aller beteiligten Pakete ein und stellt das Fenster dar. Das Schließen des Fensters benötigt keinen Zugriff auf die installierten Pakete. Die Aktionen der Laufzeitumgebung betreffen alle Pakete und werden von den einzelnen Paketen nicht wahrgenommen, daher sind sie blassgrau und unter Verwendung flexibler Kanten dargestellt.

[16] Zur Unterstützung von Locales bietet das Mozilla-Framework einen zentralen Mechanismus, so dass nicht jedes Paket eine eigene Lösung implementieren muss.

#### 3.1.2.3 Integration durch Überlagerung (Overlays)

Die Chrome-Registry bietet mehrere Mechanismen, um nicht nur ganze Chrome-Provider zu anderen Paketen ergänzen zu können, sondern auch Ergänzungen und Ersetzungen an Teilen der Chrome-Provider vorzunehmen. Ein Manifest-Eintrag der Art `override` erlaubt das vollständige *Ersetzen* einer einzelnen Datei eines anderen Pakets durch eine eigene. Manifest-Einträge der Art `style` und `overlay` erlauben daneben die *Integration* eigener CSS- oder XUL-Dateien in die Layout- bzw. Oberflächenbeschreibungen eines anderen Pakets. [Vgl. MDC07d]

Während XUL-Basisdokumente immer die Struktur und die Bedienelemente eines ganzen *Fensters* (Anwendungsfenster, Dialog, etc.) spezifizieren, sind XUL-Overlay-Dokumente nicht ohne ein Basisdokument verwendbar. Ist in der Chrome-Registry vermerkt, dass ein Basisdokument durch ein Overlay-Dokument überlagert werden soll, so werden die Elemente beider Dokumente zusammengeführt. Beide Dokumente müssen zu diesem Zweck die Elemente in derselben XML-Hierarchie anordnen. Dabei dürfen in beiden Dokumenten Knoten oder Äste des XML-Baumes fehlen, in denen keine Überlagerung stattfindet. Die Zuordnung der zu überlagernden Elemente findet anhand von IDs statt, die sowohl im Basisdokument als auch im Overlay-Dokument verwendet werden. Werden zwei Elemente überlagert, so werden die an diesen Elementen hängenden Äste des XML-Baumes vereinigt. Da die Reihenfolge von Elementen in einer XML-Datei mitunter maßgeblich für die Reihenfolge der Darstellung ist (z. B. bei Menüeinträgen), können Elemente mit Positionsangaben versehen werden. Treffen bei der Überlagerung zweier Elemente verschiedene Werte für dasselbe Attribut des Elements aufeinander, so hat das Attribut des Overlay-Elements Vorrang. [Vgl. MDC05]

Die Chrome-Registry pflegt eine persistente Datenbank aller XUL-Dokumente und ihrer Overlay-Beziehung. Overlay-Dokumente können auch explizit im Basisdokument angefordert werden, um z. B. eine modulare Aufteilung oder Wiederverwendung von XUL-Beschreibungen zu ermöglichen. Die Zusammenführung von Basisdokument und Overlay-Dokumenten findet statt, sobald das Basisdokument zur Darstellung angefordert wird. [Vgl. McF03, Abschnitt 12.1, S. 413ff.]

Chrome-Pakete und die Technik der XUL-Overlays erlauben eine sehr flexible und eventuell eigenwillige Anpassung der Benutzungsoberfläche. Um ein einheitliches Erscheinungsbild der Mozilla-Anwendungssuite und ihrer Extensions gewährleisten zu können, werden in [MDC07e] Konventionen zur Oberflächengestaltung und Namensgebung vorgeschlagen. In den Konventionen findet sich auch ein Hinweis, dass aufeinander aufbauende Extensions – also Abhängigkeiten zwischen Extensions – zwar technisch möglich, aber unerwünscht sind.

Die Eigenschaften, dass kein eigentlicher Lebenszyklus definiert ist und dass Änderungen nur nach einem Neustart der Anwendung wirksam werden, gelten für die Overlay-Beziehung genauso wie für Pakete allgemein.

#### 3.1.2.4 Installation (XPInstall/Mozilla-Toolkit)

Die Installation von Mozilla-Paketen oder von XPCOM-Komponenten im System des Anwenders kann auf verschiedene Weisen erfolgen – manuell oder automatisiert,

als einzelne Ergänzung oder gebündelt mit der Kernanwendung. McFarlane gibt in [McF03, Abschnitt 17.1, S. 707ff.] einen Überblick über die jeweils notwendigen Schritte. Insbesondere ist es sowohl bei der Installation von Paketen als auch bei der von Komponenten anders als bei der von NPAPI-Plugins nicht ausreichend, die richtige Datei an der richtigen Stelle im Dateisystem abzulegen. Es muss jeweils die Chrome- oder XPCOM-Datenbank mit Informationen über die installierten Elemente aktualisiert werden – entweder manuell oder per Werkzeug. Zudem pflegt das Mozilla-Kernsystem weitere Datenbanken (*Registries*), die neben Versionierungs- und Deinstallationsinformationen der installierten Komponenten und Pakete auch Konfigurationsdaten enthalten. Einige dieser Datenbanken werden einmal pro installierter Mozilla-Anwendung, andere einmal pro Anwender gepflegt, wobei ein Anwender auch mehrere Profile verwalten kann [siehe McF03, Abschnitt 17.3.2, S. 726ff.]. Finden Änderungen an den Datenbanken außerhalb des Prozesses der Mozilla-Anwendung statt, so werden sie von der Anwendung erst beim Neustart wahrgenommen. Aber auch im Fall, dass die Installation innerhalb des Anwendungsprozesses stattfindet, ist in den meisten Fällen ein Neustart der Mozilla-Anwendung notwendig, bevor die Änderungen wirksam umgesetzt werden können.

Die *XPInstall*-Komponente (*Cross(**X**)-**P**lattform Install*) bietet eine einheitliche, vom Mozilla-Projekt gepflegte Implementierung der immer wiederkehrenden Installationsaufgaben für Pakete und Komponenten an. Um die XPInstall-Komponente zu nutzen, müssen die Dateien lediglich in eine Archivdatei mit der Endung `.xpi` (ausgesprochen „Zippy“) verpackt und mit einer Installationsbeschreibung versehen werden [vgl. MDC06a]. Die XPCOM-Komponente mit der XPInstall-Funktionalität ist in der Regel im Lieferumfang von Anwendungen der Mozilla-Familie enthalten, sie kann aber auch im Rahmen eines separaten Installationsprogramms verwendet werden [siehe MDC02].

In früheren Versionen der XPInstall-Technik (z. B. der Mozilla-Gesamtsuite vor der Aufteilung in Einzelanwendungen) wird die Installationsbeschreibung imperativ in JavaScript verfasst. Das Skript wird im Rahmen der XPCOM-/Gecko-Laufzeitumgebung ausgeführt und darf die zur Verfügung stehenden XPCOM-Komponenten nutzen, darunter insbesondere die Funktionen der XPInstall-Komponente. Die Funktionen arbeiten komplexe Teilschritte der Installation ab, z. B. werden mit einem einzigen Aufruf alle Dateien eines Chrome-Paket-Teiles (z. B. Content, Skin oder Locale) in ein temporäres Verzeichnis kopiert, vorläufig konfiguriert und zur endgültigen Installation vorgemerkt. Die Funktionen unterstützen ein Transaktionskonzept für die gesamte Installation, so dass alle XPCOM-Komponenten, Chrome-Pakete und anderen Teile des `.xpi`-Archivs am Ende eines fehlerfreien Installationsprozesses auf einen Schlag in die Anwendung integriert werden. Diese Technik erlaubt zwar eine flexible Zusammenstellung der `.xpi`-Archive und verbirgt viele plattformspezifische Details (z. B. das Installationsverzeichnis) vor den Skripten, allerdings muss jedes `.xpi`-Archiv die Ablauflogik der Installation erneut enthalten. Beispielsweise müssen Beginn und erfolgreiches Ende der Transaktion ebenso wie der Abbruch im Fehlerfall im Skript explizit über XPInstall-Funktionsaufrufe ausgelöst werden. [Vgl. MDC06c]

In neueren Anwendungen der Mozilla-Familie wird eine deklarative Variante der Installationsbeschreibung unterstützt. Diese Technik wird als Teil des *Mozilla-Toolkits* bereitgestellt, welches auch die Aufgaben der Chrome-Registry, die Verwaltung von

Benutzerprofilen, eine Verwaltung der installierten Extensions und einen Software-Aktualisierungsdienst für Pakete aller Art beinhaltet. Die Dateien im `.xpi`-Archiv müssen entsprechend einer vorgegebenen Struktur [siehe MDC07b] angeordnet sein, im Falle von Chrome-Paketen fallen Paketstruktur und `xpi`-Struktur weitgehend zusammen. Insbesondere die Konvention zur Unterscheidung plattformspezifischer Alternativen ist dieselbe.

Die Installationsbeschreibung besteht bei der deklarativen XPInstall-Technik aus einem weiteren, in der XML-basierten Sprache RDF (*Resource Description Framework*) verfassten Manifest [siehe MDC07g], welches den zentral implementierten Installationsprozess parametrisiert. Die Parameter lassen sich in vier Funktionsbereiche einordnen.[17] Zudem sind einige der Parameter zwingend erforderlich, andere können weggelassen werden.

- Zwei notwendige Parameter legen einen global eindeutigen Bezeichner (`id`) und eine Versionsnummer (`version`) der Anwendung oder Erweiterung im `.xpi`-Archiv fest. Beide Werte müssen einer vorgegebenen Syntax folgen, um zu den Softwareverwaltungsmechanismen des *Toolkits* kompatibel zu sein.

- Eine ganze Reihe von Parametern dient der informativen Beschreibung der zu installierenden Software, der Benennung von verantwortlichen Software-Entwicklern und dem Verweis auf Webseiten der Software-Entwickler. Notwendig ist hierbei nur die Angabe eines menschenlesbaren Namens der installierten Anwendung oder Extension. In neueren Versionen des *Toolkits* lassen sich diese Informationen auch internationalisiert in verschiedenen Locales angeben.

- Sechs weitere Parameter dienen der Integration der zu installierenden Software in die Softwareverwaltung des *Toolkits*. Notwendig ist die Festlegung, ob das `.xpi`-Archiv eine Anwendung, eine Extension oder eine der anderen *Add-on*-Varianten enthält. Weiterhin kann eine URL parametrisiert werden, welche zur automatisierten Überprüfung der Verfügbarkeit von Software-Updates dient. Drei der Parameter enthalten Adressen zu Teilen des Chrome-Pakets, mittels derer Informations- und Eigenschaftsdialoge der installierten Software in die grafische Oberfläche der Softwareverwaltung integriert werden können. Es lässt sich auch angeben, ob die Software des `.xpi`-Archivs in der Softwareverwaltung für den Anwender unsichtbar bleiben soll.

- Drei Parameter erlauben die Festlegung der Voraussetzungen, unter denen die Software des `.xpi`-Archivs installiert werden kann. Notwendig ist bei *Add-ons* die Angabe der erweiterten Anwendung, zu der das *Add-on* kompatibel ist. Hierbei wird auf die Parameter `id` und `version` der Anwendung Bezug genommen, die Benennung mehrerer kompatibler Anwendungen und Versionen ist möglich. Zusätzlich ist es in neueren Versionen des *Toolkits* möglich, Abhängigkeiten zu anderen Extensions anzugeben. Bei plattformabhängiger Software können außerdem die von den im `.xpi`-Archiv enthaltenen Bibliotheken unterstützten Binärschnittstellen spezifiziert werden.

[17] Die Untergliederung der Installationsparameter in vier Funktionsbereiche stammt vom Autor dieser Arbeit, in [MDC07g] wird lediglich nach notwendigen und optionalen Parametern unterschieden.

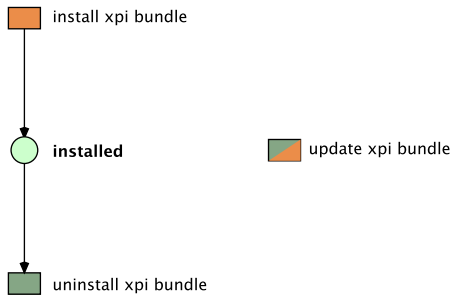



Abbildung 3.5: Das Lebenszyklus-Netz der XPInstall/Toolkit-Softwareverwaltung

[Vgl. MDC07g]

Wird das Mozilla-*Toolkit* zur Software-Verwaltung eingesetzt, so unterscheidet sich die Installation der Anwendung kaum von der von Extensions oder anderen *Add-ons*. Auch die Hauptanwendung kann als `.xpi`-Archiv verpackt und mit einem deklarativen Installationsmanifest versehen werden. Das Archiv muss lediglich ein weiteres Manifest enthalten, welches zunächst wieder einen eindeutigen und einen menschenlesbaren Namen, eine Versionsangabe und eine eindeutige Identifikation des Kompilats (*Build Identifier*) enthalten muss. Hinzu kommen Optionen zur Konfiguration der Softwareverwaltung des Toolkits und zur Spezifikation der benötigten Gecko-Version. Um die Auslieferung von Anwendungen und/oder Extensions bündeln zu können, darf ein `.xpi`-Archiv auch aus mehreren `.xpi`-Archiven oder Chrome-Paketen zusammengesetzt sein. [Vgl. MDC06b, MDC06d, MDC07i]

Die XPInstall-Komponente und die Softwareverwaltung des Mozilla-*Toolkits* schränken den Lebenszyklus der installierten Software nicht ein. Von der Installationsbeschreibung abgesehen kommt der Lebenszyklus ohne aktive Beteiligung der installierten Software aus. Daher fällt das Lebenszyklus-Netz in Abbildung 3.5 sehr einfach aus. Die XPInstall-Komponente beherrscht nur Installation und Deinstallation der Software. Daneben bietet das *Toolkit* noch die Update-Funktion, die allerdings lediglich durch eine Sequenz aus Deinstallation und Installation realisiert wird.[18] Im Lebenszyklus Netz ist die Transition daher zweifarbig dargestellt, sie wird mittels (nicht dargestellter) synchroner Kanäle mit den anderen Funktionen verschmolzen.

Ergänzt man die Deklarationen der synchronen Kanäle und berücksichtigt die Kopplung der Softwareverwaltung mit den Lebenzyklen der verschiedenen Architekturebenen der Mozilla-Softwarearchitektur, so erhält man ein Netz wie in Abbildung 3.6. Neben den bereits aus Abbildung 3.5 bekannten vier Elementen, die sich nun auf der linken Seite des Netzes wiederfinden, finden sich auf der rechten Seite mehrere farblich

[18] Obwohl die Update-Funktionalität des *Toolkits* in der Regel während der Laufzeit der Anwendung zum Einsatz kommt, besteht das Problem fort, dass die meisten Änderungen an installierten Paketen erst nach einem Neustart der Anwendung wirksam werden. Um die Zahl der Neustarts für den Anwender zu reduzieren, kann das *Toolkit* die Verfügbarkeit neuerer Versionen der installierten Pakete direkt beim Start des Systems prüfen, so dass die Datenbanken noch vor der Initialisierung der eigentlichen Anwendung aktualisiert werden können.

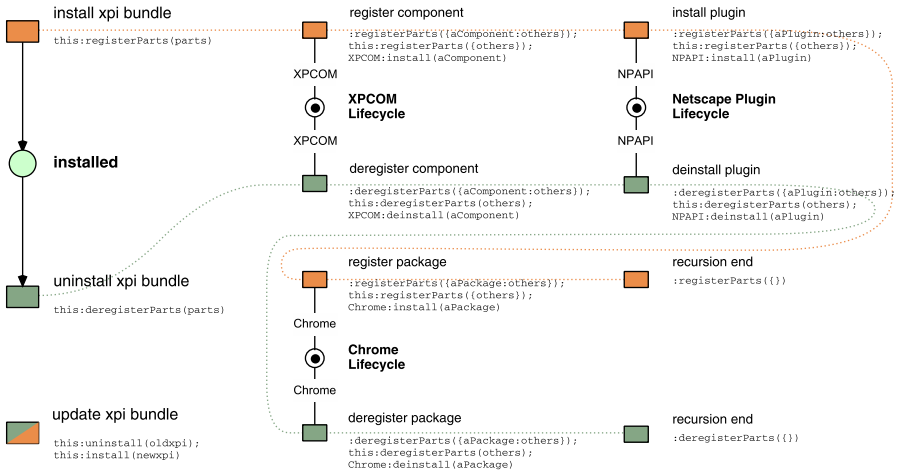



Abbildung 3.6: Das XPInstall/Toolkit-Netz mit Steuerung der anderen Lebenszyklen

dem Installations- oder Deinstallationsschritt zugeordnete Transitionen. Die Lebenszyklusnetze der verschiedenen Architekturebenen werden jeweils über Testkanten aus den farblosen Stellen referenziert.

Das Netz in Abbildung 3.6 reizt die Flexibilität synchroner Kanäle mit einer Technik aus, die z. B. in [Röl05] erläutert wird und der Programmierung von Rekursion in funktionalen Programmiersprachen entlehnt ist. Die Technik erlaubt es, beliebig viele Transitionen in einem synchronen Schaltvorgang zu verschmelzen. In diesem Fall wird damit die synchrone (De-)Installation aller Komponenten, Pakete und Plugins eines `.xpi`-Archivs modelliert.

Der Kanal `:registerParts` hat einen Parameter vom Typ Liste. Die Transition „install xpi bundle“ wird nur aktiviert, wenn ihr Downlink mit der Liste der Teile des `.xpi`-Archivs durch eine aktivierbare Transition mit passendem Uplink verbunden werden kann. Die drei kleinen orangefarbenen Transitionen zum Installieren von Komponenten, Plugins und Pakete verfügen sowohl über einen Up- als auch über einen Downlink dieses Kanals. Im Uplink verlangen sie eine nicht-leere Liste, deren erstes Element vom passenden Typ (Komponente, Plugin oder Paket) sein muss. Über den Downlink verlangen die Transitionen eine erneute Synchronisation, wobei aber das Kopfelement der Liste abgetrennt wurde. So kann eine Kette von synchronisierten Transitionen entstehen. Die einzelne kleine orangefarbene Transition rechts unten im Bild schließt die Kette ab, indem sie als Kanalparameter eine leere Liste erwartet.

Sind alle Transitionen der Kanalkette aktiviert (möglicherweise einzelne Transitionen auch mehrfach mit unterschiedlichen Bindungen), so enthält der Schaltschritt für jede Komponente, jedes Plugin und jedes Paket der Liste ein Transitionsvorkommen, dass dieses Element im jeweiligen Lebenszyklusnetz installiert bzw. registriert. Die gepunktete Linie in Abbildung 3.6 deutet die Verbindung der an der Kette mögli-

cherweise beteiligten Transitionen an. Es ist jedoch zu beachten, dass lediglich die erste und letzte Transition der Kette im Vorwege bekannt sind – welche der anderen drei Transitionen in welcher Reihenfolge und welcher Häufigkeit (eventuell auch gar nicht) an der Kette beteiligt sind, hängt von der Liste der Teile des `.xpi`-Archivs ab. Die Deinstallation eines `.xpi`-Archivs verwendet dieselbe Rekursionstechnik im Kanal `:deregisterParts`.

Mit der XPInstall-Komponente und der Softwareverwaltung des *Toolkits* steht somit eine anwendungs- und plattformübergreifende Plugin-Verwaltung zur Verfügung, welche kaum eine Unterscheidung zwischen Kernanwendung und Erweiterungen vornimmt. Allerdings ist diese Plugin-Verwaltung fest in eine Softwarearchitektur integriert, die mit XPCOM, Gecko/XUL/Chrome und XPInstall auf viele verschiedene Modularisierungstechniken aufsetzt und daher einen klar erkennbaren Architekturstil vermissen lässt.

### 3.1.3 Zusammenfassung

Die Familie der Anwendungen des Mozilla-Projekts und seines Vorläufers Netscape bieten auf verschiedenen Ebenen ihrer Softwarearchitektur Plugin-fähige Schnittstellen.

Die Netscape-Plugin-Programmierschnittstelle (*Netscape Plugin API, NPAPI*) ist speziell auf den Zweck hin ausgerichtet, multimediale Daten nahtlos in die grafische Oberfläche und interaktive Steuerung eines Webbrowsers integriert präsentieren zu können. Die in der portablen Sprache C spezifizierte Schnittstelle trennt genau zwei Systemkomponenten voneinander, den Webbrowser und das Plugin. Für beide Seiten gibt die Schnittstelle die jeweils bereitzustellende Funktionalität vor.

Netscape-Plugins werden als plattformspezifische dynamische Bibliotheken ausgeliefert. Für die korrekte Installation eines Plugins auf dem System des Anwenders ist in der Regel der Plugin-Hersteller zuständig, so dass diese außerhalb der Kontrolle des Webbrowsers erfolgt. Der Webbrowser sucht bei seinem Start an einer festgelegten Stelle im System nach installierten Plugins und entnimmt aus je nach Betriebssystem variierender Quelle deren Zuständigkeit für bestimmte Typen multimedialer Daten.

Der Lebenszyklus eines Plugins wird vom Webbrowser kontrolliert. Das Plugin wird bei Bedarf vom Browser aktiviert, je zu präsentierendem Datenobjekt eine Instanz des Plugins erzeugt und auch wieder vernichtet, sobald kein Bedarf mehr besteht. Während eine Instanz aktiv ist, kann das Plugin auf die in der NPAPI spezifizierten Dienstleistungen des Webbrowsers (z. B. zum Versand oder Empfang von Datenströmen über das Internet) zurückgreifen.

Im Gegensatz zur strengen, zweckgebundenen Trennung von Verantwortlichkeiten bei der NPAPI können *Extensions* in den neueren Versionen der Mozilla-Anwendungsfamilie beliebige Ergänzungen vornehmen, da die Softwarearchitektur auf mehreren Ebenen zwischen allgemeiner Plattform und darauf aufsetzender Spezialisierung trennt und daher keine Unterscheidung zwischen Kernanwendung und Erweiterung mehr benötigt.

Auf unterster Ebene regelt das in der portablen Sprache C++ spezifizierte Komponentenmodell XPCOM die Aufgaben einer allgemeinen Objektverwaltung und verwaltet einen globalen Namensraum für Komponenten ($\simeq$ Klassen) und deren Interfaces.

Der von XPCOM vorgegebene Lebenszyklus beschränkt sich auf das Registrieren der Komponenten in der zentralen Verwaltung sowie auf das Zählen von Objektinstanzen, wobei beide Aufgaben jeweils von den Komponenten selber implementiert werden müssen. Das Konzept des Dienstes wird in XPCOM durch das Entwurfsmuster einer Singleton-Komponente realisiert.

Die wesentliche Funktionalität der allen Anwendungen der Mozilla-Familie gemeinsamen Plattform wird durch einen Satz an XPCOM-Komponenten bereitgestellt und steht der auf dieser Plattform aufsetzenden anwendungsspezifischen Benutzungsoberfläche („Chrome“) uneingeschränkt zur Verfügung. Die Oberfläche einer Anwendung wird durch den Inhalt eines oder mehrerer Pakete definiert, welche XML-Dateien in Kombination mit JavaScript enthalten, die die Anordnung, Struktur und Ablaufkontrolle der Anwendung festlegen. Extensions stellen ebenso wie Anwendungen ein Paket dar und haben daher dieselben Einflussmöglichkeiten auf das Verhalten der Gesamtanwendung. Die Interpretation der aus allen Paketen zusammengeführten Oberflächenbeschreibung ist Teil der zentralen Funktionalität der Mozilla-Plattform.

Die Installation von Extensions – enthalten sie nun ein Chrome-Paket, eine XPCOM-Komponente oder eine Kombination mehrerer solcher Teile – ist ohne eine explizite Pflege verschiedener Datenbanken nicht vollständig. Eine einheitliche, portable Installationstechnik, ergänzt durch eine zentrale Softwareverwaltung, nimmt dem Anwender die mit der Installation verbundenen Aufgaben ab und verwaltet darüber hinaus einen globalen Namensraum für Anwendungen und Extensions. Da auch Versionen der installierten Software verwaltet werden, besteht die Möglichkeit einer automatischen Suche nach aktuelleren Versionen installierter Pakete.

Installationstechnik und Softwareverwaltung kontrollieren zwar die notwendigen Teilschritte einer Installation, definieren aber keinen für die installierten Komponenten oder Pakete wahrnehmbaren Lebenszyklus. Dies ist nicht notwendig, da die einzelnen Teile der Software – Chrome-Pakete, XPCOM-Komponenten oder Netscape-Plugins – über eigene Lebenszyklen verfügen, die in die Softwareverwaltung integriert werden.

## 3.2 Eclipse

Eclipse [Ecl] wird von seinen Entwicklern als *offene integrierte (Entwicklungsumgebungs-)Plattform für alles Mögliche, aber nichts Spezielles*, bezeichnet.[19] Eclipse nimmt in dreierlei Hinsicht eine Sonderrolle im Bereich der Plugin-basierten Anwendungen ein:

- Die Softwarearchitektur von Eclipse setzt geradezu radikal auf dem Begriff des Plugins auf. Abgesehen von einem kleinen Kernsystem zur Integration von Plugins wird jegliche Funktionalität der Eclipse-Plattform durch Plugins bereitgestellt.

- Die *Anwender* von Eclipse sind in der Regel gleichzeitig Software-*Entwickler*. Somit sind viele Anwender in der Lage, die Anwendung selber weiterentwickeln zu können und zu wollen.

[19] „Eclipse is an open (IDE) platform for anything, and for nothing in particular“ [Ecl, FAQ: What is Eclipse, http://wiki.eclipse.org/index.php/FAQ_What_is_Eclipse%3F].

- Die proprietäre Architektur des Eclipse-Kernsystems wurde vor einigen Jahren an eine als offener Standard definierte Softwarearchitektur angepasst, wobei die Implementierung des Kernsystems vollständig ersetzt wurde. Es stehen dadurch derzeit in Eclipse zwei orthogonale Konzepte zur Integration von Plugins parallel zur Verfügung.

Eclipse ist in der Sprache Java implementiert und nutzt als Unterbau des Plugin-Systems einige Abstraktionsmechanismen, die von der Sprache bereitgestellt werden. Diese Mechanismen werden in Unterabschnitt 3.2.1.2 kurz erläutert.

Eclipse entstand zunächst unter der Ägide von IBM, wurde dann aber unter eine offene Lizenz gestellt und wird nun von einer Open-Source-Community weiterentwickelt, die von der *Eclipse Foundation* mit diversen Mitgliedern aus der Industrie betreut wird. Insbesondere IBM pflegt mit der WebSphere-Entwicklungsumgebung ein um etliche Plugins erweitertes kommerzielles Produkt, das auf der Eclipse-Architektur aufsetzt. Daneben verfügt Eclipse als prominentes Open-Source-Projekt über eine breite Anwender- und Entwicklergemeinde, der auch Personen aus akademischen Umfeld angehören. Daraus ergibt sich eine Vielzahl von frei verfügbaren Veröffentlichungen und Diskussionen über die Systemarchitektur, die zusätzlich zur Dokumentation herangezogen werden können.

Obwohl viele Anwender Eclipse nur als Java-Entwicklungsumgebung wahrnehmen, umfasste es von Anfang an eine allgemeine, Plugin-basierte Software-Plattform (den *Eclipse Platform Core*) sowie eine allgemeine grafische Oberfläche für Entwicklungsumgebungen (die *Eclipse Platform UI* oder auch *Workbench*). Der entwicklungsumgebungsunabhängige Anteil beider Pakete wird inzwischen auch separat als *Eclipse Rich Client Platform (RCP)* vertrieben. In die allgemeine Plattform lassen sich unter anderem die *Java Development Tools (JDT)* integrieren, so dass eine Java-Entwicklungsumgebung zur Verfügung steht. Zahlreiche weitere Plugins stellen Entwicklungsunterstützung für Programmiersprachen wie C++ oder PHP sowie für spezielle Einsatzgebiete wie z. B. WebServices, den Embedded-Bereich oder die Erzeugung von Wirtschaftsberichten bereit. Insbesondere wird auch die Entwicklung von Plugins für Eclipse durch die *Plugin Development Environment (PDE)* unterstützt. [Vgl. RW06, S. 1ff.]

Beim Übergang von Eclipse Version 2.1 zur Version 3.0 haben sich die Entwickler entschieden, ihre eigene Implementierung eines Plugin-Systems durch eine auf dem Rahmenwerk der OSGi (***O**pen **S**ervice **G**ateway **I**nitiative*) aufbauende Implementierung namens *Equinox* zu ersetzen [siehe GHM$^+$05]. Das Rahmenwerk der kommerziell orientierten OSGi stellt mehrere in Java spezifizierte, dünn und flexibel gehaltene Abstraktionsschichten zur dienstorientierten Kapselung und Kopplung unabhängiger Anwendungen und Anwendungsteile bereit. Gewissermassen kann die Spezifikation als einfaches Komponentenmodell (siehe Unterabschnitt 2.2.2) angesehen werden, wobei anstelle des Begriffs Komponente der Begriff des *Bundles* zentral ist. Für einen ersten Überblick über die Konzepte und Möglichkeiten des OSGi-Rahmenwerkes bietet sich das Whitepaper [OSG05] der OSGi zur Lektüre an. Es existieren diverse Implementierungen der OSGi-Spezifikationen, darunter neben Equinox und weiteren Open-Source-Varianten (z.B. Knopflerfish und Apache Felix) auch etliche kommerzielle Implementierungen.

## 3.2.1 Plugins

In den folgenden Unterabschnitten wird die in Eclipse realisierte Softwarearchitektur vorgestellt. Zunächst bedarf es in Unterabschnitt 3.2.1.1 einer Eingrenzung des Plugin-Begriffs, wie er von Eclipse-Entwicklern verstanden wird. Das Eclipse-Plugin-System baut ebenso wie das OSGi-Rahmenwerk auf dem Java-Klassenlademechanismus auf, der in Unterabschnitt 3.2.1.2 vorgestellt wird. Die auf dem Mechanismus aufsetzende Plugin- bzw. Bundle-Verwaltung ist Gegenstand des Unterabschnitts 3.2.1.3. In der laufenden Plattform sind Plugins und Bundles einem Lebenszyklus unterworfen, der das Thema von Unterabschnitt 3.2.1.4 ist. Die Verbindung von Plugins zu einem Gesamtsystem erfolgt durch drei Relationen. Die erste Relation ist zentrales Element der klassischen Eclipse-Architektur und basiert auf dem Konzept der Erweiterung (*extension*), das in Unterabschnitt 3.2.1.5 erläutert wird. Orthogonal dazu werden im OSGi-Rahmenwerk Beziehungen zwischen Bundles durch das in Unterabschnitt 3.2.1.6 beschriebene Konzept des Dienstes (*service*) geknüpft. Die dritte Beziehung zwischen Plugins bzw. Bundles besteht aus Abhängigkeiten (*dependencies*), deren Zusammenspiel mit den anderen beiden Beziehungen das Thema des Unterabschnitts 3.2.1.7 ist. Die Entwickler von Eclipse wünschen sich neben der Kombinationsmöglichkeit unzähliger Einzelfunktionalitäten auch ein homogenes Erscheinungsbild der gesamten Anwendung. Unterabschnitt 3.2.1.8 gibt daher eine Sammlung von Prinzipien wieder, welche die Integration von Plugins in die den Nutzern und Entwicklern präsentierte Gesamtanwendung verbessern soll (ähnlich dem von Bauordnungen vorgegebenen Einfügungsgebot in der Stadtplanung).

### 3.2.1.1 Definitionsversuche

Jede Beschreibung der Eclipse-Softwarearchitektur benötigt den zentralen Begriff des *Plugins*. Trotzdem findet sich in diversen Veröffentlichungen keine einheitliche und präzise Definition dieses Begriffs. Gamma und Beck weichen in ihrem Buch „Contributing to Eclipse – Principles, Patterns and Plug-Ins“ (das den Begriff *Plugin* sogar im Untertitel trägt) auf die Analogie einer Steckdosenleiste aus, in welche Stecker verschiedener Form für verschiedene Geräte passen. Sie definieren den Plugin-Begriff nur über seine Tätigkeitsform, so dass Erweiterungen in einen Erweiterungspunkt (*extension point*) eingestöpselt (*plugged-in*) werden können. [GB03, S. 5]

Clayberg und Rubel verzichten in ihrem Buch „Building Commercial-Quality Plug-ins“ [CR06] gar komplett auf eine Definition des Begriffes. Stattdessen werden im dritten Kapitel des Buches technische Details des Dateisystems sowie der Erzeugung, Konfiguration und Verwaltung von Plugins erläutert.

Im erstmalig 2001 erstellten und zuletzt 2006 überarbeiteten Whitepaper „Eclipse Platform Technical Overview“ [RW06] wird ein Plugin als *die kleinste Einheit von Eclipse-Plattformfunktionalität, die separat entwickelt und ausgeliefert werden kann*[20], definiert. Dazu merken die Autoren an, dass zwar kleine Werkzeuge als einzelnes Plugin erstellt werden, die Funktionalität komplexerer Werkzeuge hingegen in der Regel

[20] „*A plugin is the smallest unit of Eclipse Platform function that can be developed and delivered separately.*“ [RW06, S. 4]

auf mehrere Plugins aufgeteilt wird. Auch in diesem Papier erschöpft sich damit die konzeptionelle Beschreibung von Plugins, alles Folgende sind technische Details.

Bolour definiert in seinem Eclipse-Community-Artikel „Notes on the Eclipse Plug-in Architecture“ [Bol03] ein Eclipse-Plugin als *Komponente, die einen bestimmten Diensttyp innerhalb der Eclipse-Workbench anbietet.*[21] Wieder folgen der knappen Definition ausschließlich technische Details.

In einem Zeitschriftenartikel, der die Architektur und Arbeitsweise des erneuerten, auf OSGi basierenden Plugin-Systems vorstellt, beschreibt Lippert den Begriff über seine softwarearchitektonischen Merkmale: Plugins besitzen „*einerseits explizit definierte Schnittstellen und Abhängigkeiten zu anderen Plug-ins und definieren andererseits Erweiterungspunkte, an denen ihre Funktionalität durch andere Plug-ins erweitert werden kann, sowie Erweiterungen, mit denen sie die Funktionalität anderer Plug-ins erweitern.*“ [Lip05b, S. 19]

#### 3.2.1.2 Dynamisches Laden von Klassen in Java

Ein Eclipse-Plugin besteht – technisch gesehen – ebenso wie ein OSGi-Bundle aus einer Menge von Dateien, meist Java-Klassen mit der Plugin-Funktionalität, aber auch anderen benötigten Ressourcen wie z. B. Bildern oder Hilfetexten. Sowohl das klassische Plugin-System von Eclipse als auch die Architektur der OSGi greifen bei dem Unterfangen, die Klassen und weiteren Ressourcen eines Plugins bzw. Bundles in das System zu integrieren, auf grundlegende Mechanismen der objektorientierten Programmiersprache Java und der Java-Laufzeitumgebung (*Java Virtual Machine, JVM*) zurück.

Die Sprache Java und ihre Laufzeitumgebung wurden von Anfang an auf die Ausführbarkeit und Transportierbarkeit von kompiliertem Code auf verschiedenen Betriebssystemen und Prozessorarchitekturen hin ausgelegt [vgl. LY99, Abschnitt 1.1]. Die Ausführbarkeit auf verschiedenen Hard- und Softwareplattformen wird gemeinhin als *Plattformunabhängigkeit* bezeichnet. Die Transportierbarkeit von Code (z. B. über ein Netzwerk) qualifiziert die Sprache nicht nur für Internet-Anwendungen, sondern beinhaltet auch die für Plugin-Systeme relevante Eigenschaft, Code nachträglich in ein laufendes System integrieren und auch wieder daraus entfernen zu können.

Wie bei dynamischen Bibliotheken (vgl. Unterabschnitt 3.1.1.2) muss der Code auch in der Java-Laufzeitumgebung geladen (*Loading*) und verknüpft (*Linking*) werden. Die maßgebliche Einheit ist dabei die *Klasse*, wobei der Begriff in diesem technischen Kontext auch *Interfaces* umfasst.[22] Die Aufgabe des Ladens übernimmt ein Klassenlader (*Classloader*), er muss das Kompilat aus einer ihm bekannten Quelle beziehen (oder anderweitig erzeugen) und für die JVM aufbereiten [vgl. GJSB05, Abschnitt 12.2]. Zum Aufbereiten gehört auch das Laden von Superklassen oder implementierten Interfaces der zu ladenden Klasse, da der Code einer Klasse ohne diese nicht vollständig wäre [vgl. LY99, Abschnitt 5.3.5]. Das Verknüpfen wird direkt von der JVM durchgeführt,

[21] „A plug-in in Eclipse is a component that provides a certain type of service within the context of the Eclipse workbench.“ [Bol03, Abschnitt 2]

[22] Somit entspricht der Begriff *Klasse* der technischen JVM-Ebene eher dem Begriff *Typ* auf hochsprachlicher Ebene, der wie z. B. in [GJSB05, Abschnitt 4] als Oberbegriff für die Sprachkonzepte *Class*, *Interface*, *Primitive Type* und *Array Type* verwendet wird.

es umfasst eine strukturelle Korrektheitsprüfung des Codes, das Vervollständigen der Klassenstruktur und das Auflösen symbolischer Referenzen zu weiteren Klassen, die z. B. als Feldtyp oder Methodenparameter verwendet werden [vgl. GJSB05, Abschnitt 12.3]. Das Auflösen symbolischer Referenzen darf aufgeschoben werden, bis die Referenzen benötigt werden [vgl. LY99, Abschnitt 5.4.3].

Die symbolischen Referenzen zwischen Klassen ergeben sich aus den im Quellcode verwendeten Klassennamen und werden beim Kompilieren in einer Tabelle abgelegt. Wenn eine Klasse eine andere in irgendeiner Form referenziert (z. B. als Superklasse, Variablentyp, Methodenparameter usw.), so spricht man gemeinhin von einer *Abhängigkeit* (*Dependency*) von der referenzierten Klasse, da die referenzierende Klasse ohne die referenzierte nicht verwendbar ist. Beim Auflösen der Abhängigkeiten können andere Versionen der beim Kompilieren referenzierten Klasse verwendet werden, solange alle referenzierten Namen, Felder, Methoden und Parameterlisten unverändert sind.

Neben der technischen Aufgabe des Ladens haben Classloader in Java die Funktion der Definition von Namensräumen und Sichtbarkeitsbeziehungen für Klassen. In einer JVM existieren üblicherweise mehrere Classloader, die sogar auf unterschiedlichen Implementierungen basieren dürfen – so kann eine Anwendung speziell auf ihre Bedürfnisse zugeschnittene Classloader definieren. Es ist möglich, dass mehrere Classloader abweichenden Code für eine Klasse desselben Namens kennen und laden. Somit ist der *definierende Classloader* ein identifikationsrelevanter Bestandteil einer Klasse in der Laufzeitumgebung, jeder Classloader definiert einen eigenen Namensraum. [vgl. LY99, Abschnitt 5.3].

Die Architektur der JVM verlangt, dass Classloader in einer Delegationsbeziehung stehen – jeder Classloader muss, wenn der Code eine Klasse angefordert wird, die Aufgabe zuerst an seinen übergeordneten (*Parent*) Classloader delegieren. Nur, wenn die Kette der übergeordneten Classloader die Klasse nicht definieren kann, darf ein Classloader sich selber an der Aufgabe versuchen. Zudem wird beim Auflösen symbolischer Referenzen einer Klasse zuerst immer derjenige Classloader herangezogen, der diese Klasse definiert hat. Diese Unterscheidung zwischen *initiierendem* Classloader und *definierendem* Classloader stellt sicher, dass alle Klassen, die einem Classloader direkt oder über seine übergeordneten Classloader zugänglich sind, stets vom selben Classloader definiert werden. Durch das Delegationsprinzip entstehen Sichtbarkeitsbereiche – Klassen aus übergeordneten Classloadern verdecken Definitionen derselben Klasse in untergeordneten Classloadern, und Klassen aus untergeordneten oder anderen nicht in der Delegationskette enthaltenen Classloadern können aus dem Namensraum eines initiierenden Classloaders heraus nicht referenziert werden. [Vgl. LB98, Abschnitt 2.2]

Das Entfernen einer Klassendefinitionen aus dem System ist nur möglich, indem der definierende Classloader entfernt wird – andernfalls entstünden Inkonsistenzen, falls eine Klasse im selben Namensraum wiederholt geladen und initialisiert würde. Das Entfernen eines Classloaders kann wiederum nicht explizit angefordert werden, da sonst bereits aufgelöste Referenzen zu Klassen im Namensraum des gelöschten Classloaders fehlerhaft würden. Ein Classloader wird automatisch aus dem System entfernt, wenn kein Objekt im System mehr eine Referenz auf den Classloader oder eine der von ihm definierten Klassen hält – gleiches muss für alle untergeordneten Classloader gelten, sofern solche existieren. [Vgl. GJSB05, Abschnitt 12.7]

Für die Implementierung von Plugin-Systemen steht mit der Möglichkeit des dynamischen Nachladens von Code sowie der Definition von Namensräumen und Sichtbarkeitsbereichen eine relativ mächtige und flexible Grundlage bereit. Die Abhängigkeiten zwischen Klassen sind allerdings in Form der symbolischen Referenzen fest in den Kompilaten kodiert. Dies schränkt die Flexibilität des dynamischen Integrierens von Code ein, da beim Auflösen der Referenzen nur die gemäß des statischen Typsystems der Sprache Java erlaubten Abweichungen toleriert werden. Die Implementierung eines Plugin-Systems erfordert zudem die Implementierung eines oder mehrerer individueller Classloader, die in der Lage sind, während der Laufzeit weitere Quellen für die Kompilate zu installierender Plugins aufzunehmen und die definierten Klassen konsistent in die Classloader-Hierarchie einzubinden.

Sowohl das klassische Eclipse-Plugin-System als auch das OSGi-Rahmenwerk verwenden eine spezialisierte Classloader-Implementierung, die im Wesentlichen jedem Plugin bzw. Bundle einen eigenen Classloader zuordnet. Die Eigenschaften dieser Classloader-Architektur werden im Unterabschnitt 3.2.1.7 kurz umrissen.

#### 3.2.1.3 Verwaltung und Programmierschnittstelle

Damit das Eclipse-Plugin-System die Klassen, Ressourcen und Plugin-übergreifenden Abhängigkeiten eines Plugins verwalten kann, müssen in der Menge der Dateien des Plugins zwei oder mehr Text- oder XML-basierte *Manifest*-Dateien in einem festgelegten Format mit festgelegten Namen enthalten sein, welche die im Plugin enthaltene Funktionalität und die Voraussetzungen ihrer Einsetzbarkeit maschinenlesbar beschreiben. Somit sind die Eigenschaften des Plugins der Eclipse-Laufzeitumgebung auch ohne das Instantiieren von Java-Klassen zugänglich.

Manifeste deklarieren unter anderem die Abhängigkeiten von anderen Plugins, sowie die bereitgestellten Erweiterungspunkte und Erweiterungen. Seit der Umstellung auf das OSGi-Rahmenwerk können zusätzlich bereitgestellte und genutzte Dienste deklariert werden. Die beiden notwendigen Manifeste (`META-INF/MANIFEST.MF` und `plugin.xml`) unterscheiden sich hinsichtlich ihres Ursprungs und ihrer Syntax. Während `MANIFEST.MF` dem JAR-Mechanismus der Java-Laufzeitumgebung entstammt, Schlüssel-Wert-Paare im Klartext enthält und auch für OSGi-spezifische Attribute verwendet wird, ist die im XML-Format abgefasste `plugin.xml` der Kern der Beschreibung klassischer Eclipse-Plugins. Im Zuge der Umstellung auf das OSGi-Rahmenwerk haben sich einige Aufgaben wie z. B. die Identifikation des Plugins durch Namen und Versionsnummer oder die Deklaration der Abhängigkeiten zwischen den Manifesten verschoben.

Der Punkt, an dem die Funktionalität des Plugins mit der der restlichen Plattform integriert werden kann, ist eine im Plugin implementierte Unterklasse von `org.eclipse.core.runtime.Plugin`. Diese Klasse stellt Methoden zur Benachrichtigung über Start und Beendigung des Plugins, zum Auffinden und Einlesen eigener Plugin-Ressourcen sowie zum Konfigurieren der Plugin-Parameter bereit. Einfache Plugins, die auf diese Dienste nicht angewiesen sind und denen die Deklarationen der Manifeste zur Integration ins System ausreichen, können auf die Plugin-spezifische Implementierung dieser Unterklasse verzichten. Benötigt ein Eclipse-Plugin Informationen über andere

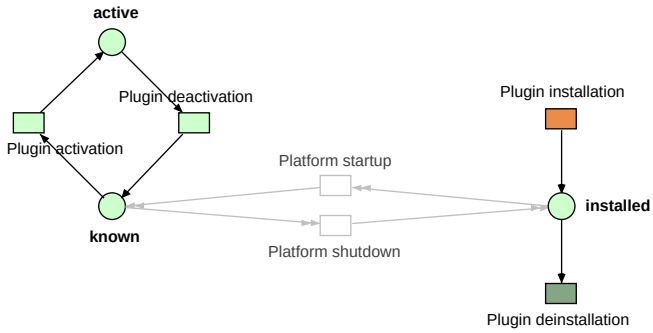



Abbildung 3.7: Der Lebenszyklus eines klassischen Eclipse-Plugins als Petrinetz

Plugins, so kann es jederzeit direkt auf die global bekannte Plugin-Verwaltungsinstanz (`PluginRegistry`) zugreifen.

Naturgemäß hat die Umstellung der Eclipse-Plugin-Architektur auf das OSGi-Rahmenwerk zu massiven Veränderungen der `Plugin`-Klasse geführt, bei der Kompetenzen an andere Klassen delegiert wurden bis hin zur vollständigen Ablösung durch eine Implementierung des Interfaces `BundleActivator`. Ein `BundleActivator` bekommt beim Starten des Plugins eine Referenz auf den `BundleContext` übergeben, welcher nach Maßgabe der OSGi den ausschließlichen Anknüpfungspunkt an das Rahmenwerk, andere Bundles und verfügbare Dienste darstellt. Die alten Methoden und Schnittstellen stehen aber zwecks Wahrung der Kompatibilität weiterhin zur Verfügung.

#### 3.2.1.4 Lebenszyklen

Der Lebenszyklus eines klassischen Eclipse-Plugins ist recht einfach. Er wird durch die technischen Gegebenheiten definiert und lässt sich wie in Abbildung 3.7 durch ein Petrinetz entsprechend der in Unterabschnitt 4.4.2.1 diskutierten Technik darstellen. Das Plugin wird *installiert*, indem seine Dateien an passender Stelle in die Verzeichnishierarchie der Eclipse-Installation hineinkopiert werden. Beim nächsten Start der Eclipse-Plattform werden alle installierten Plugins automatisch gefunden und ihre Manifeste gelesen. Zur Laufzeit wird das Plugin geladen und *aktiviert*, sobald seine Funktionalität benötigt wird. Falls beim Aktivieren des Plugins Fehler auftreten, wird das Plugin sofort wieder *deaktiviert*, andernfalls passiert dies erst beim Beenden der Eclipse-Plattform oder auf explizite Anforderung hin. Ein dynamisches Deaktivieren und wieder Aktivieren einzelner Plugins während der Laufzeit ist zwar möglich, wurde aber ursprünglich nicht thematisiert. Eine Installation oder gar Deinstallation von Plugins in der laufenden Plattform war vor der Umstellung auf das Rahmenwerk der OSGi nicht möglich.

Da das Einlesen aller installierten Plugins sowie das Herunterfahren der Plattform sämtliche Plugins auf einmal betrifft, sind die entsprechenden Transitionen in Abbildung 3.7 mit Doppelpfeil-Kanten (in Anlehnung an die Darstellung flexibler Kanten in

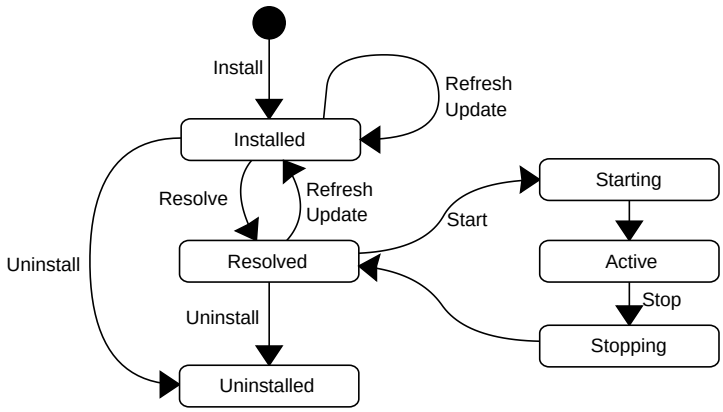



Abbildung 3.8: Der Lebenszyklus eines OSGi-Bundles [nach OSG06a, S. 80]

Referenznetzen) angebunden. Die blassgraue Farbe deutet an, dass die entsprechenden Übergänge nicht zum eigentlichen Lebenszyklus eines einzelnen Plugins gehören, da sie nicht explizit angestoßen werden können.

Das Aufschieben der Plugin-Aktivierung bis zum Zeitpunkt des eigentlichen Bedarfs bezeichnen Gamma und Beck als *verzögertes Laden* [*Lazy Loading Rule*, GB03, S. 16]. Dieser Begriff reflektiert die Tatsache, dass der Code des Plugins nicht schon beim Start der Plattform, sondern erst bei der Aktivierung des Plugins über einen Java-Classloader *geladen*, also der Laufzeitumgebung zugänglich gemacht wird. Der Umsetzung des *verzögerten* Ladens wird sowohl in der Eclipse-Plattformimplementierung, der Implementierung vieler Plugins als auch in fast allen Leitfäden zur Eclipse-Plugin-Programmierung eine hohe Aufmerksamkeit zuteil. Dies mag darin begründet liegen, dass viele Eclipse-Anwender gleichzeitig Eclipse-Entwickler sind und eine schnell startende Plattform benötigen, um ihre frisch implementierten Plugins bequem im Gesamtkontext testen zu können.

Für das reine Plugin-Konzept mit dem Ziel der Erweiterbarkeit des Systems ist das verzögerte Laden allerdings nur von geringer Relevanz. Zudem kann diese Eigenschaft in der Eclipse-Plattform nur durch kooperative Implementierung aller erweiterbaren Plugins erreicht werden (siehe Unterabschnitt 3.2.1.5).

Das Eclipse-System teilt jedem Plugin einen eigenen Classloader zu, der einzig dem Plattform-Classloader untergeordnet ist. Die Trennung der Plugin-Classloader sorgt für getrennte Namensräume der Plugins, so dass keine Namenskollisionen zwischen unabhängigen Plugins zu befürchten sind. Um dennoch symbolische Referenzen zwischen Klassen voneinander abhängiger Plugins auflösen zu können, beherrscht die Eclipse-*PluginClassLoader*-Implementierung die Delegation des Definierens von benötigten Klassen an alle `PluginClassLoader` der im Manifest als Abhängigkeit deklarierten Plugins.

Mit der Umstellung der Eclipse-Laufzeitumgebung auf das OSGi-Rahmenwerk wurde der vorhandene Plugin-Lebenszyklus durch den Lebenszyklus von Bundles im Sinne der Spezifikationen der OSGi abgelöst. Dem Bundle-Lebenszyklus hat die OSGi ein eigenes Kapitel gewidmet [OSG06a, Kapitel 4], dem auch das in Abbildung 3.8 dargestellte Zustandsdiagramm entstammt. Alle Zustandsübergänge werden durch die Implementierung des OSGi-Rahmenwerks verwaltet, es finden also auch Operationen wie das Installieren oder Deinstallieren von Bundles zur Laufzeit statt.[23] Die meisten der im Zustandsdiagramm dargestellten Übergänge werden explizit (z. B. vom Administrator oder durch andere Bundles) angefordert. Ausnahmen sind zum einen die unbenannten Übergänge beim Starten und Stoppen, zum anderen die Übergänge *Resolve* und *Refresh*, die vom OSGi-Rahmenwerk bei Bedarf automatisch angestoßen werden können.

Bei der *Installation* eines Bundles werden die zugehörigen Dateien von einer externen Quelle (z. B. Dateisystem oder Internet) aus eingelesen und das Bundle anhand seiner Manifeste auf Zulässigkeit (*Validity*) geprüft. Um das Bundle im System nutzen zu können, müssen seine in den Manifesten deklarierten Abhängigkeiten (siehe Unterabschnitt 3.2.1.7) aufgelöst (*resolved*) werden, indem passende andere Bundles gesucht und verknüpft werden. Erst nach erfolgreicher Abhängigkeitsauflösung erzeugt das OSGi-Laufzeitsystem einen passend konfigurierten Classloader für das Bundle [vgl. OSG06a, Abschnitt 3.8]. Anschließend kann das Bundle *gestartet* werden, wobei das Starten als zeitbehafteter Vorgang durch zwei Zustandsübergänge modelliert wird. Das Starten von OSGi-Bundles entspricht dem Aktivieren klassischer Eclipse-Plugins. Das *Stoppen* von Bundles ist wiederum ein zweischrittiger Übergang. Das OSGi-Rahmenwerk übernimmt dabei das Auflösen aller Dienstnutzungsbeziehungen, so dass das stoppende Bundle sich nur um die Freigabe anderweitig reservierter Ressourcen kümmern muss. Ein nicht-aktiviertes Bundle kann *deinstalliert* werden. Das Rahmenwerk achtet dabei darauf, dass Klassen und Ressourcen des deinstallierten Bundles noch so lange zur Verfügung stehen, bis alle nutzenden Bundles neu aufgelöst wurden. Mittels einer Aktualisierung (*Update*) kann ein nicht aktiviertes Bundle durch eine andere Version seiner selbst abgelöst werden. Die Auswirkungen der Aktualisierung sind aber erst abgeschlossen, wenn alle abhängigen bzw. Dienste-nutzenden Bundles aufgefrischt (*refreshed*) wurden. Eine Aktualisierung bzw. Auffrischung von Bundles führt im Wesentlichen dazu, dass die Abhängigkeiten der Bundles neu aufgelöst werden müssen. Da die Rekonfiguration der Delegationsbeziehung von Classloadern der Java-Architektur zuwiderliefe, bedeutet dies, dass der gesamte Java-Code eines aufgefrischten Bundles neu geladen werden muss. [Vgl. OSG06a, Abschnitte 4.3.3–4.3.9]

Abbildung 3.9 stellt den von der OSGi spezifizierten Lebenszyklus als Petrinetz dar. Die grafische Anordnung der Elemente ist hier so gewählt, dass sie dem Lebenszyklusnetz klassischer Eclipse-Plugins (vgl. Abbildung 3.7) möglichst ähnlich ist. Auf den ersten Blick stellt der OSGi-Lebenszyklus eine Verfeinerung des Eclipse-Lebenszyklus' dar: Der Zyklus zum Starten und Stoppen ist bei der OSGi feiner modelliert, und die

[23] Mit der Betrachtung von Installation und Deinstallation als vom Rahmenwerk vorgenommene Operationen geht einher, dass eine OSGi-Implementierung über einen persistenten Speicher verfügen muss, in dem installierte Bundles erhalten bleiben, auch wenn das System insgesamt heruntergefahren und später wieder gestartet wird.

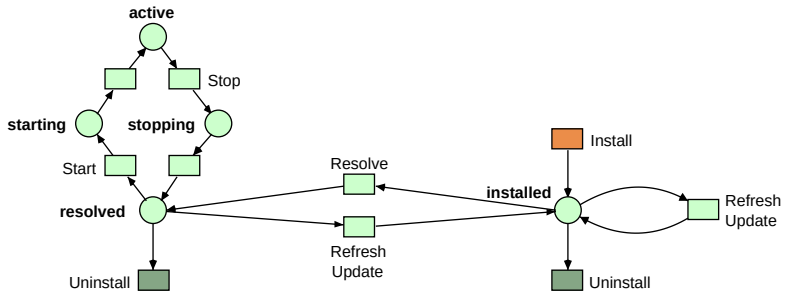



Abbildung 3.9: Der Lebenszyklus eines OSGi-Bundles als Petrinetz

Transitionen zum Aktualisieren bzw. Auffrischen fehlen im Eclipse-Konzept ebenso wie der explizite Zustand für deinstallierte Bundles.

Ein wesentlicher Unterschied zwischen beiden Lebenszyklen ist allerdings an diesem Netz kaum zu erkennen. Der von der OSGi definierte Lebenszyklus integriert sämtliche Zustände, alle Übergänge inklusive Installation und Deinstallation werden vollständig von der OSGi-Implementierung verwaltet. Dagegen besteht das Petrinetz des klassischen Eclipse-Lebenszyklus aus zwei indirekt (durch Zustandsübergänge der Plattform) verbundenen Teilen, von denen nur der linke Teil tatsächlich durch die Plattform verwaltet wird.

Um diesen Unterschied zu verdeutlichen, enthält das Netz in Abbildung 3.10 weitere, blassgrau dargestellte Übergänge, die wie im Eclipse-Lebenszyklusnetz nicht direkt vom verwalteten Bundle abhängen. Das Persistieren der installierten Bundles wird durch die mit „Platform restart“ benannte Transition verdeutlicht. Sie umfasst die beiden im Eclipse-Netz einzeln ausgewiesenen Übergänge „Platform shutdown“ und „Platform startup“, da – anders als bei klassischen Eclipse-Plugins – bei nicht laufender Plattform keinerlei Änderungen am Lebenszyklus von Bundles möglich sind. Der Übergang bringt sämtliche Bundles vorübergehend in den Zustand „installed“, gemäß der OSGi-Spezifikation werden aber beim nächsten Hochfahren der Plattform alle Bundles, die vor dem Herunterfahren im Zustand „active“ waren, sofort wieder gestartet.

Die Inhibitor-Kanten an der Transition „Platform restart“ stellen sicher, dass sämtliche aktiven Bundles vor dem Herunterfahren der Plattform gestoppt werden. Da die Bundles beim Wiederhochfahren sämtliche Zustände und Übergänge bis zum aktivierten Zustand erneut durchlaufen müssen, ist im Netz keine Kante eingezeichnet, welche die Zustandsmarken unmittelbar in den Zustand „active“ legt. Das Herunter- und Hochfahren der Plattform sind also keine atomaren Übergänge, sondern mehrschrittige Prozesse, die jedes Bundle entsprechend seines individuellen Zustandes vor dem Herunterfahren unterschiedlich behandeln.

Der neue Zustand *noch verwendet* (*still referred*) repräsentiert die Fähigkeit des OSGi-Rahmenwerkes, bei Deinstallation oder Update von Bundles die Instanz der alten Version aufzubewahren, bis keine Referenzen mehr von anderen Bundles aus

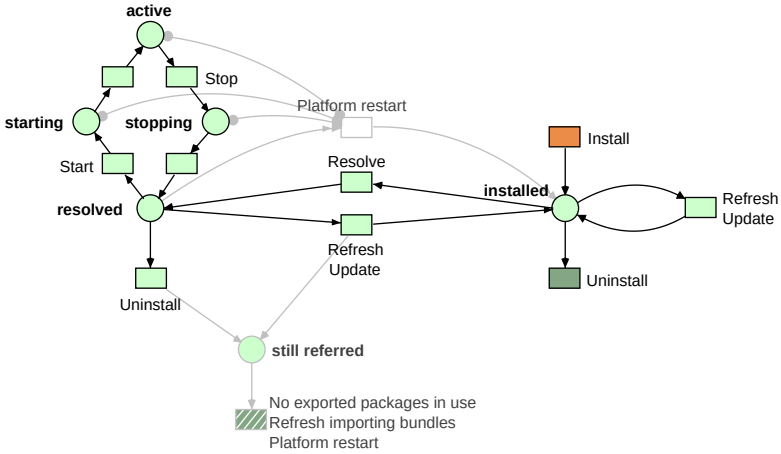



Abbildung 3.10: Vervollständigte Version des OSGi-Lebenszyklus als Petrinetz

bestehen. Falls ein bereits aufgelöstes Bundle aktualisiert wird, geht die alte Version in den Zustand *noch verwendet*, während gleichzeitig die neue Version im Zustand *installiert* beginnt.

An dem Bundle-Lebenszyklus sind im OSGi-Rahmenwerk weitere Lebenszyklen gekoppelt. Im Unterabschnitt 3.2.1.6 werden Lebenszyklen zur Konfiguration von Diensten vorgestellt werden.

Die Detailliertheit der OSGi-Spezifikationen ergibt sich aus dem schwerpunktmäßigen Einsatz des OSGi-Rahmenwerkes in weitgehend unterbrechungsfrei laufenden, teilweise auch fernwartbaren Systemen. Die damit einhergehende Anforderung von dynamischem Installieren, Aktualisieren und Deinstallieren erfordert eine tiefgehende Spezifikation der einzelnen Zustandsübergänge, damit die Veränderung des gesamten Systems wohldefiniert vonstatten gehen kann. Diese Fähigkeit des OSGi-Rahmenwerkes war der Anlass zum Abkehr der Eclipse-Entwickler von ihrem eigenen Plugin-System, da sie das Installieren und Austauschen von Plugins zur Laufzeit in Eclipse integrieren wollten.

#### 3.2.1.5 Erweiterungen (Extensions)

Jedes Eclipse-Plugin hat die Möglichkeit, einen oder mehrere *Erweiterungspunkte* (*Extension Point*) zu deklarieren. Ein Erweiterungspunkt spezifiziert eine Schnittstelle, an der der Plugin-Entwickler Erweiterungsmöglichkeiten durch andere Plugins vorgesehen hat. Plugins, die sich in einen Erweiterungspunkt einklinken wollen, deklarieren eine passende *Erweiterung* (*Extension*). Die Beziehung zwischen Erweiterung und Erweiterungspunkt ist eine $n$ : 1-Beziehung, d.h. jede Erweiterung ist fest an einen Erweiterungspunkt gebunden, aber ein Erweiterungspunkt kann beliebig oft erweitert werden. Die Zahl der Erweiterungen und Erweiterungspunkte, die ein Plugin dekla-

riert und/oder nutzt, ist nicht begrenzt. Es ist auch erlaubt, dass ein Plugin sich selbst erweitert, d.h. passende Erweiterungen zu einem selbstdefinierten Erweiterungspunkt anbietet.

Erweiterungspunkte und Erweiterungen werden im `plugin.xml`-Manifest spezifiziert, wobei die Deklaration eines Erweiterungspunkts ein XML-Schema für die Deklaration von passenden Erweiterungen vorgibt. Zusätzlich werden im erweiterbaren Plugin meist ein oder mehr Java-Interfaces bereitgestellt, die von Klassen des erweiternden Plugins implementiert werden sollen. Instanzen dieser Klassen werden erzeugt, um im Falle der Nutzung der erweiterten Funktionalität die Kontrolle vom Erweiterungspunkt an die Erweiterung zu übergeben (*Callback*). Besteht kein Bedarf einer solchen Kontrollflussübergabe, können Erweiterungen auch gänzlich ohne Java-Code auskommen. [Vgl. Bol03, Abschnitt 2.3]

Die Eclipse-Laufzeitumgebung sammelt alle deklarierten Erweiterungspunkte und Erweiterungen beim Systemstart und bietet sie den Plugins in Form eines Verzeichnisses (*Registry*) an. Die Entscheidung, welche (und wie viele) der von Plugins deklarierten Erweiterungen ein Erweiterungspunkt tatsächlich verwendet, obliegt dem erweiterbaren Plugin. Entsprechend der *Lazy Loading Rule* können solche Entscheidungen anhand der deklarierten Informationen getroffen werden, ohne die erweiternden Plugins aktivieren zu müssen. Das verzögerte Laden von erweiternden Plugins ist allerdings nur möglich, wenn die Implementierung des Erweiterungspunkts explizit darauf ausgelegt ist, die Aktivierung von Erweiterungen so lange wie möglich aufzuschieben, indem z. B. zunächst Platzhalterobjekte erzeugt werden. [Vgl. CR06, Kapitel 17]

Die Deklaration von Erweiterungen und Erweiterungspunkten wird durch die Eclipse-Plugin-Entwicklungsumgebung (*Plugin Development Environment, PDE*) massiv unterstützt, so dass Entwickler in der Regel die Manifeste nicht im XML-Quelltext erstellen und pflegen müssen. Zudem achtet die PDE darauf, dass Bezeichnungen, Typen und Abhängigkeiten im Quellcode eines Plugins mit den im Manifest deklarierten übereinstimmen. Diese Werkzeugunterstützung hält den Aufwand, der durch die mit dem deklarativen Erweiterungsmechanismus verbundenen redundanten Datenhaltung zwangsläufig entsteht, in für die meisten Entwickler vertretbaren Grenzen.

Der Erweiterungsmechanismus in seiner ursprünglichen Form ermöglicht die Deklaration einer statischen Anwendungsarchitektur. Da das Verzeichnis mit Erweiterungsinformationen bei Systemstart erstellt wird, können die Plugins davon ausgehen, dass sich die Erweiterungsbeziehungen während der Laufzeit nicht verändern. Dieser Umstand hält den Implementierungsaufwand niedrig, da Plugins nur bei ihrer Aktivierung einmalig das Verzeichnis abfragen müssen [vgl. RW06, S. 5]. Mit der Umstellung auf das OSGi-Rahmenwerk ist es möglich geworden, erweiternde Plugins zur Laufzeit zu installieren und zu deinstallieren, wodurch sich die Menge der Erweiterungen eines Erweiterungspunkts dynamisch verändern kann. Die zusätzlichen Pflichten, die auf einen Plugin-Entwickler zukommen, wenn sein Code dynamisch deinstallierbar sein soll und seine Erweiterungspunkte mit einer sich dynamisch verändernden Menge an Erweiterungen zurecht kommen müssen, umreißen Lippert und Kolb in [LK05]. Allerdings werden noch längst nicht alle existierenden Eclipse-Plugins den erhöhten Anforderungen gerecht.

#### 3.2.1.6 Dienste (Services)

Mit der Umstellung der Eclipse-Plugin-Architektur auf das OSGi-Rahmenwerk steht allen Plugins bzw. Bundles die von der OSGi in [OSG06a, Kapitel 5, S. 101–115] in Version 1.3 spezifizierte Dienstleistungsschicht (*Service Layer*) zur Verfügung.

Da das ursprüngliche Einsatzgebiet von OSGi-Implementierungen im Wesentlichen unterbrechungsfrei laufende, Ressourcen-arme Embedded-Geräte betraf, unterscheiden sich die Zielsetzungen der Dienstleistungsschicht etwas von denen der Eclipse-Erweiterungsarchitektur. Die Spezifikation nennt als vorrangige Ziele der Dienstleistungsschicht, es zu ermöglichen, dass zunächst gegenseitig unbekannte Bundles kollaborieren und dynamische Änderungen der Umgebung unmittelbar verarbeitet werden können. Dies soll auf eine sichere und inspizierbare Weise ermöglicht werden. Zudem sollen Dienste und deren Implementierungen versionierbar und die Persistenz von Dienstidentitäten auch über Plattform-Neustarts hinweg gegeben sein [vgl. OSG06a, S. 101].

Ein Dienst muss *spezifiziert*, *angeboten* und *genutzt* werden, diese drei Aufgaben können von unterschiedlichen Bundles wahrgenommen werden. Die Verbindung zwischen Anbietern und Nutzern wird durch die drei Schritte Veröffentlichen, Auffinden und Ankoppeln (*publish, find and bind*) realisiert, welche über eine zentrale, von der OSGi-Implementierung bereitzustellende Verwaltung (*Service Registry*) laufen. Jeder Dienst kann nur einmal spezifiziert werden,[24] aber die Zahl der Dienstanbieter und -nutzer ist keinen Beschränkungen unterworfen. Ein Dienstanbieter implementiert das Dienst-Interface und veröffentlicht sein implementierendes Objekt bei der Dienstverwaltung. Ein Dienstnutzer erfragt passende Dienstanbieter bei der Verwaltung, wobei die Anfrage durch einen in Java implementierten Filter verfeinert werden kann. Hat er einen Dienstanbieter ausgewählt, kann er sich an dessen Dienstobjekt binden. Ein Dienstnutzer muss jederzeit damit rechnen, dass sich die Menge der angebotenen Dienste verändert. Solange ein Dienst allerdings gebunden ist, besteht eine Abhängigkeit zwischen den Lebenszyklen der beteiligten Bundles.

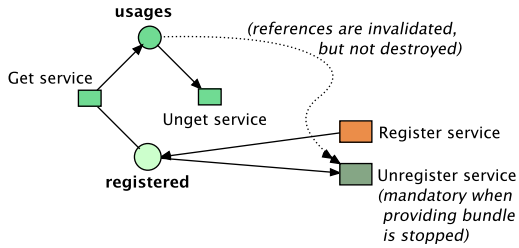



Abbildung 3.11: Lebenszyklus von OSGi-Diensten

Dienste sind im OSGi-Rahmenwerk ebenfalls einem Lebenszyklus unterworfen, der zudem an den in Unterabschnitt 3.2.1.4 vorgestellten Bundle-Lebenszyklus gekoppelt

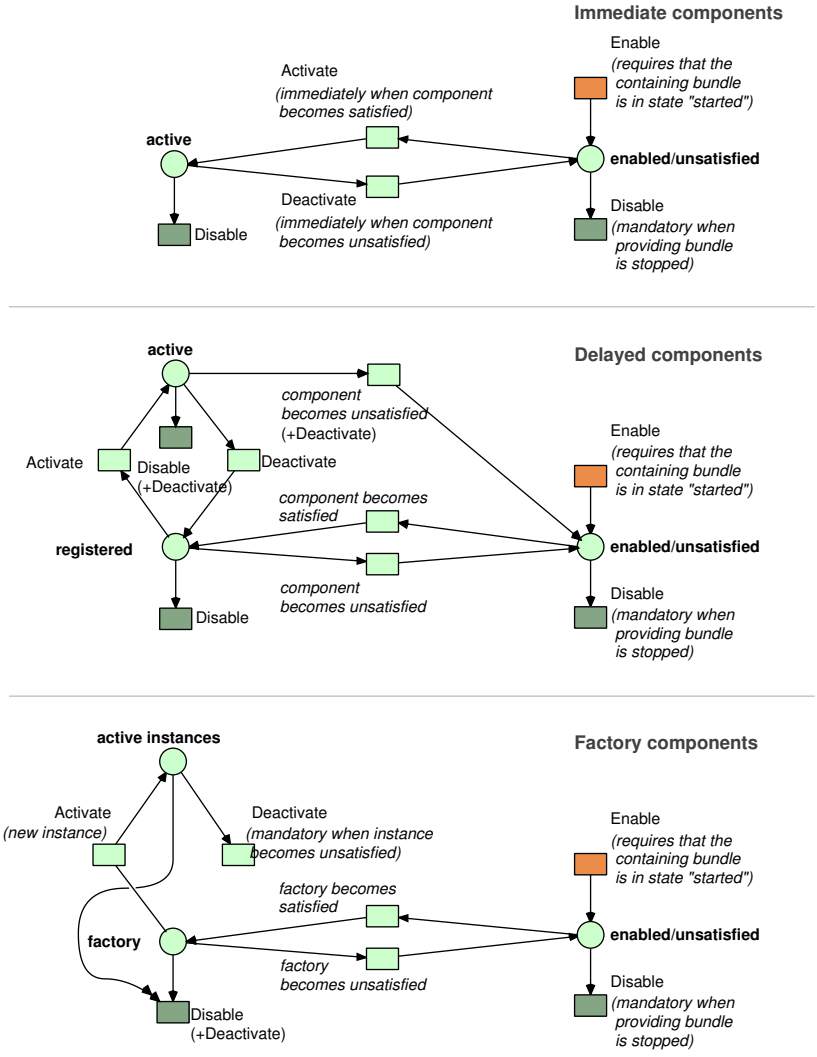
[24] Die Existenz inkompatibler Spezifikationen eines Dienstes innerhalb einer Laufzeitumgebung ist vorgesehen. Die Eindeutigkeit der Dienstspezifikation wird dann durch die Dienstverwaltung sichergestellt, welche die Sichtbarkeitsbereiche der Bundles so berücksichtigen muss, dass kein Bundle mehr als eine Spezifikationen desselben Dienstes zu sehen bekommt. [Vgl. OSG06a, S. 114]

ist. Abbildung 3.11 stellt die in [OSG06a, Kapitel 5] eingeflochtene Beschreibung des Lebenszyklus als Petrinetz dar. Ein dienstanbietendes Bundle darf in den drei Lebenszykluszuständen „starting“, „active“ oder „stopping“ (vgl. Abbildung 3.10) beliebig viele eigene Dienste registrieren und de-registrieren. Das Rahmenwerk stellt zudem sicher, dass alle Dienste eines gestoppten Bundles wieder de-registriert werden. Registrierte Dienste können von mehreren Bundles gebunden („Get Service“), genutzt und wieder freigegeben („Unget Service“) werden. Ein dienstnutzendes Bundle begibt sich für die Dauer der Bindung in eine dynamische Abhängigkeit zum Lebenszyklus des Dienstes. Beim De-Registrieren eines Dienstes werden alle Bindungen aufgelöst und die nutzenden Bundles darüber informiert. Die tatsächliche Freigabe aller aus der Dienstnutzung hervorgegangenen Objekte muss dann im nutzenden Bundle explizit programmiert werden, da das Rahmenwerk hierauf keine Einflussmöglichkeit hat.

Im Vergleich zu Eclipse-Erweiterungen fällt die deutliche Betonung der Dynamik des Systems auf, hingegen fehlt ein der *Lazy Loading Rule* entsprechendes Ziel. Dies schlägt sich darin nieder, dass im Basis-Manifest eines Bundles zwar Abhängigkeiten, aber keine Dienste deklariert werden. Die grundlegende OSGi-Architektur spezifiziert Dienste und ihre Attribute in Form von Java-Interfaces. Somit können Dienstanbieter und -nutzer nicht aneinander gekoppelt werden, ohne alle Bundles bereits beim Start der Laufzeitumgebung über einen Java-`Classloader` zu laden. Abhilfe bieten die auf Initiative der Eclipse-Entwickler in der aktuellen OSGi-Spezifikation in den standardisierten Dienstkatalog aufgenommenen *deklarativen Dienste* (*Declarative Services*) [OSG06b, Kapitel 112, S. 281–315]. Falls in einer Implementierung des OSGi-Rahmenwerkes die optionale Komponentendienstumgebung (*Service Component Runtime, SCR*) installiert ist, können die von Bundles bereitgestellten und benötigten Dienste deklarativ mittels Manifesten spezifiziert werden. Die SCR berücksichtigt bei der Vermittlung zwischen Dienstanbietern und -nutzern deklarierte Diensteigenschaften sowie Abhängigkeiten des Dienstanbieters von anderen Dienstanbietern. Die von der SCR durchgeführte Kopplung zwischen Anbieter und Nutzer kann entsprechend der Deklaration unterschiedlich eng ausfallen und eine unterschiedliche Lebensdauer aufweisen. Zu diesem Zweck kann die SCR mit zwischengeschalteten Platzhalterobjekten oder individuell je Nutzung erzeugten Dienstinstanzen arbeiten, Bundles bei Bedarf aktivieren und direkt auf die Deaktivierung von Bundles reagieren.

Die SCR kennt drei Varianten des Lebenszyklus' einer Dienstkomponente (*Service Component*), die in Abbildung 3.12 als Petrinetze dargestellt sind. Grundlage der Netze ist die Beschreibung der Lebenszyklen anhand von Aktivitätsdiagrammen in [OSG06b, Abschnitt 112.5, S. 298–305]. Im Wesentlichen werden dabei zwei Zustände verwaltet: Die Verfügbarkeit (*enabled*) der Dienstkomponente hängt von der Aktiviertheit des bereitstellenden Bundles ab, kann aber auch programmgesteuert ausgesetzt werden. Verfügbare Dienstkomponenten können wiederum unter bestimmten Bedingungen aktiviert und deaktiviert werden, wobei nur eine aktive Komponente als Dienstanbieter gebunden werden kann. Zu den notwendigen Bedingungen gehört neben der Verfügbarkeit vor allem die Erfüllung von Abhängigkeiten zu anderen Diensten (*satisfied*), falls der Dienstanbieter gleichzeitig als Nutzer auftritt.

Während *immediate Components* aktiviert werden, sobald ihre Abhängigkeiten erfüllt sind, unterscheiden *delayed Components* zusätzlich noch den Zustand „registered“. Sie werden in der Regel erst aktiviert, wenn eine tatsächliche Dienstnutzung ansteht.




*Note: "becomes unsatisfied" in OSGi Declarative Services Specification includes the case where the component is disabled. Here this case has been modelled explicitly.*

*Note: The coupling of "Enable" and "Disable" with the bundle's life–cycle holds for all occurrences of these events, although it is not mentioned at every transition.*

Abbildung 3.12: Lebenszyklen von deklarativen OSGi-Diensten

*Factory Components* zeigen nach außen dieselben Zustände wie *delayed Components*, erzeugen aber intern je Nutzung eigene Dienstkomponenten-Instanzen. Da diese Instanzen parametrisiert erzeugt werden, kann sich die Menge ihre zu erfüllenden Abhängigkeiten voneinander und von der Grundkonfiguration unterscheiden.

Die Übergänge der drei Lebenszyklusvarianten werden in der Regel von der SCR auf Grundlage der deklarativen Dienstbeschreibungen verwaltet, ohne dass der Programmierer des Bundles weitere Rücksicht nehmen muss. Insbesondere implementiert die SCR zusätzlich zu den durch das OSGi-Rahmenwerk verwalteten statischen Abhängigkeiten zwischen Bundles eine automatisierte Verwaltung dynamischer oder funktionaler[25] Abhängigkeiten zwischen Dienstkomponenten, die z. B. in Ketten oder Hierarchien aus Dienstanbietern und -nutzern organisiert sein können.

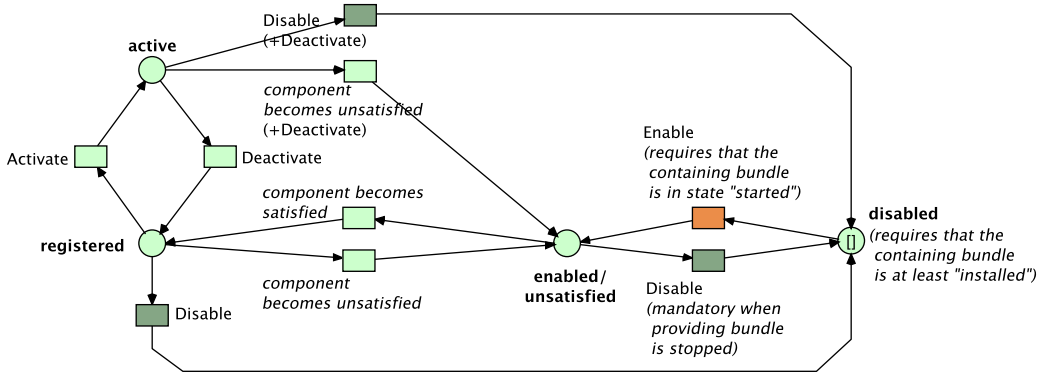



Abbildung 3.13: Vervollständigter Lebenszyklus einer Dienstkomponente

Die Kopplung des Lebenszyklus' einer Dienstkomponente an den Bundle-Lebenszyklus wird im Petrinetz in Abbildung 3.13 stärker verdeutlicht, in dem der Zustand der Nicht-Verfügbarkeit explizit modelliert ist. In diesem Zustand ist die Dienstkomponente für andere Komponenten im System nicht sichtbar. Die Deklaration der Dienstkomponente ist allerdings der SCR schon bekannt, da das die Komponente enthaltende Bundle bereits im OSGi-Laufzeitsystem installiert ist. Die Zustände „disabled“ und „enabled“ entsprechen also den Bundle-Lebenszykluszuständen „installed“ und „started“ (mit der Ausnahme, dass auch Dienstkomponenten eines gestarteten Bundles „disabled“ sein dürfen). Die SCR nutzt diesen Zusammenhang, indem sie den Lebenszyklus von Bundles steuert, um Dienstkomponenten nach Bedarf verfügbar zu machen.

Obwohl die OSGi-Dienstleistungsschicht nun seit drei Jahren in der Eclipse-Plattform zur Verfügung steht, werden Eclipse-Plugins weiterhin im Wesentlichen anhand des Erweiterungskonzepts implementiert. Im Rahmen der Umstellung wurde die Orthogonalität der beiden Konzepte als Grund genannt, beide Konzepte nebeneinander beizubehalten [vgl GHM+05, S. 294]. Letztendlich scheint aber das Dienstleistungskon-

[25] Das Adjektiv „*funktional*“ wird in weiten Teilen dieser Arbeit nicht im mathematischen Sinne verwendet. Eine „*funktionale Abhängigkeit*“ bedeutet eine Abhängigkeit vom Vorhandensein bestimmter Funktionalität im System (siehe auch Unterabschnitt 3.4.2).

zept im Rahmen von Eclipse-Plugins eine untergeordnete Rolle zu spielen. Clayberg und Rubel erläutern beispielsweise in der für die aktuelle Eclipse-Version überarbeiteten Auflage ihres Buches [CR06] zwar die Architektur der OSGi, erwähnen aber das Konzept der OSGi-Services mit keinem Wort.

Für die Bevorzugung des Erweiterungskonzepts in Eclipse sprechen mehrere Gründe, sowohl pragmatische als auch konzeptionelle. Pragmatisch gesehen ist zum ersten die gesamte existierende Eclipse-Komponenteninfrastruktur durch Erweiterungspunkte definiert. Erweiternde Plugins müssen also dem von den Plattform-Plugins verwendeten Erweiterungskonzept treu bleiben, und eine Umstellung zentraler Schnittstellen hätte Inkompatibilitäten mit allen existierenden Plugins zur Folge. Zum zweiten ist zwar die Spezifikation, aber nicht die Implementierung der deklarativen Dienste so weit fortgeschritten, dass die *Lazy Loading Rule* wieder greifen kann. Zum dritten war eine der PDE vergleichbare Werkzeugunterstützung zur Pflege von Deklarativen-Dienst-Manifesten bis vor Kurzem[26] schlichtweg nicht vorhanden [vgl. Bar07, Making a Choice].

Konzeptionell spricht für die Beibehaltung des Erweiterungskonzepts dessen bessere Eignung für Plugin-Systeme im Vergleich zum Dienstleistungskonzept. Das Dienstleistungskonzept ist zwar flexibler, trägt aber weniger zur Strukturierung des resultierenden Systems bei (vgl. Diskussion in Unterabschnitt 4.1.3.1). Beispielsweise erscheint eine direkte Abbildung des Eclipse-Erweiterungskonzepts auf das reine OSGi-Dienstkonzept problematisch: Die $1 : n$-Beziehung zwischen einem Erweiterungspunkt und mehreren Erweiterungen müsste auf eine $1 : n : 1$-Beziehung zwischen Spezifikation, mehreren Anbietern und einem Nutzer abgebildet werden. Die OSGi-Dienstverwaltung unterstützt nur die Suche nach Anbietern von Dienstleistungen, nicht nach potentiellen Nutzern. Da ein Erweiterungspunkt nach passenden Erweiterungen suchen können muss, sind Erweiterungspunkte eher als Nutzer der von Erweiterungen bereitgestellten Funktionalität denn als Anbieter von Dienstleistungen für Erweiterungen zu verstehen. Eine Einschränkung der Dienstnutzerzahl auf 1 wird aber vom OSGi-Rahmenwerk nicht unmittelbar unterstützt, müsste also von jedem Dienstanbieter (d.h. jeder Erweiterung) explizit in der Implementierung berücksichtigt werden.[27]

#### 3.2.1.7 Abhängigkeiten (Dependencies)

Ein Plugin ist von einem anderen Plugin *abhängig* (*dependent*), wenn mindestens eine seiner Java-Klassen oder -Interfaces von einem im anderen Plugin definierten Java-Typen erbt, auf statische Methoden oder Felder eines solchen Typs zugreift oder einen solchen Typen im Quellcode zur Deklaration eigener Variablen, Parameter oder Rückgabewerte verwendet (siehe hierzu auch die Erläuterung der Java-Classloader-Architektur in Unterabschnitt 3.2.1.2). Im Quellcode lassen sich solche Abhängigkeiten gut

[26] Die Werkzeugunterstützung für Deklarative-Dienst-Manifeste ist erstmals in Eclipse Version 3.5 enthalten, welche erst im Juni 2009 – also ca. 4 Jahre nach der Umstellung auf die OSGi-Plattform – fertig gestellt wurde.

[27] Zur Beschränkung der Nutzerzahl ist anzumerken, dass auch das Rahmenwerk für das klassische Eclipse-Erweiterungskonzept keine Beschränkung der Nutzerzahl von Erweiterungen durchsetzt. Standardmäßig wird zwar nur der deklarierte Erweiterungspunkt über die installierten Erweiterungen informiert, aber die Erweiterungsverwaltung hält diese Informationen als global zugängliches Singleton auch für beliebige Plugins zugänglich bereit.

an den durch die Sprache Java vorgeschriebenen `import`-Deklarationen erkennen. Es handelt sich also um eine *statische Abhängigkeit* auf Ebene des Java-Typsystems, die bereits bei der Kompilation bekannt ist und die auch während der Laufzeit unverändert besteht.

Solche Abhängigkeiten entstehen notgedrungen, wenn eine Erweiterung auf Schnittstellen des erweiterten Plugins zurückgreift oder wenn ein Dienstanbieter oder -nutzer die Schnittstellen des implementierten Dienstes referenziert. Weitere Abhängigkeiten können beispielsweise entstehen, wenn auf den Code anderer Plugins im Sinne einer Funktionsbibliothek zurückgegriffen wird. Strenggenommen stellt auch eine Funktionsbibliothek eine Dienstleistung bereit, diese wird aber meist nicht über die Dienstabstraktion deklariert, da sie in der Regel zustandslos genutzt werden kann.

Klassische Eclipse-Plugins deklarieren ihre Abhängigkeit von einem anderen Plugin, indem sie Namen und Version des benötigten (*required*) Plugins im Manifest eintragen. Dieser Eintrag bewirkt, dass die Eclipse-Laufzeitumgebung das abhängige Plugin nur aktiviert, wenn alle benötigten Plugins vorhanden sind. Das abhängige Plugin kann dann während seiner Lebensspanne davon ausgehen, dass alle Klassen und die gesamte Funktionalität des angeforderten Plugins zur Verfügung stehen. Somit entsteht eine partielle Ordnung über den Plugins, die sich auch in der Delegationsbeziehung der Classloader der Plugins widerspiegelt.

OSGi-Bundles haben zusätzlich die Möglichkeit, lediglich das Vorhandensein bestimmter Java-Packages durch eine `Import-Package`-Deklaration im Manifest anzufordern. In diesem Fall wird die Abhängigkeitsbeziehung feiner granuliert, da sie nicht mehr zwischen ganzen Bundles, sondern nur noch zwischen einzelnen Java-Packages besteht. Zudem muss jedes OSGi-Bundle festlegen, welche Java-Packages es anderen Bundles zur Verfügung stellen (exportieren) möchte. Auch eine grobe Abhängigkeit auf der Bundle-Granularitätsebene umfasst dann lediglich die exportierten Packages. Die individuell konfigurierten Classloader der Bundles im OSGi-Rahmenwerk realisieren eine Delegationsbeziehung, die der Abhängigkeitsbeziehung entsprechend feiner – also auf Package-Ebene – granuliert ist. Dabei ist die OSGi-Classloader-Implementierung in der Lage, aus mehreren verfügbaren Implementierungen desselben Packages in verschiedenen Bundles eine auszuwählen, so dass eventuell Code eines Bundles teilweise (genauer: Package-weise) durch Code anderer Bundles verdeckt werden kann.

Im OSGi-Rahmenwerk ist grundsätzlich das Bundle, welches ein Package exportiert, austauschbar. Die statische Abhängigkeit zwischen Bundles verwandelt sich damit in eine zur Laufzeit variable Beziehung, die allerdings nur bei bestimmten Lebenszyklus-Zustandsübergängen verändert werden kann. Damit entsteht ein Flexibilitätsgewinn, der z. B. das Einspielen korrigierter oder alternativer Fassungen des exportierenden Plugins ermöglicht. Diese Flexibilität bedeutet aber eine besondere Rücksichtnahme im abhängigen Plugin, da es nicht mehr davon ausgehen kann, dass die angeforderte Funktionalität dauerhaft vom selben Anbieter erbracht wird.

Das Rahmenwerk der OSGi bietet weitere Möglichkeiten, Abhängigkeitsbeziehungen zwischen Bundles zu deklarieren, die in speziellen Situationen nützlich sein können. So können *Fragmente* von Bundles abgespalten werden, um alternative Elemente wie z. B. internationalisierte Sprachdaten aus einem Bundle auszulagern, ohne weitere Bundle-Lebenszyklen verwalten zu müssen. Die Ressourcen des Fragments werden dann in den Classloader des zugehörigen Bundles integriert. Zusätzliche Flexibilität bieten

*dynamische Imports*, die zwar ebenfalls im Manifest deklariert werden, aber nicht zum Zeitpunkt der Abhängigkeitsauflösung des Bundles bereits erfüllt sein müssen, sondern erst bei Bedarf zu einer dynamischen Abhängigkeit zwischen Bundles führen. Details zu diesen und weiteren Möglichkeiten finden sich in der Spezifikation [OSG06a, Kapitel 3].

#### 3.2.1.8 Prinzipien

Ein Großteil der zu Eclipse verfügbaren Literatur gibt im Wesentlichen Hilfe für Entwickler zu zwei Punkten: Zum einen wird der Umgang mit den Eclipse-Werkzeugen zur Softwareentwicklung beschrieben und zum anderen werden Anleitungen gegeben, wie sich eigene Plugins gut in die umfangreiche und bereits vorhandene Komponenten-Infrastruktur einfügen können.

Clayberg und Rubel geben beispielsweise in [CR06] jede Menge Anleitungen, wie man eigene Funktionalität bruchlos in die von zentralen Plugins der Eclipse-Plattform angebotenen Erweiterungspunkte eingliedert, und welche Werkzeuge zur Unterstützung dieser Arbeit zur Verfügung stehen. Zudem verweisen sie immer wieder auf Anforderungen und Empfehlungen eines für kommerzielle Eclipse-Ableger in der Produktfamilie der *Rational Software Development Platform* verfassten Kriterienkataloges, der z. B. das Verhalten der Anwendung in bestimmten Situationen oder die dem Anwender präsentierte Nomenklatur vorgibt.

Gamma und Beck verfolgen in [GB03] im Prinzip dasselbe Ziel, gehen aber etwas distanzierter heran. Zunächst definieren sie den Kreis der Eclipse-Involvierten näher: Anwender (*User*) benutzen, was der Eclipse-Standardumfang an Werkzeugen hergibt, und können sich zu Konfigurierern (*Configurer*) weiterentwickeln, die das System an ihre Bedürfnisse anpassen, indem sie von Plugins gebotene Konfigurationsmöglichkeiten nutzen. Erweiterer (*Extender*) programmieren eigene Ergänzungen als Plugins und werden zu Veröffentlichern (*Publisher*), wenn sie ihre Plugins anderen Anwendern zugänglich machen. Stattet ein Plugin-Entwickler seine Software mit Erweiterungspunkten (siehe Unterabschnitt 3.2.1.5) aus, so wird er zum Ermöglicher (*Enabler*) weiterer Erweiterungen. [Vgl. GB03, S. 1ff.]

Gamma und Beck stellen dann Regeln auf, welche eine bessere Einfügung der Erzeugnisse von Erweiterern (inklusive Veröffentlichern und Ermöglichern) in die Eclipse-Plattform gewährleisten sollen. Einige dieser Regeln geben einen guten Einblick in die Intentionen, die hinter der Eclipse-Plugin-Architektur stehen.

Die erste Regel [*Contribution Rule*, GB03, S. 15] betont den radikalen Ansatz der Softwarearchitektur der Eclipse-Plattform, dass jegliche Funktionalität als externer Beitrag, d.h. als Plugin, aufzufassen ist – es gibt keine Kernfunktionalität außer dem Plugin-System selber. Ferner soll bereitgestellte Funktionalität in mehrere Schichten – sprich: mehrere Plugins – aufgeteilt werden. Eine IDE-spezifische Vorgabe liefert die *STRATA Rule* [GB03, S. 356], die eine Trennung zwischen allgemeiner und programmiersprachenspezifischer Funktionalität sowie zwischen Kern- und Oberflächenfunktionalität empfiehlt. Um das Gesamtsystem möglichst flexibel zu halten, fordert die *Invitation Rule* [GB03, S. 65] Plugin-Entwickler auf, Erweiterungsmöglichkeiten in ihrer bereitgestellten Funktionalität zu erkennen und diese explizit als Erweiterungspunkte zu deklarieren [*Explicit Extension Rule*, GB03, S. 66].

Aus dem Verzicht auf eine Anwendungs-Kernfunktionalität ergibt sich, dass eine Eclipse-Installation üblicherweise aus so vielen Plugins besteht, dass für eine akzeptable Performanz beim Start des Systems nur wenig Zeit für jedes einzelne Plugin aufgebracht werden kann. So motiviert sich die bereits mehrfach angesprochene *Lazy Loading Rule* [GB03, S. 16], welche vorgibt, dass Plugins erst so spät wie möglich geladen und initialisiert werden sollen. Es handelt sich hierbei also um eine Regel, die zwar „nur“ aus Performanzgründen aufgestellt wird, aber erheblichen Einfluss auf die Gestaltung von Schnittstellen innerhalb der Softwarearchitektur hat. Dass gerade diese aus funktionaler Sicht zweitrangige Regel in der Eclipse-Literatur so stark betont wird, liegt vermutlich daran, dass das Einhalten dieser Regel durch die Softwarearchitektur nicht erzwungen, sondern nur unterstützt wird. Stattdessen müssen alle Erweiterer (insbesondere die Ermöglicher) ihren Code explizit auf diese Regel hin ausrichten.

Einige Regeln definieren die bereits in Unterabschnitt 3.2.1.5 behandelte Erweiterungsbeziehung zwischen Plugins. Zunächst sollen Erweiterungspunkte beliebig viele Erweiterungen akzeptieren [*Diversity Rule*, GB03, S. 67]. In Gegenrichtung stellt die *Sharing Rule* [GB03, S. 24] klar, dass Erweiterungen Funktionalität hinzufügen, aber nicht ersetzen können. Bietet ein Plugin gleichzeitig zu einem eigenen Erweiterungspunkt passende Funktionalität mit an, soll diese als gleichberechtigte Erweiterung eingebunden werden [*Fair Play Rule*, GB03, S. 66], so dass Plugins sich selber erweitern dürfen und sollen.

Etliche Regeln sollen einen guten, einheitlichen Stil beim Bereitstellen und Nutzen von Erweiterungspunkten forcieren. Selbstverständlich müssen die Verträge vorgegebener Schnittstellen eingehalten werden [*Conformance Rule, Program to API Contract Rule*, GB03, S. 26, S. 117] sowie möglichst wohldefinierte, stabile und minimale Schnittstellen veröffentlicht werden [*Explicit API Rule, Stability Rule, Defensive API Rule*, GB03, S. 266]. Die *Safe Platform Rule* [GB03, S. 44] und *Good Fences Rule* [GB03, S. 77] fordern Ermöglicher auf, ihre Erweiterungsschnittstelle fehlertolerant zu implementieren, so dass fehlerhafte Erweiterungen keine Folgefehler oder gar Arbeitsunfähigkeit im erweiterten Plugin bewirken können. Gleichzeitig sollen Erweiterungen eigene Fehler erkennen und sich selber durch aussagekräftige Fehlermeldungen als Verursacher zu erkennen geben [*Responsibility Rule*, GB03, S. 171].

Die dem Anwender präsentierte Zahl der Erweiterungen soll mit einigen weiteren Regeln in Zaum gehalten werden. Muss in der Implementierung eines Erweiterungspunkts zwischen mehreren verfügbaren Erweiterungen gewählt werden, so ist die Entscheidung dem Anwender zu überlassen [*User Arbitration Rule*, GB03, S. 222], wobei zwischen häufig genutzten und anderen, unüblichen Erweiterungen unterschieden werden darf [*Other Rule*, GB03, S. 242]. Zudem sollen Erweiterungen ihre Funktionalität nur zur Verfügung stellen, wenn das System einen – deklarativ spezifizierten oder von der Implementierung überprüften – Systemzustand eingenommen hat, in dem die Funktionalität auch sinnvoll nutzbar ist [*Relevance Rule*, GB03, S. 43]. Als abstrakte Empfehlung fordert die *Integration Rule* [GB03, S. 129] Entwickler von Erweiterungen auf, eigene Konzepte und Mechanismen zugunsten von bereits in der Eclipse-Plattform verwendeten Metaphern und Erweiterungspunkten zurückzustellen, so dass eine nach außen hin nahtlose Integration der erweiternden Funktionalität in das Gesamtsystem möglich wird.

### 3.2.2 Zusammenfassung

Eclipse ist sowohl eine allgemeine IDE als auch eine allgemeine Anwendungs-Plattform, die ausschließlich aus Plugins besteht. Das zugrunde liegende Kernsystem zur Verwaltung von Plugins verwendet die `Classloader`-Abstraktion der Implementierungssprache Java, um Code aus verschiedenen Quellen zur Laufzeit zu einer Anwendung zusammenzufassen. Die Eclipse-eigene Implementierung des Kernsystems wurde vor kurzem durch das ebenfalls Java-basierte OSGi-Rahmenwerk zur Verwaltung von Bundles ersetzt, um dynamisch Code ins System aufnehmen und daraus entfernen zu können. Die Spezifikationen der OSGi definieren zu diesem Zweck einen ausgefeilten Bundle-Lebenszyklus, der neben dem Starten und Stoppen von Bundles auch deren Installation, Aktualisierung und Deinstallation umfasst. Die alten Schnittstellen wurde beibehalten und auf das neue System abgebildet, so dass beide Softwarearchitekturen nebeneinander und ineinander integriert zur Verfügung stehen.

Es gibt keine einheitliche und kaum eine präzise Definition eines Eclipse-Plugins, lediglich technische Beschreibungen der Bestandteile. Ein klassisches Eclipse-Plugin besteht aus Klassen, Ressourcen, Manifesten und einer `Plugin`-Klasse. Architektonisch definiert sich ein Plugin über seine explizit deklarierten Schnittstellen, Abhängigkeiten, Erweiterungspunkte und Erweiterungen. Ein OSGi-Bundle ist einem Eclipse-Plugin strukturell ähnlich, anstelle der Erweiterungsbeziehung wird aber eine Dienstleistungs- und -nutzungsbeziehung zwischen den Bundles deklariert. Seit der Umstellung des Plugin-Systems auf das OSGi-Rahmenwerk werden Eclipse-Plugins auf OSGi-Bundles abgebildet, eine automatische Abbildung der Erweiterungs- auf die Dienstbeziehung findet allerdings nicht statt. Die Abhängigkeitsbeziehungen zwischen Plugins müssen ebenso wie die Erweiterungs- und Dienstbeziehungen in Manifesten deklariert werden, damit die Plugin- bzw. Bundle-Verwaltung die Informationen zur Steuerung des Gesamtsystems nutzen kann.

Eclipse-Plugins, die einen *Erweiterungspunkt* deklarieren, können die Funktionalität von Plugins, die passende *Erweiterungen* deklarieren, integrieren. Dabei bietet üblicherweise das erweiterbare Plugin ein Interface an, das erweiternde Plugins implementieren, um ihre Funktionalität anzubieten. Dadurch entsteht notwendigerweise eine statische Abhängigkeit zum erweiterbaren Plugin. Die Erweiterungsbeziehung verändert sich während der Laufzeit nicht. Sie ist eine $1 : n$-Beziehung zwischen Erweiterungspunkt und Erweiterungen, wobei jedes Plugin beliebig viele Erweiterungspunkte und Erweiterungen beinhalten kann. Weitere Beziehungen zwischen Plugins können lediglich über die Abhängigkeitsbeziehung zwischen Plugins definiert werden. Die PDE unterstützt Entwickler bei der sonst arbeitsintensiven Manifestpflege.

OSGi-Bundles können sich über *Dienste* koppeln. Dabei lassen sich drei Parteien identifizieren: öffentliche Spezifikation (Interfaces), Dienstanbieter (private Implementierung) und Dienstnutzer (private Implementierung). Die Dienstnutzungsbeziehung ist $1 : n : m$ (Spezifikation, Anbieter, Nutzer), eine statische Abhängigkeit besteht nur jeweils zwischen Anbieter und Spezifikation sowie Nutzer und Spezifikation, die Beziehungen zwischen Anbieter und Nutzer können zur Laufzeit entstehen und sich auflösen. Neben einfachen, in Java spezifizierten Diensten können in einem OSGi-Rahmenwerk mit installierter SCR komplexe Dienstbeziehungen in Manifesten deklariert werden. Die Dienstdeklarationen umfassen neben Angebot und Nutzung auch Anga-

ben zur Instanziierung der anbietenden Objekte sowie Abhängigkeitsketten zwischen Dienstangeboten. Es gibt keine der PDE vergleichbare Werkzeugunterstützung.

Eine ausschließlich auf Plugins aus unterschiedlichen Quellen basierende Anwendung wie Eclipse, die zudem jeden Anwender ermutigt, sich auch als Entwickler derselben Anwendung zu betätigen, kann nur mit Mühe ein einheitliches Erscheinungsbild und Bedienkonzept aufweisen. Daher stellen Gamma und Beck in [GB03] Regeln auf, welche die bessere Einfügung eigener Erweiterungen in die Eclipse-Plattform gewährleisten sollen. Abgedeckt werden dabei neben den grundlegenden Eigenschaften der Erweiterungsbeziehung zwischen Plugins das „Wohlverhalten“ von Erweiterungspunkten bzw. Erweiterungen und die Präsentation der Funktionsvielfalt gegenüber den Anwendern.

## 3.3 Weitere Arbeiten

Neben den in den vorigen Abschnitten vorgestellten prominenten Open-Source-Systemen mit ausgefeilten Plugin-Architekturen gibt es noch unzählige weitere Software-Systeme, die sich mehr oder weniger stark mit den Attribut der Erweiterbarkeit durch Plugins schmücken. Diese können im Rahmen dieser Arbeit nicht alle betrachtet werden. Allerdings existieren ein paar Projekte, deren Autoren sich ähnlich grundlegend mit der Architektur von Plugin-Systemen auseinandergesetzt haben.

Die folgende Auswahl ist nicht vollständig, sie umfasst im Wesentlichen solche Arbeiten, die besonders zur Klärung oder Definition des Plugin-Konzepts beitragen. Nicht in der Auswahl enthalten sind Arbeiten, die sich ausschließlich im Umfeld der CBSE (siehe Abschnitt 2.2) einordnen lassen. Wie im Unterabschnitt 2.2.5 klar geworden ist, haben komponentenbasierte Systeme zwar viele Gemeinsamkeiten mit Plugin-Systemen, die Ansätze verfolgen aber unterschiedliche Ziele.

Im Bereich der wissenschaftlichen Publikationen finden sich außer den in den Unterabschnitten 2.2.5 und 2.4.3 vorgestellten Arbeiten nur wenige, die sich explizit dem Thema der Plugin-Systeme widmen. Allerdings handelt es sich bei Plugin-Systemen um dynamisch rekonfigurierbare Softwarearchitekturen, deren Fragestellungen auch in anderen Forschungsbereichen relevant sind. In der folgenden Zusammenstellung werden auch einige dieser Fragestellungen kurz angeschnitten.

### 3.3.1 Java Plugin Framework (JPF)

Das Java Plugin Framework [JPF] implementiert eine an das System der klassischen Eclipse-Plugins (vgl. Abschnitt 3.2) angelehnte und um einige Fähigkeiten erweiterte Softwarearchitektur. Die Entwicklung des JPF begann 2004 und hat im Januar 2007 zu einer ersten als „stabil“ bezeichneten Version geführt. Das JPF ist in der Lage, zentrale Elemente des Systems wie z. B. die Plugin-Datenbank (*PluginRegistry*), das Dateiformat der Manifeste oder das Auffinden von Plugin-Dateien durch individuelle Implementierungen zu ersetzen. [Vgl. Ols07c, Ols07b]

Interessant ist die vom Autor betonte Trennung der Konzepte *Modularisierung* und *Erweiterbarkeit*. Die Modularisierung wird durch *Plugins* erreicht, welche es erlauben,

die Abhängigkeiten auf Klassenebene durch deklarierte Abhängigkeiten auf der Modulebene zu gewährleisten. Für die Anwendungsfunktionalität und den Softwareentwurf relevant ist jedoch die Beziehung zwischen *Extension Points* und *Extensions*. [Vgl. Ols07a]

Die Classloader-Architektur des JPF ähnelt der des klassischen Eclipse-Plugin-Systems und beherrscht auch das Auflösen von Klassenabhängigkeiten *entgegen* der Delegationsrichtung. Die Eclipse-OSGi-Implementierung *Equinox* beherrscht ebenfalls das Auflösen von symbolischen Referenzen entgegen oder abseits der Richtung der deklarierten Bundle-Abhängigkeiten, der entsprechende Mechanismus heißt *Buddy Class Loading* und kann für verschiedene Auflösungswege konfiguriert werden [vgl. Ble06]. Diese Fähigkeit wird benötigt, wenn der Code eines Plugins über die Java-*Reflection*-Schnittstelle auf die Deklarationen von Klassen aus untergeordneten Plugins zugreifen möchte. Dies ist beispielsweise bei dem in der Sprache Java vorgesehenen Serialisierungsmechanismus notwendig, wird aber auch von anwendungsübergreifenden Funktionsbibliotheken wie z. B. Logging-Rahmenwerken verwendet.

### 3.3.2 NetBeans

NetBeans [Net] ist – ähnlich wie das in Abschnitt 3.2 bereits ausführlich vorgestellte Eclipse – eine in Java implementierte Open-Source-IDE, deren Code ein allgemeines Rahmenwerk zur Entwicklung erweiterbarer Software umfasst. Die Entwicklung des NetBeans-Projektes wird im Wesentlichen von der Firma Sun unterstützt, findet aber in einer Open-Source-Community mit diversen industriellen Partnern statt. Die im Rahmen von NetBeans und Eclipse zur Verfügung stehenden Werkzeuge decken aus Sicht eines Anwenders der IDE (also eines Software-Entwicklers) im Kern dieselben Bereiche ab. Unterschiede bestehen jedoch in der darunter liegenden Architektur der jeweiligen Plattformen, und damit auch der grundlegenden Plugin-Systeme.

**Module** Das zentrale Konzept zur Gliederung der Plattform ist in NetBeans das „Modul“, welches aber auch gelegentlich als „Plugin“ bezeichnet wird. Boudreau et al. differenzieren in [BTW07] die beiden Konzepte nicht über ihre identischen praktischen und technischen Merkmale, sondern über ihre Verwendung. Da der Begriff „Plugin“ sich von der Tätigkeit des Einklinkens (*to plug in*) ableitet, sei er nur auf Module anwendbar, welche die Kernfunktionalität einer Anwendung oder Plattform erweitern. Die Module hingegen, die als gleichberechtigte Teile den Kern einer Anwendung oder Plattform ausmachen, könnten nicht als Plugins bezeichnet werden. Dagegen könnten mehrere Module gemeinsam ein Plugin formen, wenn sie Teile einer inhaltlichen Anwendungserweiterung sind. [Vgl. BTW07, S. 22]

Ein NetBeans-Modul besteht zunächst aus einer JAR-Datei, deren Manifest Attribute zur Versionierung und zur Beschreibung statischer und funktionaler[28] Abhängigkeiten des Moduls enthält. An die Stelle einer einheitlich vorgegebenen Erweiterungsbeziehung à la Eclipse oder Dienstbeziehung à la OSGi (vgl. Abschnitt 3.2) treten in der NetBeans-Architektur zwei alternative Konzepte, um Beziehungen zwischen

[28] Das Adjektiv „*funktional*“ wird in weiten Teilen dieser Arbeit nicht im mathematischen Sinne verwendet. Eine „*funktionale Abhängigkeit*“ bedeutet eine Abhängigkeit vom Vorhandensein bestimmter Funktionalität im System (siehe auch Unterabschnitt 3.4.2).

den Modulen aufzubauen. Zum einen können mithilfe von *Lookups* Dienste im Sinne der Java-`ServiceLoader`-Architektur deklariert, gefunden und genutzt werden, zum anderen können Module über das Konzept eines *Dateisystems* lose gekoppelte Beziehungen herstellen. Beide Konzepte sind im Rahmenwerk bereits implementiert und unterstützen eine deklarative statische sowie eine programmgesteuerte dynamische Konfiguration.

**Lookups** Zentrales Element der Dienstbeziehung ist die Klasse `Lookup`, welche zum einen die Funktionalität des Java-`ServiceLoader`s (vgl. Unterabschnitt 3.3.7) dupliziert, diese aber erheblich erweitert. Ein globaler `Lookup` sammelt alle den Classloadern bekannten Service-Manifeste und instantiiert auf Nachfrage die so konfigurierten Implementierungen. Falls ein Teil des Systems ebenfalls solche Sammel- und Suchfunktionalität benötigt, können anders spezialisierte `Lookups` lokal von Objekten bereitgestellt werden. `Lookup`s beziehen ihre Informationen nicht nur aus Manifesten, sondern können auch programmgesteuert verändert werden, wobei `Lookup`-Nutzer bei Interesse über Änderungen der Ergebnisse ihrer Suche informiert werden. So eignet sich das `Lookup`-Konzept auch zur Implementierung beliebiger Sammlungen wie z. B. einer Menge ausgewählter Objekte im Editor. [Vgl. BTW07, Kapitel 4 und 5].

Das `Lookup`-Konzept kann sowohl die Erweiterungsbeziehung als auch die Dienstbeziehung einfach nachbilden. Ein Erweiterungspunkt spezifiziert sowohl das durch Erweiterungen zu implementierende Interface und nutzt einen `Lookup`, um nach installierten Erweiterungen zu suchen. Bei der Dienstbeziehung wird das Dienst-Interface durch ein drittes Modul spezifiziert, von dem dann sowohl die Implementierung als auch der Nutzer abhängig sind. In diesem Fall sucht der Dienstnutzer über den `Lookup` nach angebotenen Dienstimplementierungen. Beide Beziehungsarten können dynamisch veränderlich ausgelegt werden, da das `Lookup`-Konzept dies unterstützt. Die Reaktion auf dynamische Veränderungen muss aber beim jeweiligen `Lookup`-Nutzer explizit implementiert werden.

**Dateisystem** Das Dateisystemkonzept vereinheitlicht physische, verteilte oder virtuelle Dateisysteme, so dass im Code nicht zwischen echten Dateien, Netzwerkzugriffen oder logischen Strukturen im Speicher unterschieden werden muss. Die gemeinsame Eigenschaft aller Dateisysteme ist deren hierarchische Organisation. Zudem werden alle Dateien inhaltlich typisiert und es können Verweise quer durch die Baumstruktur angelegt werden. Die NetBeans-Plattform nutzt diese Eigenschaften, um die Systemkonfiguration aus (hierarchisch strukturierten) XML-Dateien als Dateisystem abzubilden [vgl. BTW07, Kapitel 6]. So kann ähnlich dem Prinzip der Mozilla-XUL-Overlays (vgl. Unterabschnitt 3.1.2.3) aus den XML-Dateien mehrerer Module eine einheitliche Struktur erzeugt werden. Darüber hinaus erlaubt das NetBeans-Dateisystem die programmgesteuerte Modifikation der virtuellen Verzeichnisse und Dateien, so dass auch Konfigurationsänderungen persistiert werden können.

Das NetBeans-Dateisystem kann ebenfalls zur Konfiguration dynamischer Erweiterungsbeziehungen herangezogen werden. Ein Erweiterungspunkt gibt ein Verzeichnis in der Hierarchie des Dateisystems vor, dessen Inhalt er auswertet oder laufend auf Veränderungen überwachen lässt. Erweiterungen legen in dem vorgegebenen Verzeich-

nis logische Dateien an, die auf ihre angebotene Funktionalität verweisen. Das virtuelle Dateisystem wird in der NetBeans IDE beispielsweise verwendet, um das Anwendungsmenü zu konfigurieren, jede virtuelle Datei entspricht dabei einem Menüeintrag. Das Konzept eines Dateisystems erlaubt allerdings Seiteneffekte zwischen Modulen quer zu Erweiterungs- oder Dienstbeziehungen, indem beispielsweise ein Modul Dateien löscht, die ein anderes Modul angelegt hat (um z. B. überlappend angebotene Funktionalität zu verbergen oder zu ersetzen).

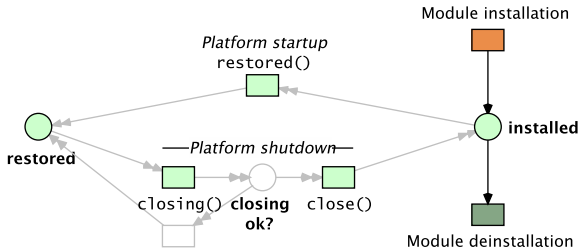



Abbildung 3.14: Der Lebenszyklus von NetBeans-Modulen als Petrinetz

**Lebenszyklus** Die dynamischen Fähigkeiten des `Lookup`- und des Dateisystem-Konzepts erlauben es der NetBeans-Plattform, den Lebenszyklus der Module sehr einfach zu gestalten (siehe Abbildung 3.14). Ein Modul kann lediglich zum Zeitpunkt des Systemstarts „restored“ und zum Zeitpunkt des Systemendes „close“ benachrichtigt werden. Die NetBeans-Plattform garantiert zwar die Einmaligkeit dieser Benachrichtigung pro Systemlaufzeit, jedoch nicht, dass diese Nachrichten den allerersten bzw. -letzten Aufruf von Modulcode darstellen. [Vgl. BTW07, Abschnitt 3.3, S. 29ff.]

Die Klasse `ModuleInstall`, welche Benachrichtigungen über Lebenszyklusereignisse für ein Modul entgegennimmt, kennt neben „restored“ auch die Ereignisse „installed“ und „updated“. Letztere werden *alternativ* zu „restored“ ausgelöst, wenn ein Modul erstmalig nach seiner Installation oder Aktualisierung gestartet wird. Beide Ereignisse sind allerdings inzwischen als missbilligt (*deprecated*) gekennzeichnet. Analog existiert ein Ereignis „uninstalled“, welches im Rahmen der Deinstallation eines Moduls im laufenden System auftreten kann. Eine entsprechend verfeinerte Variante des Lebenszyklusnetzes ist in Abbildung 3.15) dargestellt. Die blassgrau dargestellten Kanten, Stellen und Transitionen sind den Modulen gegenüber nicht sichtbar und werden von der Plattform gesteuert. Wie in Unterabschnitt 4.4.2.1 erläutert stehen flexible Kanten für Übergänge, welche die Plattform für alle Module zeitgleich durchführt. [Vgl. Net, API-Dokumentation Version 6.0]

Die Komplexität des vervollständigten Netzes hat zwei Ursachen: Zum einen ist der Modul-Lebenszyklus nur indirekt dokumentiert und scheint somit nicht klar definiert zu sein. Zum anderen stellen die Aufrufe „validate“ und „closing“ eine Besonderheit dar – sie treten jeweils vor einem eigentlichen Ereignis wie z. B. „installed“ oder „close“ auf und ermöglichen jedem Modul, den Vorgang abzubrechen. Im Falle von „validate“

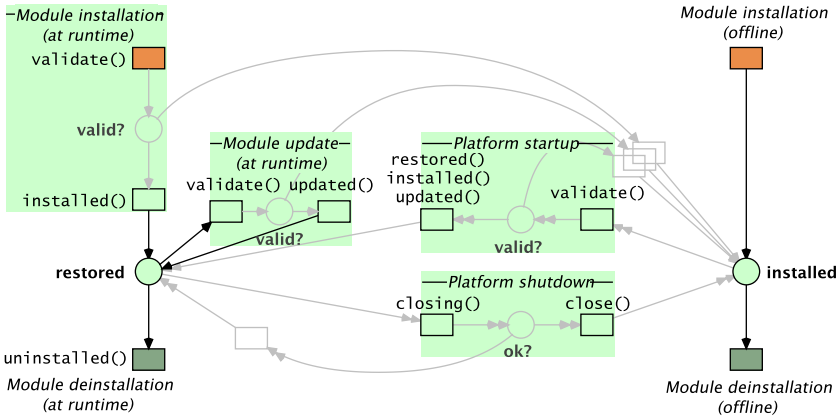



Abbildung 3.15: Der vervollständigte Lebenszyklus von NetBeans-Modulen

wird nur die Aktivierung des ablehnenden Moduls ausgesetzt – so kann z. B. auf nicht erfüllte Abhängigkeiten reagiert werden. Verweigert ein Modul jedoch das Ereignis „closing“, so wird das Herunterfahren der gesamten Plattform abgebrochen.

Die NetBeans-Benutzungsoberfläche integriert einen Modul-Manager, über den Module installiert und deinstalliert sowie aktiviert und deaktiviert werden können. Die Abbildung dieser Aktionen auf den Modul-Lebenszyklus ist leider nicht dokumentiert. Dies mag damit zusammenhängen, dass die deklarativen Mechanismen zur Spezifikation von Diensten und Erweiterungen eines Moduls (z. B. über das virtuelle Dateisystem) nicht an den Modul-Lebenszyklus gekoppelt sind. Tatsächlich wird in der NetBeans-Dokumentation dazu geraten, möglichst von der Verwendung der Modul-Lebenszyklusereignisse abzusehen.

**Richtlinien** Die Autoren von [BTW07] stellen – ähnlich wie die Autoren von [GB03] – einige Richtlinien zusammen, welche für das Implementieren erweiterbarer Systeme geeignet sind. Hierunter fallen zum einen Empfehlungen, auf bestimmte Entwurfsmuster zurückzugreifen, die der Entkopplung von Modulen dienlich sind (vgl. Abschnitt 2.4). Hierzu zählen insbesondere die Verwendung des `Lookup`-Konzepts und des virtuellen Dateisystems.

Weitere Ratschläge behandeln die Aufteilung von Code auf Module. Generell raten die Autoren dazu, Module nicht nur als Einheiten zusammenhängender Anwendungsfunktionalität zu betrachten. Module dienen ihrer Ansicht nach der Bändigung der Komplexität verteilt entwickelter Software: Modulgrenzen reduzieren die Größe der von den Entwicklern zu berücksichtigenden Beziehungen, bremsen die schleichende Vermaschung der Abhängigkeitsbeziehungen im Code und erlauben die Analyse oder das Validieren eingeschränkter Funktionalitätsbereiche.

Die Autoren empfehlen beispielsweise, API-Spezifikation und Implementierung auf verschiedene Module aufzuteilen. So werden alternative Implementierungen möglich,

außerdem können Spezifikation und Implementierung individuell versioniert werden. Zyklische statische Abhängigkeiten werden von den Autoren nicht befürwortet und von der NetBeans-Plattform nicht zugelassen. Für zwei häufige Fälle zyklischer Modulbeziehungen halten die Autoren Lösungsvorschläge parat: *Funktionale Abhängigkeiten* werden nicht bei der Installation von Modulen, sondern erst dynamisch zum Zeitpunkt der Aktivierung von Modulen überprüft. So kann z. B. die Anforderung, dass zu einer API-Spezifikation auch eine Implementierung existieren muss, als funktionale Abhängigkeit realisiert werden. *Brücken* (*bridges*) sind Module, welche die Funktionalität zweier anderer Module miteinander verbinden, indem sie sich bei beiden als Erweiterung einbinden. Eine Brücke löst zyklische Beziehungen zwischen zwei Modulen auf, ohne dass deren Code statisch miteinander verzahnt werden muss. [Vgl. BTW07, Kapitel 3 und Anhang A][29]

### 3.3.3 Client Application Platform .NET (CAP.NET)

Basierend auf einer Diplomarbeit präsentieren Wolfinger et al. in [WDPM06] eine Plugin-Architektur für die .NET-Umgebung namens CAP.NET. Das vorgestellte System beinhaltet nicht nur eine grundlegende Plugin-Verwaltung, sondern auch eine Architekturvorgabe für eine grafische Benutzungsschnittstelle. Letztere ist für die Betrachtung im Rahmen dieser Arbeit aber nicht relevant.

Die CAP.NET-Plugin-Architektur ist an die klassische Eclipse-Plugin-Architektur angelehnt. Plugins können *Extension Slots* und zu solchen Slots passende *Extensions* deklarieren. Eine Extension kann zusätzliche, vom Slot vorgegebene Parameter mit Werten belegen, um die Integration der Extension in den Slot zu beschreiben. Anstelle einer separaten XML-Beschreibung der deklarierten Slots, Extensions und Parameter, wie sie in Eclipse vorgesehen ist, werden .NET-Attribute im Quellcode der Plugins eingesetzt. Plugins werden in Form von .NET-*Assemblies* ausgeliefert. Die Attribute lassen sich zur Laufzeit aus der Plugin-Bibliothek extrahieren, ohne den Code der Bibliothek zu laden, so dass auch hier – wie bei Eclipse – ein verzögertes Laden möglich ist. Plugin-Bibliotheken in einem definierten Verzeichnis werden anhand dieser Attribute automatisch zur Startzeit des Systems erkannt und in eine zentrale Plugin-Datenbank (*Registry*) eingetragen. Ein Entfernen oder Austausch von Plugins zur Laufzeit ist nicht möglich, da die .NET-Umgebung das Entfernen von Bibliotheken nur in getrennten Speicherbereichen unterstützt und die dadurch notwendige Verwendung von entfernten Methodenaufrufen (*Remote Procedure Call, RPC*) eine große Performance-Einbuße darstellen würde.[30] [Vgl. WDPM06, Abschnitte 4 und 6]

Im Rahmen ihrer Arbeit geben Wolfinger et al. einige Konzeptbeschreibungen, die einen höheren Abstraktionsgrad als die in den Unterabschnitten 3.2.1.1 und 3.1.1.1 wiedergegebenen Versuche aus dem Netscape/Mozilla- und Eclipse-Umfeld aufweisen. Sie

[29] Die hier diskutierten Lösungsvorschläge zur Vermeidung zyklischer Abhängigkeiten von Boudreau et al. sind bereits im Unterabschnitt 2.4.4 unter den Namen *Separate API Plugin* und *Bridge Plugin* kurz vorgestellt worden, da sie sich als Entwurfsmuster zur Dekomposition von Plugins betrachten lassen.

[30] Die Unzulänglichkeiten der .NET-Umgebung im Hinblick auf das dynamische Laden und Entfernen von Code werden unabhängig auch von Escoffier et al. in [EDH06] diskutiert. Anlass der Analyse von Escoffier et al. ist der Versuch, eine der OSGi-Architektur (siehe Unterabschnitte 2.2.3 und 3.2.1) ähnliche Plattform für .NET zu entwickeln.

ordnen dabei Plugin-Systeme als erfolgreichen Entwicklungsschritt auf dem Weg zur Marktreife komponentenbasierter Systeme ein, daher verwenden sie die Begriffe *Plugin* und *Komponente* ohne eine unterscheidende Definition (siehe Unterabschnitt 2.2.5).

Eine *Plugin-Plattform* zeichne sich nach Wolfinger et al. dadurch aus, dass sie es Komponenten ermöglicht, sich zur Start- oder Laufzeit in einen Anwendungskern einzuklinken (*to plug in*). Die Plattform müsse zu diesem Zweck Mechanismen zum Einklinken (*Plugging*) und zum Auffinden (*Discovery*) der Auslieferungseinheiten (*Units of Deployment*) bereitstellen. Der *Plugging*-Mechanismus definiert die bereits erwähnten festen Regeln zur kontrollierten Integration von Erweiterungen in das System. Ein Beispiel dafür ist die Beziehung zwischen *Extension Slot* und *Extension*, welche den deklarierenden Komponenten die Rollen *Extension Host* (Anbieter des Slots) und *Extension Contributor* (Anbieter der Extension) zuweist und als alleinige Beschreibung zur Integration des Systems ausreicht. Das *Deployment* der Plugin-Komponenten wird durch die bereits erwähnte Anwendungsähnlichkeit der Komponenten geprägt – eine Komponente kann mehrere Slots und Extensions bereitstellen, um inhaltlich zusammenhängende Funktionalität anzubieten. Der *Discovery*-Mechanismus schließlich sorgt dafür, dass die Installation und Aktivierung von Plugin-Komponenten einfach und ohne fehleranfälligen Konfigurationsaufwand möglich ist. Typischerweise wird ein Plugin lediglich in ein Repository eingefügt oder wieder daraus entfernt, um es zu aktivieren bzw. zu deaktivieren – die Plugin-Plattform integriert automatisch alle Plugins im Repository. Neben diesen drei essentiellen Mechanismen könne eine Plugin-Plattform auch das dynamische Einklinken zur Laufzeit (*hot plugging*), das automatisierte Aktualisieren installierter Komponenten (*auto-update*) oder das Einkapseln potentiell sicherheitsgefährdender Plugins in eine eingeschränkte Ausführungsumgebung (*sandboxing*) anbieten. [Vgl. WDPM06, Abschnitt 2]

Eine Diskussion der eben vorgestellten Definitionen erfolgt in Abschnitt 4.1 in Zusammenhang mit der dort erfolgenden Konsolidierung des Plugin-Begriffs.

### 3.3.4 MagicBeans

Chatley beschreibt in seiner Dissertation [Cha05][31] das Java-basierte Plugin-Rahmenwerk MagicBeans, welches auf einem formalen Modell der Struktur und des Verhaltens von Plugin-Systemen basiert. Das MagicBeans-Rahmenwerk implementiert ein selbstorganisierendes Plugin-System, wobei die Herstellung und Auflösung von Verbindungen zwischen den hinzugefügten oder entfernten Plugins vollständig der Kontrolle der zentralen Verwaltung unterliegt. Mit Hilfe des formalen Modells werden beim Hinzufügen einer Komponente durch die zentrale Verwaltung mögliche Verbindungen zwischen allen Plugins im System ermittelt, wobei etwaige im formalen Modell formulierte Einschränkungen berücksichtigt werden. Die Verwaltung etabliert dann eine maximale Menge möglicher Verbindungen ohne Einflussnahme durch Nutzer oder Komponentencode.

Chatley sieht Plugin-Systeme als Variante der komponentenbasierten Systeme (siehe Unterabschnitt 2.2.5). Um die Begriffe aus der CBSE bruchlos übernehmen zu

[31] Der Dissertation von Chatley sind mehrere Workshop- und Konferenzbeiträge vorangegangen [z. B. CEM03, CEM04, CEK+04], welche einzelne Aspekte der Arbeit beleuchten. Der Autor empfiehlt aber die Dissertation als *die* Referenz auf seine Arbeit [siehe Cha05, S. iii].

können, nennt er einen Erweiterungspunkt *Anforderung* (*requirement*), obwohl – wie er selber betont – in Plugin-Systemen keine Notwendigkeit der Verbindung mit einer passenden Erweiterung, von Chatley *Angebot* (*provision*) genannt, besteht [vgl. Cha05, Abschnitt 3.1, S.35f.].

**Java-Rahmenwerk** Im Gegensatz zu vielen anderen hier vorgestellten Java-basierten Plugin-Systemen, welche die Bereitstellung mehr oder weniger umfangreicher Metadaten in Form von Manifesten vorsehen, setzt MagicBeans ausschließlich auf die Ausdrucksmöglichkeiten der verwendeten objektorientierten Programmiersprache Java. Das selbstorganisierende System nutzt den Java-Reflection-Mechanismus, um Anforderungen und Angebote einer Komponente zu identifizieren. Ein Angebot wird durch eine Klasse definiert, die ein öffentliches Java-Interface implementiert [vgl. Cha05, Abschnitt 4.2, S.58f.]. Eine Anforderung wird durch ein Objekt definiert, das eine Callback-Methode entsprechend dem Entwurfsmuster *Dependency Injection* (siehe Unterabschnitt 2.4.2) enthält. Der Methodenparameter muss durch ein öffentliches Java-Interface typisiert sein und ein Speicherplatz für die übergebene Objektreferenz im eigenen Objekt vorhanden sein[32] [vgl. Cha05, Abschnitt 4.3, S. 59f.]. Das Definieren von passenden Angeboten zu Anforderungen findet also direkt über den von Java bereitgestellten Namensraum der öffentlichen Interfaces statt.

**Formales Modell** Dem MagicBeans-Rahmenwerk liegt ein algebraisch spezifiziertes Modell der Struktur eines Plugin-Systems zugrunde. Das Modell unterscheidet zunächst Interfaces und implementierende Klassen, darauf aufbauend Komponenten sowie Verbindungen (*binding*). Eine Komponente umfasst Anforderungen (durch Interfaces definiert) und Angebote (durch Klassen definiert). Eine Verbindung zwischen zwei Komponenten ordnet einer Anforderung der einen Komponente ein Angebot der anderen Komponente zu. Verbindungen zwischen einer Anforderung und einem Angebot derselben Komponente werden explizit unterbunden. [Vgl. Cha05, Abschnitt 3.2.1, S. 40ff.]

Das Modell erlaubt unterschiedliche Beschränkungen der Multiplizität von Anforderungen und Angeboten. Eine Anforderung erlaubt zunächst die Verbindung mit unbeschränkt vielen passenden Angeboten. Sie kann alternativ mit einer individuellen Kardinalität modelliert werden, so dass die maximale Zahl verbundener Angebote beschränkt wird [Cha05, Abschnitt 3.2.4, S.46ff.]. Um die Multiplizität von Angeboten zu beschreiben, kann im Modell zwischen verschiedenen globalen Verbindungsrichtlinien (*binding policy*) gewählt werden. Eine Verbindungsrichtlinie legt z. B. fest, dass jedes Angebot nur an exakt eine Anforderung gebunden werden kann, eine alternative Richtlinie erlaubt die Bindung an beliebig viele Anforderungen. Weitere Richtlinien können nach Wünschen des Anwendungskonstrukteurs vorgegeben werden.

Das Modell unterstützt die drei Operationen Hinzufügen, Ersetzen und Entfernen von Komponenten, die jeweils aus einer algebraisch gegebenen Konfiguration eine neue

[32] Der Umstand, ob die übergebene Objektreferenz in einem skalaren Feld aufbewahrt oder einer `Collection`-Struktur hinzugefügt wird, definiert die Kardinalität der Anforderung, also die maximal zulässige Zahl an Angeboten, die diese Anforderung aufnehmen kann. [Vgl. Cha05, Abschnitt 4.3.1, S. 62f.]

berechnen. Das formale Modell ist zunächst in der Spezifikationssprache *Alloy* [All] verfasst, so dass automatisiert Konfigurationsbeispiele konstruiert werden können. Eine Vorgabe eines expliziten Anfangskonfiguration oder die Beschreibung von Schnittstellenverhalten ist nicht möglich. Chatley stellt daher eine weitere, in der Architekturbeschreibungssprache *Darwin*$^{i}$ (siehe Unterabschnitt 2.3.4) verfasste Variante des Modells vor, die auch die Beschreibung von Schnittstellenverhalten mittels einer einfachen Prozessalgebra erlaubt. Als Architekturbeschreibungssprache bietet Darwin auch eine grafische Darstellung der Komponenten und ihrer Verbindungen. Zudem steht im Darwin-Modell eine höhere Ausdrucksstärke zur Spezifikation von Einschränkungen für zulässige Systemkonfigurationen zur Verfügung. [Vgl. Cha05, Kapitel 5].

**Bewertung** Die MagicBeans-Rahmenwerk bietet dem Komponentenentwickler einen sehr schlanken Ansatz. Der zur Verwaltung erforderliche Code in jeder einzelnen Komponente umfasst lediglich die oben beschriebenen Dependency-Injection-Methoden. Durch die direkte Verwendung der Java-Interfaces zur Komponentenschnittstellendeklaration entfällt die explizite Angabe von Metadaten. Der Preis dafür ist der Verzicht auf einen verzögerten Lademechanismus, wie ihn z. B. Eclipse propagiert (siehe Unterabschnitt 3.2.1.4). Vielmehr müsste MagicBeans alle öffentlichen Klassen einer Komponente *sofort* zum Zeitpunkt ihrer Integration ins Plugin-System laden, um alle Anforderungen und Angebote identifizieren zu können.[33]

Der Verzicht auf Metadaten bedeutet auch einen Verzicht auf explizit deklarierte statische Abhängigkeiten, so dass jede Komponente in sich vollständig sein muss, um geladen werden zu können. Dabei klärt Chatley aber nicht, welcher Komponente er ein öffentliches Java-Interface zuschlägt, das schließlich von mehreren Komponenten zur Definition von Anforderungen und Angeboten genutzt werden muss. Die Lösung, das Interface mit allen Komponenten auszuliefern, die es benötigen, führt zur in Unterabschnitt 3.3.8 besprochenen Problematik des dynamischen Bindens mit möglichen Namens- und Versionkonflikten. In Kombination mit separaten Classloadern für jede Komponente würde diese Lösung zudem zur Problematik der sich überdeckenden Sichtbarkeitsbereiche (siehe Unterabschnitt 3.2.1.2) führen, so dass die Kopien des Interfaces nicht mehr als identisch erkannt werden würden.

Die zentrale Plugin-Verwaltung des MagicBeans-Systems behält auch beim Entfernen von Komponenten aus dem System die volle Kontrolle über die Verbindungen zwischen den Komponenten. In vielen anderen dynamischen Plugin-Systemen wird ein Plugin über seine Entfernung informiert, anschließend setzt die Verwaltung auf dessen Kooperation beim Auflösen aller von ihm ausgehenden Verbindungen. In MagicBeans werden niemals direkte Objektreferenzen zwischen Komponenten ausgetauscht – stattdessen richtet die Verwaltung für jede übergebene Referenz ein Proxy-Objekt ein. Dieses Objekt leitet zwar die Kommunikation ungehindert weiter, erlaubt aber die Kontrolle der Lebensdauer der Referenz durch die zentrale Verwaltung [vgl. Cha05,

[33] Zu welchem exakten Zeitpunkt das in Java implementierte System die vorhandenen Anforderungen und Angebote einer Komponente tatsächlich ermittelt, lässt Chatley leider offen. Seine Diskussion der „rückwirkenden Benachrichtigung“ einer Klasse über ihre Verbindungen lässt vermuten, dass das System die Anforderungen und Angebote einer Komponente erst Stück für Stück beim Instantiieren einzelner Objekte auswertet. [Vgl. Cha05, Abschnitt 6.6.1, S. 112f.]

Abschnitt 6.6.4, S. 115f.]. Der Preis für diese Ausweitung der Kontrolle ist offensichtlich eine gewisse Performance-Einbuße, die durch das Weiterleiten entsteht.

Das im formalen Modell fixierte Verbot von Verbindungen zwischen Anforderung und Angebot einer einzigen Komponente stellt einen interessanten Widerspruch zur im Eclipse-System propagierten *Fair Play Rule* [GB03, S. 66], also dem Entwurfsmuster des *Self-extending Plugin* dar (siehe Unterabschnitte 3.2.1.8 und 2.4.4). Leider begründet Chatley das Verbot von mit sich selbst verbundenen Komponenten nicht inhaltlich, sondern behauptet nur, dass solche Konfigurationen unerwünscht seien [vgl. Cha05, S. 54]. Die Einführung von Kardinalitäten gilt im Modell nur für Anforderungen – es sieht keine Kardinalitäten für Angebote vor. Die stattdessen vorgeschlagenen globalen Verbindungsrichtlinien bieten keine direkte Möglichkeit, die Kardinalität individueller Angebote unterschiedlich zu definieren. Somit kann im Modell nicht individuell definiert werden, ob ein Angebot eine exklusive, lokale Erweiterung oder einen breit verfügbaren Dienst beschreibt. Ein Grund für diese Asymmetrie zwischen Anforderungen und Angeboten wird von Chatley leider nicht genannt.

In den von Chatley spezifizierten formalen Modellen wird, da sie in einer funktionalen Sprache abgefasst sind, immer nur eine Konfigurationsänderung zur Zeit ausgeführt. Ob die dadurch implizierte Atomizität von Hinzufüge- und Entfernungsoperationen auch in der realen Java-Implementierung sichergestellt ist, thematisiert Chatley leider nicht. Zumindest fordert er, dass eine Komponente nur aus dem System entfernt werden kann, wenn sie gerade keine Dienstleistung gegenüber anderen Komponenten aktiv erbringt. Zu diesem Zweck richtet das MagicBeans-System zu jedem Proxy-Objekt einen Monitor ein, der sowohl für die Interkomponentenkommunikation als auch für das Entfernen der Komponente betreten werden muss [vgl. Cha05, Abschnitt 6.3, S. 104ff.]. Der Beschreibung ist allerdings nicht zu entnehmen, wie das System Verklemmungen vermeidet, wenn beim Entfernen einer Komponente, die an mehreren Verbindungen beteiligt war, mehrere Monitore betreten werden müssen.

### 3.3.5 Plugins und Multiagentensysteme

In Multiagentensystemen werden Plugin-Systeme als pragmatische Lösung zur Einführung von Flexibilität und Erweiterbarkeit im jeweiligen System verwendet.[34] Damit unterscheidet sich die Betrachtungsweise von Plugin-Systemen im Bereich der AOSE nicht von der vieler erweiterbarer Softwaresysteme, die mit anwendungsspezifischen Plugin-Schnittstellen ausgestattet sind. Es gibt viele Beispiele für Agentenplattformimplementierungen, in denen die Implementierung von Agentenverhalten, Ontologien oder Interaktionsmustern über eine Plugin-Schnittstelle in das MAS integriert wird.

JADE [BRP+] lässt sich beispielsweise beim Aufruf parametrisieren, so dass Klassen mit weiterer Funktionalität eingebunden werden können.[35] Seit Version 3.4 bietet JADE auch einen einfachen Mechanismus, der Agentencode in JAR-Dateien finden

[34] Die Konzepte von Multiagentensystemen (MAS) und ihre Verwendung im Bereich der agentenorientierten Softwareentwicklung (AOSE) sind das Thema des Abschnitts 2.5.

[35] Einige JADE-Erweiterungen wie z. B. das „JADE-Security-Plugin“ tragen zwar die Bezeichnung „Plugin“, werden aber durch explizite manuelle Veränderung des Programmaufrufs integriert. Bei diesen „Plugins“ handelt es sich daher eigentlich um nachträglich eingebrachte Modifikationen der installierten Codebasis, die nicht im Rahmen eines von der Kernanwendung vorgesehenen Plugin-Systems automatisch integriert werden.

und laden kann („JAR Agent“), wenn diese Dateien in einem speziellen Verzeichnis abgelegt sind. Dieser Mechanismus lässt sich als Plugin-System bezeichnen, das ein zum Programmstart erweiterbares System ermöglicht. [Vgl. BRP$^+$, JADE Administrator's Guide]

Capa läuft ohnehin im Rahmen des in dieser Arbeit entwickelten Plugin-System des Petrinetzsimulators Renew und kann darauf auch für eigene Belange zurückgreifen. Insbesondere können die in den Abschnitt 2.5.2 vorgestellten Programmierelemente wie Protokollnetze oder Wissensbasisinhalte von Plugins bereitgestellt werden. Zudem definiert Capa Erweiterungspunkte für Ontologien, Inhaltssprachen, Nachrichtenkodierungen oder Nachrichtentransportprotokolle [vgl. Duv02, Abschnitt 5.3.5].

Die Implementierung des MadKit [GFM] folgt einer vollständig erweiterbaren Softwarearchitektur. Hier können Plugins das System insbesondere um zusätzliche Programmier- oder Skriptsprachen erweitern, die dann zur Spezifikation oder Implementation von Agenten zur Verfügung stehen.

Ein etwas weiter gehendes Beispiel aus dem agentenorientierten Bereich ist das Verhandlungsrahmenwerk von DynamiCS [TGML98, TSL99]. In diesem Rahmenwerk können Agenten dynamisch aus Plugins komponiert werden, wobei verschiedene Arten von Plugins unterschieden werden: *Rollen* stellen eine Schnittstelle sowie deren Implementierung für einen Agenten bereit. Dabei kann es sich um grundlegende Fähigkeiten wie die allgemeine Kommunikationsfähigkeit handeln, aber auch um Rollen eines Interaktionsmusters (z. B. Käufer oder Verkäufer). *Substitute* stellen zu einer vorgegebenen Schnittstelle eine alternative oder ergänzende Implementierung bereit. *Konfigurationen* stellen weder Schnittstelle noch Implementierung bereit, sondern steuern andere Plugins, indem Sie deren Parameter manipulieren. Die Software-Architektur der Agenten umfasst also mehrere Systemkomponenten. Insbesondere Substitute erfüllen den Plugin-Begriff im Sinne der bisher vorgestellten Arbeiten, da sie an Schnittstellen (Erweiterungspunkte) anderer Plugins (erweiterbarer Komponenten) angeschlossen werden.

Das „E2“ Plugin-System [VT05] baut auf der klassischen Eclipse-Plugin-Architektur (vgl. Abschnitt 3.2) auf und erlaubt das dynamische Komponieren von Agenten aus lokalen Plugins und verteilten Diensten. Bei der Komposition der Plugins wird nicht nach domänenspezifischen Konzepten wie Verhalten, Ontologie oder Interaktion unterschieden. Der Komposition zugrunde liegt die in der Eclipse-Architektur modellierte Erweiterungsbeziehung – über eine grafische Benutzungsschnittstelle können zueinander in Erweiterungsbeziehung stehende Plugins ausgewählt und als Agent instantiiert werden. Ein Agent *besteht* demnach aus einer Gruppe von Plugins, statt das sich eine generische Agentenimplementierung der von den Plugins bereitgestellten Verhaltensweisen bedient. Die Integration findet über zwei zentrale, generische Extension Points statt, welche als Extensions deklarierte API-Methoden und Dienstleistungen der Agenten-Plugins entgegennehmen.

Zur Funktionalität der mitgelieferten Kern-Plugins des E2-Systems gehören eine generische grafische Benutzungsschnittstelle, in der die Fähigkeiten der Agenten als Ereignisse betrachtet werden, die sich nach dem aus Unix-Kommandointerpretern bekannten Prinzip der *Pipe* miteinander verketten lassen, sowie ein Transportsystem für Agentennachrichten. Das E2-System nutzt die fortgeschrittenen Möglichkeiten der

OSGi-basierten Eclipse-Laufzeitumgebung nicht, sondern implementiert selber eine dynamische Plugin- und Erweiterungsverwaltung.

Theoretisch eignen sich agentenorientierte Konzepte hervorragend zur Modellierung und Implementierung von Plugin-Architekturen. Nach Kenntnisstand des Autors dieser Arbeit hat sich aber bisher noch kein Plattformentwickler und auch keine Forschergruppe an der Verbindung beider Konzepte in Gegenrichtung zu den eben vorgestellten Ansätzen versucht. Obwohl die Anwender von Agentenplattformimplementierungen selber wieder Entwickler sind, wie es auch bei IDEs wie Eclipse und NetBeans der Fall ist, wird der Kreis zur Verwendung des eigenen Softwareentwicklungsparadigmas als Grundlage für die Erweiterbarkeit der Plattformfunktionalität (also die Verwendung von Agenten als konzeptionelle Grundlage für Plugins) bisher nicht geschlossen.

### 3.3.6 Integrieren von Modulsystemen

Die bisher vorgestellten Plugin-Systeme, Modulsysteme oder Komponentenmodellimplementierungen bieten Softwareentwicklern zwar jeweils ein Rahmenwerk, welches die Integration erweiternder Funktionalität in komplexe Anwendungen ermöglicht. Aber die Integrierbarkeit eines Plugins, Moduls oder einer Komponente ist naturgemäß auf Anwendungen beschränkt, die ebenfalls auf demselben Rahmenwerk aufbauen. Aus Sicht eines Entwicklers von allgemein nützlichen Erweiterungen wäre es aber wünschenswert, die eigene Funktionalität in mehrere Anwendungen bzw. Rahmenwerke integrieren zu können.

So erläutert beispielsweise der Online-Artikel [Buf04] sehr implementierungsnah, wie ein Plugin implementiert werden kann, das in die konkurrierenden Java-basierten IDEs Eclipse (siehe Abschnitt 3.2) und NetBeans (siehe Unterabschnitt 3.3.2) integrierbar ist. Beide Plattformen verwenden unterschiedliche Konventionen, wie die angebotene Funktionalität in den jeweiligen Manifesten deklariert werden soll. Diese Deklarationen können im Plugin einfach gleichzeitig nebeneinander vorgenommen werden. Ein größeres Problem stellt die Integration in die grafischen Benutzungsoberflächen beider IDEs dar, da nicht nur individuelle Klassenhierarchien für IDE-Konzepte wie z. B. Aktionen, Optionen oder Sichten existieren, sondern auch die zugrunde liegenden GUI-Bibliotheken (SWT bei Eclipse, AWT/Swing bei NetBeans) keinen gemeinsamen Nenner haben. Ein Entwickler hat hier im Wesentlichen zwei Möglichkeiten: Entweder entscheidet er sich für eine Klassenbibliothek und führt die Integration in der anderen IDE über Adaptercode durch. Oder er trennt strikt zwischen plattformunabhängiger Funktionalität und IDE-spezifischer GUI-Implementierung und erstellt von letzterer zwei alternative Varianten.

Gruber und Hall präsentieren in [GH06] ein Java-basiertes Rahmenwerk, das sich zur Integration vieler Plugin- oder Modulsysteme wie z. B. Eclipse/OSGi oder NetBeans eignet. Sie definieren Abstraktionen für die Modulsysteme, die Moduldefinitionen und die Module selber. Diese Abstraktionen werden von den einzelnen Modulsystemimplementierungen konkretisiert, so dass jedes Modulsystem mit seinen eigenen Mechanismen eine Moduldefinition zu einem Modul instantiieren kann. In den Definitionen als *Imports* und *Exports* deklarierte Abhängigkeiten zwischen Modulen werden zunächst innerhalb eines Modulsystems und anschließend über Modulsysteme hinweg mittels Verbindungsobjekten (*wires*) aufgelöst, wobei je nach beteiligten Modulsys-

temen unterschiedliche Abhängigkeitsgranularitäten und Auflösestrategien verwendet werden können. Dynamische Systeme werden unterstützt, der Lebenszyklus von Modulen umfasst zwei wahrnehmbare Zustände (*resolved* und *unresolved*). Der Prozess der Abhängigkeitsauflösung erfordert zwar einen separaten Zwischenzustand (*resolving*) aller beteiligten Module, wird nach außen aber als atomar dargestellt. Eine Modellierung detaillierterer Lebenszyklen einzelner Modulsysteme sowie Systemspezifischer Erweiterungs- oder Dienstbeziehungen findet in dem Rahmenwerk nicht statt. Konkrete Implementierungsprobleme wie z. B. die Integration unterschiedlicher Klassenbibliotheken werden von Gruber und Hall nicht behandelt.

Dass die Integration eines Plugin-Systems in ein anderes keine einfache Aufgabe ist, dokumentiert schon der Stand der Entwicklung der Umstellung des Eclipse-Plugin-Systems auf das OSGi-Rahmenwerk als neue Grundlage (siehe Abschnitt 3.2). Während die Abbildung technischer Aspekte wie z. B. der Kontrolle der unterschiedlich definierten Lebenszyklen und Abhängigkeitsbeziehungen mit dem Erscheinen der Eclipse-Version 3.0 weitgehend abgeschlossen war, verläuft die Anpassung des Verhaltens existierender Plugins an die neuen Schnittstellenverträge bezüglich der dynamischen Umgebung schleppend. Eine Abbildung der zentralen Beziehungs-Metaphern aufeinander, also in diesem Fall der Eclipse-Erweiterungsbeziehung auf die OSGi-Dienstbeziehung, ist erst gar nicht vorgesehen. Stattdessen können neue oder weiter entwickelte Plugins zwischen beiden Beziehungen wählen und so auch Plugins, die ursprünglich für unterschiedliche Rahmenwerke entwickelt wurden, miteinander verbinden. Stück für Stück könnten so Plugins von der einen zur anderen Metapher wechseln. Dass es zu einer vollständigen Umstellung von der Eclipse-Erweiterungsbeziehung auf die OSGi-Dienstbeziehung kommen wird, erscheint aber unwahrscheinlich angesichts des Umfangs der bereits existierenden Erweiterungsschnittstellen in Eclipse.

### 3.3.7 Modularisierung und Erweiterbarkeit in Java

Als objektorientierte Programmiersprache bietet Java natürlich die Möglichkeit, erweiterbare Architekturen mit geeigneten Entwurfsmustern zu implementieren (siehe auch Abschnitt 2.4). Zusätzlich laufen Bemühungen, die Syntax und Semantik der Sprache sowie die Klassenbibliothek um spezielle Konstrukte zur Implementierung von erweiterbaren Systemen zu ergänzen.

In der Klassenbibliothek der Java Standard Edition [SDN] ist ab Version 6 die Klasse `java.util.ServiceLoader` enthalten, deren Verwendung zur Implementierung erweiterbarer Softwarearchitekturen in [O'C07] erläutert wird.[36] Die `ServiceLoader`-Klasse bietet die Möglichkeit, zu einer einmal definierten Dienstschnittstelle (*Service Provider Interface, SPI*) verschiedene Dienstanbieter-Implementierungen (*Service Provider*) über die Classloader-Hierarchie zu finden und zu instantiieren. Dem Auffinden der Dienstanbieter dienen einfache Konfigurationsdateien mit den qualifizierten Klassennamen von Schnittstelle und Implementierung. Dienstanbieter-Implementierungen werden erst bei Bedarf (*lazily*) instantiiert, d.h. wenn ein Dienstnutzer eine Instanz über den `ServiceLoader` anfordert.

[36] Eine der Klasse `java.util.ServiceLoader` entsprechende Funktionalität gab es als Java-interne, für Entwickler nicht zugängliche Klasse bereits seit der Java 2 Standard Edition, Version 1.3.

Da die Classloader zur Laufzeit andere Dienstimplementierungen als zur Übersetzungszeit des Dienstnutzers bekannt verwalten können, handelt es sich beim `ServiceLoader` um einen Erweiterungsmechanismus, der das Hinzufügen und Entfernen von Funktionalität ohne Neuübersetzung der Kernanwendung ermöglicht. Die Kopplung von Dienstnutzer und -anbieter kann flexibel zur Laufzeit hergestellt werden. Der Mechanismus kann in Plugin-Systemen verwendet werden, wenn er um geeignete Classloader ergänzt wird, welche das Finden und die Konfiguration der Plugins übernehmen. Der `ServiceLoader`-Mechanismus bietet allerdings keine Unterstützung *dynamischer* Plugin-Architekturen, da abgesehen von der Instantiierung des Dienstanbieters kein Lebenszyklus verwaltet wird. Zudem weist die Dokumentation der Klasse explizit darauf hin, dass ein `ServiceLoader`-Objekt nicht auf nebenläufige Verwendung ausgelegt ist.

Aktuell laufen mehrere Standardisierungsbemühungen unter dem „Java Community Process“ (JCP), um die unmittelbare Eignung der Sprache Java für modularisierte, erweiterbare Software wie z. B. Plugin-Systeme zu verbessern. Während der „Java Specification Request“ (JSR) Nr. 291 [JSR07] direkt die OSGi-Architektur (siehe Unterabschnitte 2.2.3 und 3.2) zur Integration in die Java-Klassenbibliothek vorschlägt, erarbeitet der JSR-277 [JSR06a] ein alternatives Datei- und Manifestformat nebst passender Klassenbibliothek zum Finden, Laden und Verknüpfen von Modulen zur Laufzeit. Im Vergleich zur OSGi-Architektur verfügt der erste Entwurf des JSR-277 [Gro06] zusätzlich über einen Mechanismus, um Module bei Bedarf aus Repositories heraus nachzuladen und zu installieren. Die vorgeschlagene Architektur beschränkt sich allerdings auf das Finden, Installieren, Laden, Verknüpfen und Instantiieren von Modulen nach Bedarf – sie erlaubt bisher kein dynamisches Entladen von Modulen, definiert keinen für die Module wahrnehmbaren Lebenszyklus und gibt kein Schema für Erweiterungs- oder Dienstleistungsbeziehungen vor. Der Artikel [Han08] stellt wesentliche technische Elemente der beiden Entwürfe einander gegenüber und verweist auf weitere Informationsquellen zur Diskussion des künftigen Java-Modulkonzepts.

Um die Kapselung von Modulen zu verbessern, erarbeitet der JSR-294 [JSR06b] eine Ergänzung der in den Klassen definierbaren Sichtbarkeitsbereiche für Methoden und andere Elemente, so dass neben der Klasse und dem Java-Package auch ein Modul als Sichtbarkeitsgrenze zur Verfügung stehen kann.

Die eben genannte Ansätze setzen auf einer „reinen“ Java-Umgebung auf. Daneben existieren diverse Rahmenwerke, welche der Modularisierung dienen. Zu diesen gehören beispielsweise das bereits im Rahmen des Eclipse-Plattform in Abschnitt 3.2 angesprochene OSGi-Rahmenwerk sowie die Implementierung des bereits in Unterabschnitt 2.2.2 angesprochenen Komponentenmodells der Enterprise Java Beans (EJB).

### 3.3.8 Verteilte Entwicklung

Plugin-Systeme sollen zur Laufzeit Code von verschiedenen Entwicklern in einer Anwendung integrieren. In der Regel arbeiten alle beteiligten Code-Entwickler nach individuellen Zeitplänen, in individuellen Umgebungen und verwenden individuelle Kombinationen von unterstützenden Code-Bibliotheken. Eine Plugin-fähige Anwendung wird daher de facto *verteilt entwickelt*. Auch wenn einzelne ihrer Komponenten vielleicht aus einem zentralen Entwicklungsprozess mit vollständiger Kontrolle hervorgehen, kann es

keine zentrale Kontrolle für das Gesamtsystem inklusive sämtlicher Plugins geben, da die Plugin-Schnittstelle(n) ein offenes System definieren. [Vgl. BTW07, Abschnitt 2.1, S. 11f.]

Im Forschungsgebiet der verteilten Softwareentwicklung werden vorrangig Koordinations- und Kommunikationsschwierigkeiten innerhalb eines zusammenhängenden Entwicklerteams betrachtet, das geographisch verteilt arbeitet [vgl. Tur03, SCS06]. Bei der verteilten Entwicklung von Plugin-basierten Anwendungen geht man aber nicht von einem zusammengehörigen Entwicklerteam aus. Vielmehr arbeiten an den einzelnen Plugins unabhängige Teams, die gar nicht (oder allenfalls gelegentlich zwecks Fehlerbehebung) unmittelbar kommunizieren, sondern lediglich mittels Aktualisierung von Komponenten der Anwendung Informationen austauschen.

Aus der Situation der unabhängigen verteilten Entwicklung ergeben sich mehrere technische und semantische Probleme, die zu unterschiedlichen Forschungsgebieten gehören. Daher sollen hier nur einige der Probleme kurz umrissen und Lösungsansätze skizziert werden.

Plugin-Systeme benötigen prinzipbedingt eine Technik, Code aus unterschiedlichen Quellen zur Laufzeit in eine Anwendung integrieren zu können. Beispielsweise greifen die Netscape-Plugins ebenso wie die Mozilla-Extensions auf den betriebssystemspezifisch implementierten Mechanismus dynamischer Bibliotheken zurück (vgl. Unterabschnitt 3.1.1.2). Alternativ bauen sowohl das klassische Eclipse-Plugin-System als auch das OSGi-Rahmenwerk auf dem programmiersprachenspezifischen Mechanismus der Java-Classloader auf (vgl. Unterabschnitt 3.2.1.2). Beide Mechanismen sind in der Lage, kompilierten Code mehrerer Module bzw. Klassen *dynamisch* zu *verknüpfen* (*Linking*) – d.h. erst zur Laufzeit statt beim Kompilieren des Codes.

Beiden Mechanismen zum dynamischen Linken ist die Eigenschaft gemein, dass die beim Kompilieren verwendete Version des Codes von der beim Verknüpfen vorliegenden Version abweichen kann. Dies stellt offensichtlich eine mögliche Fehlerquelle dar, da sich Schnittstellen, Verhalten und eventuell sogar Verträge des Codes seit den individuellen Übersetzungszeitpunkten der einzelnen Komponenten verändert haben können. In objektorientierten Sprachen sind diese Abweichungen besonders relevant, wenn die Superklasse oder ein Interface einer abgeleiteten Klasse aus einem anderen Modul stammt und sich verändert hat – Szyperski nennt dies das „*Fragile Base Class Problem*“ und unterscheidet zwischen einer syntaktischen und einer semantischen Variante [vgl. Szy02, Abschnitt 7.4, S. 115ff.].

Um die Problematik der syntaktischen Abweichungen zu erfassen und einzugrenzen, befassen sich diverse Arbeiten mit dem Thema der *Binärkompatibilität*. Im Kontext der Programmiersprache Java wurde diese Eigenschaft von Anfang an in der Sprachspezifikation definiert [GJSB05, Kapitel 13], in [LB98] weiter elaboriert und in [DWE98] formalisiert. Rivières gibt in [Riv07] einen Überblick für Entwickler von Java-Komponentenschnittstellendefinitionen, welche Arten von Änderungen an der Schnittstelle die Kompatibilität zu bereits existierenden, abhängigen Komponentenimplementierungen bewahren oder brechen. Alternativ bemühen sich diverse Ansätze, den Verknüpfungsmechanismus oder das Typsystem der Sprache anzupassen, um das Problem einzudämmen und mehr Flexibilität für dynamische Änderungen zu erreichen [z. B. And00, MPG+00, RC02, HT03, MHF06].

Semantische Änderungen an Schnittstellenverträgen bleiben dagegen immer ein Problem der Entwickler und Nutzer abhängiger Komponenten, das nur mit festen Regeln und Verträgen zwischen den beteiligten Personen eingedämmt werden kann. Abhängigkeiten zwischen Komponenten müssen daher in der Regel auf eine festgelegte Menge von Versionen der referenzierten Komponente beschränkt werden, innerhalb derer der Komponentenhersteller die Kompatibilität aller oder gar die Abwesenheit jeglicher Änderungen zusichert [Szy02, Abschnitt 6.10, S. 105ff.]. Hierzu wird aber ein festes Namens- und Versionierungsschema benötigt [vgl. MS02].

Bei Plugin-Systemen tritt daher das auch bei Komponentenmodellimplementierungen bekannte Problem auf, dass mehrere Komponenten verschiedene Versionen eines Moduls oder einer Klasse voraussetzen oder dass sie zu einem Modul-, Prozedur- oder Klassennamen verschiedene Implementierungen bereitstellen können. Insbesondere unter der Betriebssystemfamilie „Microsoft Windows“ ist dieses Problem als „DLL Hell“ bekannt, da Anwendungen häufig während ihrer Installationsprozedur bestehende Versionen einer dynamischen Bibliothek ersetzen und damit unbemerkt das Verhalten bereits installierter Programme verändern. Der Mechanismus der „Shared Objects“ unter Unix löst dieses Problem, indem mehrere Versionen einer Bibliothek im Dateisystem installierbar und anhand eines festgelegten Namensschemas unterscheidbar sind [vgl. Lev00, S. 216].

Findet das Verknüpfen mehrerer Komponenten in einem globalen Namensraum statt, entstehen möglicherweise Namenskonflikte, die nicht unmittelbar zu lösen sind. Sowohl das klassische Eclipse-Plugin-System als auch das OSGi-Rahmenwerk greifen zur Lösung der Namenskonflikte auf die vorhandenen Möglichkeiten der Sprache Java zurück. Eine ausgefeilte Classloader-Hierarchie definiert (wie in Unterabschnitt 3.2.1.7 erläutert) eine auf die als notwendig deklarierten Abhängigkeiten eingeschränkte Sichtbarkeitsbeziehung. Die mit der eingeschränkten Sichtbarkeit verbundenen Probleme sowie einige Lösungsansätze schildert [Lip05a].

Ein weitreichenderes Problemfeld ist allen dynamisch komponierbaren Systemen gemein, die Module aus zum Zeitpunkt des Kompilierens unbekannten Quellen mit unbekanntem Verhalten integrieren. Prinzipbedingt scheidet in solchen Systemen die Möglichkeit aus, das Gesamtsystem bereits beim Kompilieren zu integrieren und zu verifizieren. Die Vorteile statischer Typprüfung oder nachgewiesener Sicherheit stehen somit nur eingeschränkt auf einzelne Module zur Verfügung. [Vgl. Szy02, Abschnitt 6.7, S. 98ff.]

Das verteilte Entwickeln von Code stellt einen eigenen Problembereich dar, der nicht nur in Plugin-Systemen auftritt und seit langem aktiv beforscht wird. Das Problem ist aber für den Gegenstand dieser Arbeit – die nebenläufige Konzeption von Plugin-Architekturen – nicht von vorrangiger Bedeutung. Das im Kapitel 4 entwickelte petrinetzbasierte Modell arbeitet ohne statische Typsysteme ausschließlich mit dynamischen Kommunikationstechniken. Daher soll dieser Problembereich im Folgenden nicht weiter vertieft werden.

### 3.3.9 Betriebssysteme

Gelegentlich führen einige Autoren zur Illustration von Plugin-Konzepten eine zunächst überraschend erscheinende Analogie an, indem sie Betriebssysteme zum Vergleich heranziehen.

So spielt z. B. Szyperski in [Szy02, Abschnitt 6.6, S. 95ff.] auf die Ähnlichkeit von Betriebssystemen und Komponentensystemen an, da beides *erweiterbare* Systeme sind. Offensichtlich sind Betriebssysteme in der Lage, beliebige Anwendungen zu laden und auszuführen – sie bieten wie ein Komponentensystem eine einheitliche Kommunikationsschnittstelle und zentrale Basisdienste für die Anwendungen. Die Idee der Mikro-Kernel lässt sich ebenfalls als Plugin-Architektur deuten, da Basisdienste des Betriebssystems in Anwendungen ausgelagert und damit zu austauschbaren Erweiterungen des Kernels werden.

Sind nun die Anwendungen selber wieder erweiterbar, da z. B. mit einer Plugin-Schnittstelle versehen, so entsteht insgesamt ein mehrfach erweiterbares oder womöglich sogar rekursiv erweiterbares System. Szyperski weist allerdings darauf hin, dass ein exzessiv in möglichst kleine, spezialisierte Erweiterungen aufgespaltenes System zwar die Wiederverwendbarkeit der Teile und die Flexibilität des Gesamtsystems erhöht, dafür aber zu Lasten von Effizienz, Robustheit und Performanz des Gesamtsystems geht. [Vgl. Szy02, Abschnitt 6.6, S. 95ff.]

Boudreau et al. sprechen in [BTW07, Kapitel 2, S. 11ff.] ebenfalls die Ähnlichkeit von Betriebssystemen und verteilt entwickelten, modularen Anwendungen an. Insbesondere erlauben es Paketmanagementsysteme (z. B. RPM (**R**PM **P**ackage **M**anager, siehe [Bai00]) oder DPKG/APT (**D**ebian **P**ac**k**a**g**e Manager, **A**dvanced **P**ackaging **T**ool, siehe [Deb02]) unter Linux), die Ergebnisse der Arbeit verschiedener unabhängiger Entwickler koordiniert auf einem System zu installieren, verwalten und zu deinstallieren. Ein wesentliches Element dieser Paketmanagementsysteme ist die Beschreibung von Abhängigkeiten zwischen Softwarepaketen, so dass Pakete mit unerfüllten Abhängigkeiten entweder gar nicht oder aber inklusive aller Pakete, von denen es abhängig ist, installiert werden.

Marquardt zieht in [Mar06] mehrfach Betriebssysteme als Beispiel für Vorkommen einiger Entwurfsmuster heran, die im Kontext von Plugin-Systemen verwendet werden (vgl. Unterabschnitt 2.4.3). Konkret nennt er Gerätetreiber und Bildschirmschoner als Beispiele für Betriebssystem-Plugins [vgl. Mar06, S. 309]. Die Prozedur zum Laden von Programmcode und -daten sei ein Beispiel für einen Plugin-Vertrag (*plug-in contract*) und Mechanismen wie der Suchpfad für ausführbare Dateien ein Beispiel für das Muster der Plugin-Registrierung [vgl. Mar06, S. 315 u. S. 321].

Gelegentlich wird die zentrale Verwaltungseinheit eines Plugin-Systems, die für das Finden, Laden, Verknüpfen und Ausführen der einzelnen Plugins zuständig ist, auch als „Kernel“ bezeichnet [z. B. Bir05, S. 42f.]. Dieser Begriff entstammt dem Jargon der Betriebssysteme, so dass offensichtlich eine Parallele zu jenen hergestellt werden soll.

Die Betrachtung von Betriebssystemen im Kontext der Plugin-Systeme vermengt verschiedene Abstraktionsebenen elektronischer Systeme – typischerweise wird (außerhalb des Embedded-Bereichs) anwendungsspezifische Software auf einer Ebene oberhalb der Abstraktion entwickelt, welche ein Betriebssystem als allgemeine Ausführungsumgebung bietet. Nichtsdestotrotz lassen sich Parallelen im zugrunde liegenden

Mechanismus zur Realisierung eines flexibel einsetzbaren Systems finden. Wie bereits von den zuvor zitierten Autoren erkannt wurde, handelt es sich ebenso wie bei Plugin-Systemen um flexible, dynamisch erweiterbare Systeme.

Will man Betriebssysteme als Plugin-Systeme betrachten, so ist zunächst die Frage zu klären, wie hier ein System erweitert wird. Drei Beispiele sollen den grundlegenden Gedanken illustrieren: Die Schnittstelle zum Ausführen von Programmen, die Betrachtung der Mensch-Maschine-Interaktion als Kernfunktionalität und die Verwaltung von Programmpaketen. Diese Auswahl ist bei weitem nicht vollständig, aber sie reicht aus, um die Erweiterbarkeit von Betriebssystemen im Sinne des Plugin-Konzepts zu illustrieren. Jedes Betriebssystem bietet einen Mechanismus, Programme zu laden und als Prozess auszuführen, dabei wird die Kontrolle innerhalb des Betriebssystems vorübergehend über eine festgelegte Binärschnittstelle (*Application Binary Interface, ABI*, siehe auch Unterabschnitt 3.1.1.2) an das Programm übergeben. Jedes Programm ist somit eine Erweiterung des ausgeführten Betriebssystems, die auf Basis der Erweiterungsspezifikation (also der ABI) ins System eingeklinkt werden kann. Dies gilt insbesondere, wenn sich die ausgeführten Prozesse wiederum in die Betriebssystemfunktionalität einklinken, beispielsweise als Gerätetreiber. Eine weitere Spielart dieser Erweiterungsschnittstelle sind die in Unterabschnitt 3.1.1.2 vorgestellten Mechanismen zum Bereitstellen dynamisch nachladbarer Bibliotheken – so wird jede installierte Bibliothek zu einer Erweiterung des Gesamtsystems.

Betriebssysteme implementieren wesentliche Teile der Mensch-Maschine-Schnittstelle – sie legen die Architektur für Eingaben und Ausgaben fest und verarbeiten einen gewissen Anteil der Ein- und Ausgaben selber. Da wiederum jedes laufende Programm die Verarbeitungsmöglichkeiten für Ein- und Ausgaben erweitert, ist auch hier eine Erweiterungsschnittstelle zu sehen. Als Erweiterungsschnittstellen eines Betriebssystems sind also eben jene „Dienste“ des Betriebssystems zu sehen, die von Programmen oder Prozessen genutzt werden können, um im Rahmen des Gesamtsystems ihre Funktionalität zu erbringen.

Moderne Betriebssysteme werden (außer vielleicht im Embedded-Bereich) nicht mehr monolithisch inklusive aller Anwendungen kompiliert und auf einen Chip gebrannt, sondern im Form von flexibel zusammenstellbaren Paketen ausgeliefert, so dass ein Systemadministrator die für den jeweiligen Einsatzzweck notwendigen Pakete kombinieren kann. Paketverwaltungen wie das bereits genannte RPM oder DPKG/APT erlauben es zudem, Anwendungssoftware, zentrale Bibliotheken und weitere Ressourcen nach inhaltlich zusammenhängender Funktionalität gebündelt zu installieren bzw. zu deinstallieren, während das als Basis dienende Betriebssystem durchgehend läuft. Damit liegt eine – zwar oberhalb des eigentlichen Betriebssystemkerns definierte, aber doch zentrale – Lebenszyklusverwaltung für diese Pakete vor, die auch komplexe Abhängigkeiten zwischen den Paketen erfasst und kontrolliert.

Die Analogie zwischen Betriebssystemen und Plugin-Systemen wird allerdings zur Klärung der Begriffe und Konzepte von Plugin-Architekturen in dieser Arbeit nur einen sehr geringen Beitrag leisten. Die sehr generisch entworfenen Plugin-Systeme wie Mozilla-Extensions (siehe Unterabschnitt 3.1.2), Eclipse (siehe Abschnitt 3.2) oder NetBeans (siehe Unterabschnitt 3.3.2) integrieren jeweils viele Konzepte, die sich zur Übernahme aus dem Bereich der Betriebssysteme anbieten. Zusammengenommen sind alle in diesem Unterabschnitt zuvor angesprochenen Konzepte wie z. B. die allgemeine

Schnittstelle zum Laden und Ausführen, die Installation und Konfiguration von Komponenten über ein generisches Dateisystem oder die Verwaltung von Abhängigkeiten inhaltlich gebündelter Pakete, abgedeckt, da sie in dem einen oder anderen Plugin-System in ähnlicher Form auftreten. In der weiteren Arbeit werden Betriebssysteme daher nicht weiter detailliert betrachtet, sondern allenfalls hier und da zur Illustration herangezogen.

## 3.4 Diskussion

Die vielen und vielfältigen in diesem Kapitel vorgestellten Plugin-orientierten Softwaresysteme und verwandten Forschungsbereiche sollen nun noch einmal kompakt präsentiert und einander gegenübergestellt werden. Die Gegenüberstellung erlaubt es, gemeinsame und unterschiedliche Konzepte und Eigenschaften zu erkennen und gegebenenfalls zusammenzufassen oder voneinander abzugrenzen. Die Konzepte und Eigenschaften lassen sich zudem mit denen der im vorigen Kapitel diskutierten Softwareentwicklungsansätze vergleichen.

Während der Vorstellung der einzelnen Plugin-Architekturen wurde dem jeweils definierten globalen Lebenszyklus der Plugin-Verwaltung besondere Aufmerksamkeit zuteil. Um die mit unterschiedlichen Ausdrucksmitteln mehr oder weniger präzise definierten Lebenszyklen der Systeme vergleichbar darzustellen, wurden sie in Form von Petrinetzen modelliert. Die Modellierung von Plugin-Lebenszyklen ist der Diskussionsgegenstand des Unterabschnitts 4.4.2, so dass auch der Vergleich der vorgefundenen Lebenszyklusmodelle erst dann stattfindet.

### 3.4.1 Gegenüberstellung der Plugin-Systeme

Die in diesem Kapitel vorgestellten Softwaresysteme stellen nur eine sehr kleine Auswahl aus der Menge aller existierenden Anwendungen mit Plugin-Architektur dar. Insbesondere die Beschränkung auf Open-Source-Systeme sowie einige Arbeiten aus dem akademischen Umfeld führt zu einer im Vergleich zur Marktdominanz der jeweiligen Systeme nicht repräsentativen Gewichtung der Darstellungen. Dennoch sind allein schon die in den Anwendungen der Netscape/Mozilla-Familie und in der Eclipse-Plattform enthaltenen Plugin-Systeme dermaßen vielfältig, dass die in diesem Kapitel präsentierte Auswahl fast alle der im Spektrum der existierenden Plugin-Systeme vorkommenden Konzepte abdeckt.

Der anwendungsspezifische Kontext und die betriebssystemabhängige Implementierung der Netscape-Plugin-API (**NPAPI**, siehe Unterabschnitt 3.1.1) machen diese zum typischen Stellvertreter für die unzähligen anwendungsspezifischen Erweiterungsmechanismen aktueller Standardanwendungen. Ein solches Plugin-System ist der eigentlichen Anwendungsfunktionalität untergeordnet, es realisiert in erster Linie die Möglichkeit, Code-Module auch nach der Kompilation der Hauptanwendung (des Webbrowsers) noch in die Anwendung integrieren zu können. Die Installation von Plugins erfolgt betriebssystemabhängig und wird durch Werkzeuge des Plugin-Herstellers unterstützt; eine Installation oder Deinstallation zur Laufzeit der Anwendung ist nicht vorgesehen. Die Plugin-Schnittstelle ist anwendungsspezifisch definiert und dient der

Flexibilisierung oder Auslagerung einer abgegrenzten Funktionalität aus der Kernanwendung. Die Schnittstelle definiert eine asymmetrische Beziehung zwischen zwei Beteiligten, namentlich Anwendung und Plugin. Während die Anwendung die Schnittstelle definiert, den Lebenszyklus der Plugins steuert, Dienstleistungen zur Unterstützung des Plugins anbietet und zeitweise die Programmflusskontrolle an das Plugin übergibt, erbringt das Plugin eine spezielle Variante der durch die Schnittstelle definierten Funktionalität. Der Schnittstellenvertrag, der das Verhalten beider Beteiligten regelt, wird mit den üblichen Mitteln des praktizierten Softwareentwurfs definiert, also vor allem durch Dokumentation.

**Mozilla-Extensions** (siehe Unterabschnitt 3.1.2) können dagegen beliebige Funktionalität zur Anwendung hinzuzufügen, da sie gleichberechtigt auf der grundlegendenden Infrastruktur der Anwendung aufbauend implementiert werden. Die Anwendungsarchitektur stellt auf mehreren Schichten verschiedene Komponentenmodelle und Verwaltungen bereit, die verschiedene Begriffe und Konzepte zur Realisierung einer erweiterbaren Softwarearchitektur verwenden. Neben der klassischen NPAPI sind die XPCOM-Komponentenmodellimplementierung, die Chrome-Verwaltung und das XPInstall-Toolkit zu nennen.

Das XPCOM-Komponentenmodell stellt eine Objektverwaltung bereit, die in mehreren Programmiersprachen genutzt werden kann, von denen einige nicht objektorientiert sind bzw. deren integrierte Objektverwaltung nicht genutzt wird. Die Objektverwaltung erlaubt die Definition von Interfaces und Klassen, deren Instantiierung und die Implementierung von Objektreferenzen. Objekte sind eine daten- und funktionsorientierte Abstraktion. Die Installation der Klassendefinitionen geschieht programmgesteuert über flach organisierte Registries und wird in der Regel durch Verwaltungswerkzeuge der anderen Mozilla-Architekturebenen vorgenommen. Das korrekte Verhalten zur Verwaltung von Objekten wird durch eine Dokumentation für Entwickler spezifiziert, ebenso kann das gewünschte Verhalten zur Verwendung definierter Klassen dokumentiert werden.

Die Chrome-Paketverwaltung dient im Wesentlichen der Montage der grafischen Oberfläche der Anwendung aus mehreren Paketen. Chrome-Pakete stellen keine aktiv nutzbaren Dienste oder Erweiterungspunkte bereit, sondern steuern ein oder mehrere passive Beschreibungselemente und Dateien zur Anwendungsoberfläche bei, die im Rahmen der Funktionalität einer zentralen Kernanwendung (Gecko) komponiert und dargestellt wird. Die Verschmelzung der einzelnen Elemente zu einem Ganzen wird mittels identisch benannter Knoten oder Pfadangaben definiert. Aufgrund der Reaktivität der beigesteuerten Elemente ist kein definierter Lebenszyklus oder anderweitige Verhaltensspezifikation für Pakete erforderlich. Zur Laufzeit sind de facto keine gravierenden Änderungen an der einmal kombinierten Oberflächenbeschreibung mehr möglich.

Die XPInstall-Komponente bzw. das neuere Toolkit stellen als Teil der Mozilla-Anwendungsinfrastruktur wiederverwendbare Mechanismen zur Installation, Deinstallation und Aktualisierung von Paketen zur Verfügung. Die darauf aufbauenden Installationspakete gruppieren inhaltlich zusammenhängende XPCOM-, Chrome- oder auch NPAPI-Elemente zu einer Einheit. Die Pakete sind versionierbar, um eine automatische Aktualisierung zu ermöglichen. Zwischen verschiedenen Installationspaketen können allerdings nur sehr eingeschränkt Beziehungen in Form von Anforderungen an die

Installationsumgebung definiert werden, was in etwa der Deklaration statischer Abhängigkeiten entspricht. Alle Pakete – sowohl die Kernanwendung als auch Extensions – werden gleichberechtigt flach organisiert. Die im Toolkit enthaltene Softwareverwaltung erlaubt zwar die dynamische Installation, Aktualisierung oder Deinstallation von Paketen, de facto gelten aber die Einschränkungen der zugrundeliegenden Komponentenmodelle und Verwaltungen bezüglich der dynamischen Anpassung der Konfiguration.

Die drei Architekturen XPCOM, Chrome und XPInstall stellen jede für sich kein vollwertiges Plugin-System zur Verfügung, sie werden daher in dieser Arbeit nur als mögliche Teile einer Implementierung von Mozilla-Extensions betrachtet. XPCOM-Komponenten qualifizieren sich nicht für den Komponentenbegriff wie er in Unterabschnitt 2.2.1 eingeführt wurde. XPCOM-Komponenten stellen instantiierbare Klassen dar, zum Teil werden in der Mozilla-Community sogar nur die instantiierten Objekte als „Komponenten“ bezeichnet. Sie qualifizieren sich auch nicht für den in Unterabschnitt 4.1.9 definierten Plugin-Begriff, da sie nicht zwingend in einer Erweiterungsbeziehung stehen. Der reaktive Charakter von Chrome-Paketen erlaubt nur eine Betrachtung als Bausteine einer zentral akkumulierten Oberfläche, die Verknüpfung über benannte Pfade lässt eine scharfe Schnittstellendefinition vermissen. Die Installationsmechanismen von XPInstall beschreiben zwar eine Plugin-Verwaltung, definieren jedoch keine eigene Plugin-Schnittstelle. Letzteres überlassen sie den anderen Ebenen der Mozilla-Softwarearchitektur, so dass die Kombination als vollwertiges Plugin-System betrachtet werden kann, das aber keine einheitliche Architektur aufweist.

In der **Eclipse**-Plattform (siehe Abschnitt 3.2) ist das Plugin-Konzept als grundlegender Architekturstil gewählt worden, so dass jegliche anwendungsspezifische Funktionalität (auch die Kernfunktionalität) mittels Plugins bereitgestellt wird. Eine einheitliche und präzise Definition des Plugin-Konzepts lässt sich jedoch weder in der Dokumentation noch in Büchern zur Eclipse-Plattform finden. Um die Integration von vielen Plugins zu erleichtern und ein einheitliches Erscheinungsbild zu gewährleisten, werden Plugin-Entwicklern viele Richtlinien zur Einfügung ihres Produkts in die es umfassende Software-Architektur mitgegeben. Die klassische Implementierung des Eclipse-Plugin-Rahmenwerkes wurde mit Erscheinen der Version 3.0 durch eine auf dem OSGi-Rahmenwerk basierende Implementierung abgelöst, das Plugin-Konzept blieb aber erhalten.

Im klassischen Eclipse-Plugin-System stehen Plugins zueinander in einer Erweiterungsbeziehung, in der jedes Plugin sowohl mehrere Erweiterungspunkte als auch Erweiterungen zu mehreren anderen Erweiterungspunkten anbieten kann. Die Erweiterungsbeziehung impliziert eine statische Abhängigkeit des erweiternden Plugins vom erweiterten Plugin, daneben können weitere Abhängigkeiten spezifiziert werden. Die Installation von Plugins geschieht durch Installation im Dateisystem außerhalb der Kontrolle der Anwendung. Jedes Plugin wird mit einer statischen Beschreibung seiner Abhängigkeiten sowie seiner angebotenen und genutzten Schnittstellen versehen, so dass das Eclipse-System zum Start der Anwendung die Beziehungen zwischen allen installierten Plugins ermitteln kann.

Das OSGi-Rahmenwerk organisiert stattdessen das System in Bundles, die zueinander in einer oder mehreren Dienstbeziehungen stehen können. Die Dienstschnittstellenspezifikation, die dazu passenden Dienstanbieter und auch die Dienstnutzer können

unterschiedlichen Bundles zugehörig sein. Die Dienstbeziehung impliziert jeweils eine statische Abhängigkeit vom anbietenden bzw. nutzenden zum dienstspezifizierenden Bundle, daneben können wiederum zusätzliche Abhängigkeiten definiert werden. Die Installation und Deinstallation von Bundles sind als Teil des vom OSGi-Rahmenwerk gesteuerten Lebenszyklus definiert und werden daher als Laufzeitoperationen im Rahmenwerk angeboten. Alle Bundles werden im System gleichberechtigt behandelt, allerdings können Abhängigkeitsketten zwischen Diensten spezifiziert werden.

Sowohl das Eclipse-Plugin-System als auch das OSGi-Rahmenwerk definieren einen einheitlichen Lebenszyklus für alle Plugins bzw. Bundles, das weitere Verhalten an Erweiterungsschnittstellen kann von Plugin-Entwicklern in Form von Dokumentation spezifiziert werden. Um die Schnittstellen der klassischen Eclipse-Plugins nicht ändern zu müssen, werden beim Umbau des Rahmenwerkes zwar Plugins auf Bundles abgebildet, die Erweiterungsbeziehung bleibt jedoch neben der Dienstbeziehung bestehen. Somit können zwar aktualisierte Plugins bzw. Bundles beide Beziehungen nutzen und so auch bestehende Erweiterungs- und Dienstspezifikationen zueinander in Beziehung setzen, die Masse der Eclipse-Plugins bleibt aber den bereits definierten Erweiterungsbeziehungen treu.

Die **NetBeans**-Plattform (siehe Unterabschnitt 3.3.2) basiert auf einem um das Konzept des Moduls zentrierten Architekturstil. Nach der Ansicht leitender NetBeans-Entwickler unterscheidet sich ein Modul technisch nicht von einem Plugin. Unterschiede bestünden lediglich in der Verwendung, da Module auch Kernfunktionalität bereitstellen können. Ein NetBeans-Modul kann statische und funktionale Abhängigkeiten sowie angebotene und genutzte Dienste deklarieren. Zusätzlich kann ein Modul Dateien zu einem zentralen virtuellen Dateisystem beisteuern oder sich anhand solcher Dateien konfigurieren, die Kopplung von Modulen kann so anhand von identischen Pfadangaben im Dateisystem stattfinden. Die Modulentwickler sind frei in der Wahl der Beziehungsmechanismen zwischen den prinzipiell gleichberechtigt organisierten Modulen. Dennoch wird passend zum Architekturstil die Einhaltung bestimmter Richtlinien zur Dekomposition oder Kopplung von Modulen nach funktionalen[37] und technischen Gesichtspunkten empfohlen.

Die Installation von Modulen erfolgt wahlweise außerhalb der Kontrolle der Anwendung im Dateisystem oder mittels einer zentralen Modulverwaltung zur Laufzeit. Der globale Modul-Lebenszyklus ist zwar nicht explizit im Überblick spezifiziert, die zentrale Modulverwaltung fordert aber bei vielen Zustandsübergängen eine Genehmigung durch das betroffene Modul an. Eine automatische Kopplung der lokalen Dienst- oder Dateisystem-basierten Modulbeziehungen an den globalen Lebenszyklus besteht nicht. Zu weiteren von Modulen angebotenen Beziehungen oder Schnittstellen kann eine Verhaltensspezifikation in Form von Entwicklerdokumentation erfolgen.

Das Abstraktionskonzept im formalen Modell zum **MagicBeans**-Rahmenwerk (siehe Unterabschnitt 3.3.4) ist die Komponente, welche mehrfach in einer Paarbeziehung zu anderen Komponenten stehen kann. Die formal als Zuordnung von Anforderung und Angebot definierte Beziehung wird im Rahmenwerk über die Nutzung von extern übergebenen Instanzen eines öffentlichen Interfaces (Anforderung) und über die Be-

[37] Das Adjektiv „*funktional*“ wird in weiten Teilen dieser Arbeit nicht im mathematischen Sinne, sondern in der Bedeutung „*funktionalitätsorientiert*“ verwendet.

reitstellung einer Interface-implementiertenden Klasse (Angebot) implementiert. Damit besteht zur Laufzeit zwischen den Komponenten eine klassische objektorientierte Beziehung. Da das öffentliche Interface, das das identifizierende Element der Beziehungsdefinition darstellt, keiner der beteiligten Komponenten zugeordnet wird, existieren keine Anhaltspunkte zur Unterscheidung der Beziehung entlang einer Erweiterungs- oder Dienstmetapher.

Grundsätzlich ist die Befriedigung von Anforderungen optional, so dass es sich bei jeder Anforderung de facto um eine Plugin-Schnittstelle handelt. Beginn und Ende der Beziehung unterliegen allerdings der alleinigen Kontrolle des Rahmenwerkes, welches beim Hinzufügen einer Komponente automatisch einen Satz an möglichen Paarbeziehungen ermittelt und etabliert und diese beim Entfernen der Komponente wieder auflöst (und ggf. durch alternative Paarbeziehungen ersetzt). Das Hinzufügen, Ersetzen und Entfernen von Komponenten werden als Operationen des Rahmenwerkes zur Laufzeit angeboten, welche aber auch beim Start der Anwendung auf Basis einer Suche nach im Dateisystem installierten Komponenten angestoßen werden können.

Das Verhalten der zentralen Komponentenverwaltung ist formal spezifiziert. Ebenso können Komponenten mit algebraisch spezifizierten Verhaltensbeschreibungen für Anforderungen und Angebote versehen werden. Die durch die zentrale Verwaltung hergestellten Systemkonfigurationen können durch zentral vorgegebene formal spezifizierte Beschränkungen beeinflusst werden. Ein Komponentenentwickler hat im Gegensatz zum Anwendungskonstrukteur, der die Anwendung aus Komponenten kombiniert und den Satz möglicher Beziehungen durch formal spezifizierte Beschränkungen eingrenzen kann, keinen Einfluss auf die Organisation des resultierenden Systems.

Die Ergebnisse der vorangegangenen Diskussion werden in Tabelle 3.2 zusammengefasst. Der Tabellenaufbau weist beabsichtigte Ähnlichkeiten zum Aufbau der Tabelle 2.3 auf, um eine Einordnung von Plugin-Systemen in die in Kapitel 2 vorgestellten Softwareentwicklungsansätze zu ermöglichen. Die Einordnung wird in Unterabschnitt 3.4.3 stattfinden.

Den in der Tabelle aufgeführten Bezeichnern für die *Systemkomponenten* des jeweiligen Plugin-Systems sei – abgesehen von der offenbar vorhandenen Begriffsvielfalt – keine konzeptionelle Bedeutung zugemessen. Die angeführten Bezeichner stellen letztendlich nur Variationen des Konzepts der Kapselung von Systemteilen dar, welche sich im technischen oder historischen Hintergrund unterscheiden. Allen Systemen ist gemein, dass die Integration der Systemkomponenten zu einem Gesamtsystem so flexibel bewerkstelligt wird, dass die Verwendung des Plugin-Begriffs für die Systemkomponenten zunächst gerechtfertigt erscheint – ein Abgleich mit dem in Unterabschnitt 4.1.9 definierten Plugin-Begriff wird später in Unterabschnitt 4.1.9.1 und Abschnitt 4.4 erfolgen.

Interessanter sind die beobachteten Arten von *Beziehungen* zwischen den Systemkomponenten, deren Eigenschaften in Unterabschnitt 3.4.2 genauer aufgeschlüsselt werden. Die Unterschiede in den vorgefundenen Verfahren zur *Installation* sind wieder technischer Natur. Für die konzeptionelle Betrachtung ist allenfalls relevant, ob eine Installation zur Laufzeit möglich ist oder nicht, diese Information lässt sich allerdings einfacher aus der Einteilung *statisch / dynamisch* in der Tabellenzeile *Organisation* entnehmen. Mit einer Ausnahme werden Plugins in sämtlichen betrachteten Systemen

| System | Netscape-Plugins (NPAPI) | XPCOM | Mozilla-Extensions Chrome | XPInstall/Toolkit |
|---|---|---|---|---|
| *siehe Abschnitt* | 3.1.1 | 3.1.2.1 | 3.1.2.2 und 3.1.2.3 | 3.1.2.4 |
| Systemkomp. | Code-Module | Komponenten (Klassen) | Pakete | Pakete |
| Beziehung | Modul des Browsers | Objektbeziehungen | Pfadreferenzen | – (Abhängigkeiten) |
| Installation | durch Hersteller des Moduls (plattformabhängig) | durch Operation der Laufzeitumgebung (als Teil eines Pakets) | durch Operation der Laufzeitumgebung (als Teil eines Pakets) | durch Software-verwaltung |
| Dekomposition | funktional | funktional und datenorientiert | funktional | funktional |
| Organisation | zwei Ebenen statisch | flach dynamisch | flach/zwei Ebenen statisch | flach dynamisch |
| Verhaltens-spezifikation | Schnittstellen-vertrag und Lebenszyklus | Komponenten-verträge, Objektverwaltung | einheitlich (zentral akkumulierte Beschreibung) | einheitlich (zentrale Installa-tionsmechanismen) |

| System | Eclipse-Plugins klassisch | Eclipse-Plugins OSGi | NetBeans | MagicBeans |
|---|---|---|---|---|
| *siehe Abschnitt* | 3.2.1 | | 3.3.2 | 3.3.4 |
| Systemkomp. | Plugins | Bundles | Module | Komponenten |
| Beziehung | Erweiterungsbeziehung und Abhängigkeiten | Dienstbeziehung und Abhängigkeiten | Abhängigkeiten und Dienstbez./Pfadref. | extern gekoppelte Objektbeziehung |
| Installation | im Dateisystem | durch Operation der Laufzeitumgebung | durch ModuleManager oder im Dateisystem | durch Operation der Laufzeitumgebung |
| Dekomposition | funktional | funktional | funktional | funktional |
| Organisation | flach statisch | flach dynamisch | flach dynamisch | flach dynamisch |
| Verhaltens-spezifikation | Verträge der Erweiterungspunkte, globaler Lebenszyklus | Dienstschnitt-stellenverträge, globaler Lebenszyklus | Schnittstellenverträge und globaler Lebenszyklus | formale Spezifikation |

Tabelle 3.2: Gegenüberstellung der vorgestellten Plugin-Systeme

nach funktionaler Zusammengehörigkeit gebündelt. Lediglich XPCOM beherrscht hier auch eine Gliederung entsprechend den Konzepten abstrakter Datentypen, da letztlich Systemkomponenten im Sinne einer objektorientierten Klasse definiert werden. In den existierenden Plugin-Systemen treten nur zwei Arten von *Organisationshierarchie* auf. Die Systeme sind entweder flach organisiert oder verfügen über eine minimale Hierarchie mit exakt zwei Ebenen. Bei der flachen Organisation sind alle Plugins in der Systemarchitektur gleichberechtigt und können untereinander beliebige Beziehungen eingehen, lediglich das (anwendungsfunktionalitätsfreie) Plugin-Rahmenwerk hat eine Sonderrolle inne. Falls die Kern-Anwendungsfunktionalität ebenfalls auf der Ebene des Rahmenwerkes untergebracht ist, sollte diese Ebene ebenfalls als relevant betrachtet werden, so dass eine Zwei-Ebenen-Hierarchie entsteht.

*Verhaltensspezifikationen* treten in den meisten Plugin-Systemen auf zwei Ebenen auf: Zum einen wird in der Regel ein Lebenszyklus definiert, welcher die Integration von Plugins ins System steuert. Während der Vorstellung der Plugin-Systeme wurden nach Möglichkeit die Lebenszyklen in Form von Petrinetzen extrahiert, so dass sie in Unterabschnitt 4.4.2 verglichen werden können. Zum anderen wird zur Spezifikation von Verhalten zwischen Plugins auf Möglichkeiten der zugrundeliegenden Programmier- oder Beschreibungssprachen zurückgegriffen, was in den meisten Fällen nur auf Annotationen oder Konventionen zur Benutzung von Schnittstellen hinausläuft. Ausnahmen sind neben dem bereits hervorgehobenen *MagicBeans*, welches auf eine formale Spezifikationssprache setzt, die zwei Ebenen der Mozilla-Architektur, welche überhaupt keine Verhaltensspezifikation zwischen Systemkomponenten zulassen, da die Systemkomponenten kein über das vom Rahmenwerk vorgegebene hinausgehendes Verhalten aufweisen.

Die Tabelle 3.2 deckt – ebenso wie die bisherige Diskussion in diesem Unterabschnitt – nicht alle in den vorangegangenen Abschnitten diskutierten Plugin-Systeme ab, sondern lässt einige kleinere Projekte aus, deren Konzepte sich nicht stark genug von den diskutierten prominenten Architekturen absetzen. Unter anderem sind **JPF** (Unterabschnitt 3.3.1) und **CAP.NET** (Unterabschnitt 3.3.3) so eng an der Eclipse-Plattform (siehe Abschnitt 3.2) orientiert, dass sie keine neuen Konzepte beisteuern, sondern im Rahmen dieser Arbeit nur als Anlass zur Klärung und Identifikation einiger Konzepte des Eclipse-Systems dienen. Viele der im Unterabschnitt 3.3.5 erwähnten **Agentenplattformimplementierungen** (JADE, Capa und MadKit) beinhalten lediglich Variationen des Konzepts der in eine Anwendung integrierten anwendungsspezifischen Plugin-Schnittstelle, wie sie ausführlich am Beispiel der Netscape Plugin API (siehe Unterabschnitt 3.1.1) diskutiert wird. Der innovative Beitrag der beiden anderen erwähnten Plugin-Systeme aus dem Agentenbereich (DynamiCS und E2) liegt im Bereich der dynamischen Komposition von Agenten oder Verhaltensweisen aus einer Menge von Plugins.

Die Untersuchung konkreter Plugin-Architekturen wurde in den Unterabschnitten 3.3.6 bis 3.3.9 durch einen Blick auf verwandte Themengebiete abgerundet, in denen erweiterbare Softwarearchitekturen entwickelt, unterstützt oder erforscht werden.

Die **Integration** unabhängig voneinander entstandener Plugin-Systeme (siehe Unterabschnitt 3.3.6) hält – unabhängig davon, ob nur Plugins zu mehreren Rahmenwerke kompatibel implementiert oder gar mehrere Rahmenwerke miteinander verschmolzen

werden sollen – zwei zentrale Problembereiche bereit. Zum einen müssen die Abstraktionskonzepte (Systemkomponenten und Beziehungen) der beiden Softwarearchitekturen miteinander in Einklang gebracht werden. Zum anderen sind in den verschiedenen Softwaresystemen typischerweise unabhängig voneinander Klassenbibliotheken mit der Basisfunktionalität der jeweiligen Anwendungen entstanden, deren Angebot sich bei der Integration überschneiden oder auch gänzlich inkompatibel sein kann. Insbesondere der zweite Problembereich bereitet in der Praxis die größeren Probleme. Während die Abstraktionskonzepte in der Regel aufgrund ihrer generellen Ähnlichkeit aufeinander abgebildet oder nebeneinander angeboten werden können, erfordert die Berücksichtigung mehrerer Klassenbibliotheken in der Regel eine massive Code-Duplikation oder die Einführung weiterer Abstraktionsschichten.

Die in der Sprache **Java** (siehe Unterabschnitt 3.3.7) und ihrer zentralen Klassenbibliothek bereits vorhandenen oder in Entwicklung befindlichen Mechanismen zur Implementierung erweiterbarer Systeme unterstützen im Wesentlichen das flexible Laden und Instantiieren von Klassen. Dynamische Rekonfiguration wird nur sehr eingeschränkt geboten, das Entfernen einmal geladener Klassen ist nur über zusätzliche Rahmenwerke wie z. B. EJB oder OSGi praktikabel.

Da Plugins für ein System eines Herstellers durch Dritte entwickelt werden können sollen, treffen viele Probleme der **verteilten Entwicklung** von Software auch auf die Entwicklung von Plugin-Software zu (siehe Unterabschnitt 3.3.8). Ausgenommen ist hierbei der Problembereich der geographischen Verteilung eines Entwicklerteams, da Plugin-Entwickler sich in der Regel nicht mit den Entwicklern des Rahmenwerkes koordinieren müssen. Das dynamische Verknüpfen von unabhängig kompilierten Code-Modulen kann zu Abweichungen in der verfügbaren Funktionalität zwischen Kompilations- und Ausführungsumgebung eines Moduls führen. Während sich syntaktische Änderungen durch das Konzept der Binärkompatibilität eingrenzen lassen, sind semantische Änderungen an Schnittstellenverträgen nicht tolerabel. Die eindeutige Festlegung von Schnittstellenverträgen anhand von Versionsnummern und Namensräumen kann bei unabhängiger Entwicklung nicht sichergestellt werden, da Namenskonflikte prinzipiell möglich bleiben. Eine formale Vorab-Verifikation des Gesamtsystems ist in erweiterbaren Systemen prinzipbedingt ebenfalls unmöglich.

Zwischen **Betriebssystemen** (siehe Unterabschnitt 3.3.9) und generischen Plugin-Systemen lassen sich etliche Parallelen aufzeigen. Betriebssysteme bieten eine grundlegende Verwaltung für Programmcode ähnlich einer Plugin-Verwaltung an, indem sie Dateisystem und ABI (*Application Binary Interface*) definieren. Modulare Betriebssystemarchitekturen erlauben die Installation von Gerätetreibern, Bildschirmschonern etc. als Plugins. Die für viele Betriebssysteme verfügbaren Paketverwaltungen erlauben ebenso wie Plugin-Verwaltungen eine nach Funktionalität gebündelte Installation und Deinstallation von Softwarepaketen, wobei auch statische Abhängigkeiten berücksichtigt werden können.

### 3.4.2 Beziehungsarten

In Tabelle 3.2 werden den untersuchten Plugin-Systemen verschiedene Beziehungsarten zugeordnet. Eine Beziehungsart legt die Rollen und Interaktionsmöglichkeiten von Plugins fest, wenn diese mit anderen Teilen des Systems zu einem Gesamtsystem

zusammengefügt werden. Es handelt sich somit um ein architekturprägendes Konzept, welches ähnlich wie die den Softwareentwicklungsparadigmen zugrundeliegenden Konzepte als Leitlinie bei der (De-)Komposition eines Softwaresystems dient (vgl. Kapitel 2). Sobald ein Softwareentwickler sich auf eine Beziehungsart festgelegt hat, dient diese als zentrale Metapher für die Interaktion zwischen Systemkomponenten und bestimmt die Art und Weise, wie Funktionalität zwischen Systemkomponenten angeboten, genutzt oder anderweitig zusammengefügt werden kann. Mitunter bestimmt eine Beziehungsart auch die technischen Mittel und Wege, wie in einem Plugin-System die Integration von Plugins stattfindet. Dieser technische Aspekt ist jedoch für die angestrebte Modellierung des Plugin-Konzepts von nachrangiger Bedeutung.

Die in der Tabellenzeile zur Bezeichnung der Beziehungsarten verwendeten Begriffe werden im Folgenden informell erläutert.

**Modulbeziehung** Gemeint ist hier nicht die Idee der Modularisierung [vgl. Par72], welche später zum Grundstein aller modernen Softwareentwicklungsparadigmen wurde, sondern die technische Seite der Modulbeziehung: Code aus unabhängig voneinander übersetzten Modulen kann zur Laufzeit in einer Anwendung kombiniert werden, da die Laufzeitumgebung (z. B. das Betriebssystem oder eine virtuelle Maschine) einen Lademechanismus bereitstellt, welcher als extern deklarierte Modul- und Prozedurbezeichner rechtzeitig zur Verwendung auflöst.

Die Netscape-Plugin-API (NPAPI) demonstriert zwar exemplarisch, wie diese eigentlich statische Beziehung zwischen Code-Modulen flexibilisiert werden kann, indem Modulnamen zur Laufzeit ermittelt statt fest im Code eingetragen werden (siehe Unterabschnitt 3.1.1.2). Allerdings findet genau durch diesen Schritt der Übergang zur Erweiterungsbeziehung statt.

Die Modulbeziehung wird in Unterabschnitt 4.4.1.1 detaillierter betrachtet werden.

**Objektbeziehung** Unter einer Objektbeziehung sei hier eine beliebige Beziehung zwischen Objekten verstanden. Dies bedeutet, dass die Systemkomponenten als gekapselte Objekte auf Basis instantiierter Klassen mit deklarierten Schnittstellen zu verstehen sind, zwischen denen zur Laufzeit Beziehungen auf- und abgebaut werden können, indem Objektreferenzen übergeben werden. Die Softwarearchitekturen, welche in Tabelle 3.2 durch eine Objektbeziehung charakterisiert wurden (Mozilla-XPCOM und MagicBeans), zeichnen sich dadurch aus, dass sie lediglich die Instantiier-, Identifizier- und Referenzierbarkeit ihrer Systemkomponenten definieren und die Deklaration von Schnittstellen unterstützen, aber darüber hinaus keine beziehungsspezifischen Rollen zur einheitlichen Strukturierung des Systems festlegen.

Die Objektbeziehung wird in Unterabschnitt 4.4.1.2 detaillierter betrachtet werden.

**Erweiterungsbeziehung** Die Erweiterungsbeziehung gibt eine klare Rollenteilung zwischen erweiterbarem System und Erweiterung vor: Das erweiterbare System stellt den Kontext und spezifiziert eine Erweiterungsschnittstelle, während die Erweiterung darin eingepasst zusätzliche Funktionalität bereitstellen kann. Damit wird

eine asymmetrische Beziehung aufgebaut, da die Erweiterung ohne den Kontext des erweiterbaren Systems keine sinnvoll nutzbare Funktionalität bietet. Von den betrachteten existierenden Plugin-Systemen thematisiert vor allem Eclipse die Erweiterungsbeziehung explizit (siehe Unterabschnitt 3.2.1.5). Allerdings bauen auch sämtliche anwendungsspezifischen Plugin-Systeme (wie z. B. die NPAPI) auf der nicht-rekursiven Erweiterungsbeziehung auf.

Die Erweiterungsbeziehung wird in Unterabschnitt 4.1.3 als „*intendierte Erweiterbarkeit*“ ausführlich besprochen werden, da sie als Grundlage für das referenznetzbasierte Modell des Plugin-Konzepts (siehe Abschnitt 4.3) dient.

**Dienstbeziehung** Die Dienstbeziehung umfasst drei Beteiligte: Dienstanbieter, Dienstnutzer und Dienstspezifikation. Ein Dienstnutzer kann einen Dienstanbieter nutzen, wenn dieser dieselbe Dienstspezifikation unterstützt wie vom Nutzer gewünscht. Die Dienstbeziehung ist sehr flexibel, da – eine passende Dienstspezifikation vorausgesetzt – jeder Nutzer mit jedem Anbieter eine Beziehung eingehen kann. Sie wird vor allem im OSGi-Rahmenwerk (siehe Unterabschnitt 3.2.1.6) und in NetBeans als Muster für Beziehungen zwischen den Systemkomponenten eingesetzt.

Die Dienstbeziehung wird in Unterabschnitt 4.1.3 im Vergleich zur Erweiterungsbeziehung andiskutiert und im Unterabschnitt 4.3.8.1 ins Modell des Plugin-Konzepts integriert werden.

**Abhängigkeit** In der Betrachtung existierender Plugin-Systeme sind zwei Varianten des Abhängigkeitsbegriffs aufgetreten, es kann zwischen statischer und funktionaler Abhängigkeit unterschieden werden.

Eine **statische Abhängigkeit** bedeutet, dass der Code einer Systemkomponente nicht ohne die Verfügbarkeit des Codes einer explizit benannten anderen Systemkomponente ausgeführt werden kann. Es handelt sich dabei um eine in der abhängigen Systemkomponente formulierte Bedingung an die Laufzeitumgebung. Statische Abhängigkeiten werden in vielen Plugin-Systemen (z. B. in Eclipse und dem OSGi-Rahmenwerk – siehe Unterabschnitt 3.2.1.7 –, in NetBeans und in der Mozilla-XPInstall-Komponente) zentral verwaltet, um die Einhaltung der in den Plugins deklarierten Bedingungen sicherzustellen und damit Fehlermeldungen seitens des Modulladememechanismus des zugrunde liegenden Betriebssystems zu vermeiden.

Eine **funktionale Abhängigkeit** bedeutet, dass die Funktionalität einer Systemkomponente nicht erbracht werden kann, ohne dass eine beliebige andere Systemkomponente eine explizit benannte andere Funktionalität bereitstellt. Damit ist zwar – im Gegensatz zur statischen Abhängigkeit – der Code einer Systemkomponente ohne die Verfügbarkeit einer explizit benannten anderen Systemkomponente vollständig interpretierbar, kann aber ohne die benötigte Fremdfunktionalität nicht genutzt werden. Funktionale Abhängigkeiten werden von den in diesem Kapitel betrachteten Systemen von der NetBeans-Verwaltung sowie der OSGi-Service-Component-Runtime (SCR) explizit unterstützt. Beide Systeme modellieren im Wesentlichen Dienstbeziehungen und bieten mit der

Deklaration funktionaler Abhängigkeiten eine Aufwandsreduktion bei der Konfiguration von Dienstnutzungen.

Beide Varianten der Abhängigkeitsbeziehung stellen eine asymmetrische Beziehung dar – die abhängige Systemkomponente benötigt zwingend den Code oder die Dienstleistung der anderen Systemkomponente, jene kann aber auch ohne die Einhaltung dieser Bedingung ihre Funktionalität erbringen.

Die statische Abhängigkeit wird in Unterabschnitt 4.4.1.1 noch einmal in Zusammenhang mit der Modulbeziehung diskutiert. Eine mögliche Modellierung funktionaler Abhängigkeiten wird in Unterabschnitt 4.3.8.1 im Kontext der Modellierung von Dienstbeziehungen angesprochen.

**Kopplung mittels Pfadreferenzen** Sämtliche in diesem Kapitel betrachteten Plugin-Systeme nutzen notwendigerweise Namensräume zur Identifikation der Anknüpfungspunkte für Beziehungen. Die Namensräume sind oftmals hierarchisch strukturiert (analog einem Dateisystem, der Java-Paketstruktur oder dem URI-Schema), so dass Pfade anstelle einfacher Namen verwendet werden. Zwei der betrachteten Plugin-Systeme, namentlich Mozilla-Chrome und NetBeans, verzichten jedoch auf die Einbettung der Pfadnamen in ein festes Schema wie z. B. Erweiterungs- oder Dienstbeziehungen, so dass die Pfade unmittelbar zur allgemeinen, symmetrischen Referenzierung anzukoppelnder Systemkomponenten eingesetzt werden.

Einen Überblick über wesentliche Eigenschaften der genannten Beziehungsarten gibt Tabelle 3.3. In ihr ist dargestellt, wie eine Beziehung zwischen Systemkomponenten aufgebaut werden kann und welche Rollen die Systemkomponenten in der Beziehung innehaben. Jeder Rolle ist auf knappe Weise durch die Kardinalität, in der sie im Verhältnis zu den anderen an der Beziehung beteiligten Rollen auftreten kann, und durch ihre elementaren Aufgaben in der Beziehung charakterisiert. Es werden zwei Aufgaben unterschieden: Zum einen muss die der Beziehung zugrundeliegende Schnittstellenspezifikation definiert werden. Zum anderen kann nur eine der beteiligten Rollen als Initiator der Beziehung auftreten und die Lebensdauer festlegen, indem sie eine Referenz auf andere Beteiligte nutzt. In der letzten Spalte ist daher festgehalten, welche Rollen in einer Beziehung referenzierbar und damit auch kommunizierbar sind.

Jedes Code-bereitstellende *Modul* kann von beliebig vielen anderen Modulen genutzt werden, indem seine Funktionen aufgerufen werden. Jedes aufrufende Modul kann zwar mehrere bereitstellende Module nutzen, allerdings ist statisch zu jeder aufzurufenden Funktion genau ein bereitstellendes Modul festgelegt. Die Menge der zur Verfügung stehenden Funktionen wird vom bereitstellenden Modul festgeschrieben. Da ein Modul separat kompilierbar sein soll, muss es auch alle angebotenen Schnittstellen selber deklarieren.

Referenzen zwischen *Objekten* können einen beliebigen Graphen bilden und sind jederzeit austauschbar, so dass hier eine $n : m$-Beziehung besteht. Das referenzierte Objekt hat aber keine Möglichkeit, „rückwärts“ entlang der Beziehung zu einer Referenz des referenzierenden Objekts zu gelangen. Daher obliegt die Steuerung der Lebensdauer der Beziehung alleine dem referenzierenden Objekt. Die Aufgabenverteilung unter

| Beziehungsart | Aufbau | Beteiligte Rollen und ihre Aufgaben | | | | |
|---|---|---|---|---|---|---|
| | | Rollenname | Zahlenverhältnis | definiert Schnittstelle | steuert Beziehung | Identität wird kommuniziert |
| Modulbeziehung | fest verdrahtet | Aufrufer | $m$ | | x | |
| | | Bereitsteller | $:1$ | x | | |
| Objektbeziehung | Referenzübergabe | referenzierendes | $m$ | | x | |
| | | referenziertes Objekt | $:n$ | x | | x |
| Erweiterungsbeziehung | Installation | erweiterbares System | 1 | x | x | |
| | | Erweiterung | $:n$ | | | x |
| Dienstbeziehung | Verzeichnisanfrage | Nutzer | $m$ | | x | |
| | | Spezifikation | $:1$ | x | | x |
| | | Anbieter | $:n$ | | | x |
| statische Abhängigkeit | fest verdrahtet | Abhängiger | $m$ | | | |
| | | Bereitsteller | $:1$ | x | | |
| funktionale Abhängigkeit | automat. Verzeichnisanfrage | Nutzer | $m$ | | | |
| | | Spezifikation | $:1$ | x | | x |
| | | Anbieter | $:n$ | | | x |
| Pfadreferenzen | beliebig | beliebig | $m$ | x? | x? | |
| | | Pfadname | $:1$ | | | x |

Tabelle 3.3: Eigenschaften der Beziehungsarten

den Rollen weist eine starke Ähnlichkeit zur Modulbeziehung auf. Dies ist nicht verwunderlich, da die Konzepte der Objektorientierung aus dem Modularisierungkonzept hervorgegangen sind.

In der *Erweiterungsbeziehung* ist das erweiterbare System sowohl für die Schnittstellenspezifikation als auch für die Steuerung der Beziehungsdauer zuständig. Da Erweiterungen fest auf ein konkretes erweiterbares System zugeschnitten sind, ist eine Weitergabe von Referenzen mit Ausnahme des Installationsvorgangs unnötig.

*Dienst*anbieter und -nutzer können in beliebiger Vielzahl zueinander finden, solange sie sich auf dieselbe Dienstspezifikation beziehen. In der Dienstbeziehung sollte die Spezifikation daher von einer dritten Partei definiert werden (was aber nicht immer so sein muss). Die Beziehung wird vom Dienstnutzer nach Bedarf aufgebaut, indem er sich aus einem Dienstverzeichnis Referenzen auf passende Anbieter geben lässt.

Eine *statische Abhängigkeit* legt die abhängige Systemkomponente auf genau eine andere Systemkomponente fest, ohne welches sie nicht ausführbar ist. Diese Festlegung entspricht dem Benennen eines bereitstellenden Moduls durch das aufrufende Modul in der Modulbeziehung. Die Ausführungsdauer der abhängigen Systemkomponente bestimmt eigentlich – wie in der Modulbeziehung – die Lebensdauer der Abhängig-

keitsbeziehung, allerdings kann die abhängige Systemkomponente diese nicht explizit steuern. Bei nicht-verfügbarer bereitstellender Systemkomponente kann die abhängige Systemkomponente gar nicht erst ausgeführt werden. Eine *funktionale Abhängigkeit* wird wie die Dienstbeziehung über eine externe Spezifikation definiert, an der sich beliebig viele Nutzer und Anbieter orientieren können. Der Unterschied zur Dienstbeziehung besteht darin, dass die Lebensdauer der Beziehung (wie bei der statischen Abhängigkeit) an die Ausführungsdauer der Nutzer und die Verfügbarkeit eines Anbieters gekoppelt ist, so dass Beziehungsauf- und -abbau vom Rahmenwerk entlang der deklarativ spezifizierten Abhängigkeit automatisiert durchgeführt werden können.

Die Kopplung mittels *Pfadreferenzen* legt – abgesehen von der passiven Rolle des gemeinsam bekannten Pfadnamens – überhaupt keine beteiligten Rollen fest, da jede Systemkomponente einen Pfadnamen sowohl lesend als auch schreibend (in der Dateisystemanalogie) bzw. nur passiv (in der zentralisierten Datenstrukturbildung) nutzen kann. Eine Systemkomponente kann z. B. Veränderungen im Dateisystem zwecks Beziehungsauf- und -abbau beobachten, muss es aber nicht.

Technisch gesehen sind alle beobachteten Beziehungsarten in der Lage, beliebige andere Beziehungsarten aus dieser Liste zu imitieren, wenn die Aufgaben und Möglichkeiten der beteiligten Rollen großzügig ausgelegt oder mehrere Instanzen der imitierenden Beziehung genutzt werden. Wie bereits am Beispiel der NPAPI erwähnt, kann die Modulbeziehung durch Aufsatz einer Verwaltung zur Auswahl passender Module zur Erweiterungsbeziehung ausgebaut werden, gleiches gilt für den Ausbau zur Dienstbeziehung. Die Erweiterungsbeziehung kann als spezielle Kombination zweier Dienstbeziehungen aufgefasst werden, wobei aber auch umgekehrt mehrere Erweiterungsbeziehungen zur Modellierung einer Dienstbeziehung herangezogen werden können (vgl. Unterabschnitt 4.1.3.1). Pfadreferenzen und Objektbeziehungen lassen ohnehin die Unterordnung unter andere Beziehungsarten zu, umgekehrt können sie (ebenso wie die Modulbeziehung) als degenerierte Varianten der anderen Beziehungsarten (z. B. unter Verzicht auf zu erfüllende Pflichten in Schnittstellenverträgen) aufgefasst werden. Statische und funktionale Abhängigkeiten sind wiederum der Modul- bzw. der Dienstbeziehung so eng verwandt, dass eine Abbildung leicht fällt (vgl. Abschnitt 4.4). Die in diesem Unterabschnitt eingeführte Unterscheidung dient daher mehr der Betonung bevorzugter Rollenverteilungen innerhalb einer Softwarearchitektur. Die Diskussion der Unterschiede und Ähnlichkeiten einiger Beziehungsarten wird im folgenden Kapitel gelegentlich kurz aufgegriffen werden.

### 3.4.3 Plugin-Systeme und Softwareentwicklungstechniken

Im vorigen Kapitel wurden mehrere Paradigmen und Techniken zur Softwareentwicklung einander gegenübergestellt. Die in diesem Kapitel untersuchten Plugin-Systeme können nun diesen Ansätzen ebenfalls gegenübergestellt werden. Zum einen lassen sich die in Abschnitt 2.6 tabellarisch zusammengestellten Kriterien auf die untersuchten Plugin-Systeme übertragen, so dass die prägenden Konzepte und Eigenschaften der so geprägten Softwarearchitekturen vergleichbar werden. Zum anderen stellt sich die Frage, ob sich die untersuchten Plugin-Systeme einem oder mehreren der Softwareentwicklungsparadigmen unterordnen oder vollständig durch eines oder mehrere der Softwareentwicklungstechniken beschreiben lassen.

Um die in diesem Kapitel vorgestellten Plugin-Systeme mit den in Kapitel 2 diskutierten architekturprägenden Konzepten in Beziehung setzen zu können, werden in Tabelle 3.2 ähnliche Maßstäbe wie in der Gegenüberstellung der Ansätze zur Softwareentwicklung angelegt. Im Vergleich zur Softwarearchitekturdiskussion (siehe Tabelle 2.3) haben sich dennoch einige Bezeichnungen der Tabellenzeilen verändert. Da es sich bei den in diesem Kapitel betrachteten Plugin-Systemen um konkrete Systeme anstelle von abstrakten Architekturkonzepten handelt, werden einige Kriterien weiter ausdifferenziert.

An die Stelle von architekturprägenden *Abstraktionskonzepten* treten hier konkrete Bezeichner für die *Systemkomponenten* eines Softwaresystems und ihre *Beziehungen*. Beide Bezeichner dienen wie die Abstraktionskonzepte als Metaphern, welche die Softwarearchitektur prägen und die Art der Kopplung von Systemkomponenten vorgeben. Die im Vergleich zur Softwarearchitekturdiskussion feinere Unterscheidung in zwei Metaphern – eine für Systemkomponenten und eine für Beziehungen – ist nötig, da die in der Untersuchung angetroffenen Bezeichner für Systemkomponenten eher willkürlich gewählt scheinen und nicht so starke Unterschiede benennen wie es die Begriffe suggerieren. In den vorgestellten Softwareentwicklungsparadigmen ist wiederum die bevorzugt verwendete Beziehungsart (meist die Dienstbeziehung) weniger prägend als das Abstraktionskonzept, da sie nicht fest vorgegeben wird.

Die *Integration*sweise von Systemkomponenten, welche im Vergleich der Softwareentwicklungsansätze ein Unterscheidungsmerkmal darstellt, findet bei Plugin-Systemen einheitlich in Form einer „Installation durch Anwender“ statt. Kennzeichnend ist hierbei, dass fertige Programmteile integriert werden und keine weitere Programmierung oder Konfiguration seitens des Anwenders erforderlich ist. Damit unterscheiden sich Plugins deutlich von den in Tabelle 2.3 angeführten Ansätzen, in welchen Ingenieure, Modellierer oder Entwickler die Integration zeitlich getrennt vom Einsatz des Systems beim Kunden durchführen. Anstelle eines immer gleichen Eintrags unter dem Merkmal *Integration* findet in der Zeile *Installation* in Tabelle 3.2 eine deutlich detailliertere Gegenüberstellung der dem Anwender zur Verfügung gestellten Installationswerkzeuge oder -schnittstellen statt.

Die anderen in den beiden Tabellen aufgeführten Merkmale lassen sich unmittelbar vergleichen. Alle in Kapitel 2 diskutierten Softwareentwicklungsparadigmen und -techniken haben mit den Plugin-Architekturen eine starke Orientierung zur funktionalen Dekomposition von Systemen gemein, lediglich die OOSE und die AOSE ergänzen zusätzlich eine datenorientierte bzw. prozessorientierte Dekompositionsrichtung.

Die Systemkomponenten sind im Rahmen der meisten Softwareentwicklungsparadigmen flach organisiert, da Beziehungen zwischen Systemkomponenten in alle Richtungen hergestellt werden können. Lediglich die hierarchische CBSE und die Architekturbeschreibungssprachen unterstützen eine Beschränkung der Beziehungen auf eine hierarchische Organisation. Bei den untersuchten Plugin-Systemen dominiert ebenfalls die flache Organisation auf der Ebene einer zentralen Plugin-Verwaltung. Eine feste Organisationsform findet sich nur bei den Systemen, die mit einer einzigen, anwendungsspezifisch definierten Plugin-Schnittstelle ausgestattet sind (z. B. die NPAPI und die Chrome-Oberfläche). Die Dominanz flacher Organisation ist insofern verwunderlich, als dass die Grundidee eines Plugins – das Hinzufügen spezieller Funktionalität zu einem kontextgebenden Kernsystem – eine hierarchische Organisation nahelegt.

Dieser scheinbare Widerspruch wird im Rahmen der konzeptionellen Modellierung im folgenden Kapitel aufgelöst werden.

Bei den Plugin-Architekturen geht der Trend zu einer dynamisch zur Laufzeit veränderlichen Systemkonfiguration, es existieren allerdings auch noch etliche Systeme mit einer statischen Konfiguration, die nur zu Beginn der Laufzeit ermittelt wird. Die klassische CBSE sowie die Entwurfsmuster verfolgen eher das Prinzip der statischen Konfiguration, während ADLs teilweise Rekonfiguration zur Laufzeit unterstützen und die AOSE ein vollständig dynamisches, sich selbst organisierendes System betrachtet.

Mit Ausnahme der AOSE, in der Interaktionen als gleichberechtigtes, verhaltensorientiertes Abstraktionskonzept neben der statischen Systemstruktur stehen, unterstützen alle Ansätze und Systeme nur die Annotation von Verhalten an Schnittstellen. Je nach System erfolgt die Definition des gewünschten Verhaltens durch Entwicklerdokumentation in Form von Schnittstellenverträgen oder durch (semi-)formale Annotation an der eigentlichen Schnittstellenbeschreibung bzw. -implementierung.

Im direkten Vergleich zu rein objektorientiert entwickelten Systemen unterscheiden sich Plugin-Systeme vor allem durch ihre Granularität und den Fokus auf wechselnde Verfügbarkeit von Funktionalität. Rein objektorientierte Systeme werden zunächst unter der Annahme einer geschlossenen Welt aus Klassen entworfen, lediglich die Beziehungen zwischen Instanzen (Objekten) werden dynamisch und flexibel aufgebaut. Die Beziehungen zwischen Plugins sind (wie die von Klassen) statisch vorgegeben, um eine einfache Integration zu ermöglichen.

Mit komponentenbasierten Systemen teilen Plugin-Systeme die Idee der wohldefinierten Schnittstellen und der flexiblen Komposition von Systemkomponenten (siehe auch Diskussion in Unterabschnitt 2.2.5). Ausgenommen sind hier lediglich die Systeme mit einer einzelnen, anwendungsspezifischen Plugin-Schnittstelle, welche – bis auf die Plugins – monolithisch konstruiert sein können. Reine Plugin-Systeme können aber als komponentenbasierte Systeme angesehen werden. Allerdings sind Plugins nicht so generisch wiederverwendbar wie Komponenten, da sie in der Regel auf *ein* konkretes Plugin-System zugeschnitten sind (siehe Diskussion zur Problematik von Plugins zu mehreren Plugin-Systemen in Unterabschnitt 3.3.6). Dieses Argument wird in Unterabschnitt 4.1.9.1 weiter ausgeführt werden.

Die Techniken der Beschreibung von Softwarearchitekturen und der Definition von Entwurfsmustern werden auf einer anderen Abstraktionsebene als der eines konkreten Plugin-Systems definiert. Beide Techniken lassen sich zur Beschreibung von Plugin-Systemen einsetzen. Die grundlegenden Prinzipien reiner Plugin-Systeme wie z. B. Eclipse und NetBeans könnten zur Definition eines Architekturstils im Rahmen der Beschreibung von Softwarearchitekturen herangezogen werden – dieser Gedanke wird in Unterabschnitt 4.5.4 im Anschluss an die konzeptionelle Modellierung kurz diskutiert werden. In einem anwendungsspezifischen Plugin-System ist dagegen die Integration von Plugins weniger architekturprägend. Stattdessen kann die Plugin-Schnittstelle nebst zugehöriger Verwaltung – wie in Unterabschnitt 2.4.3 vorgestellt – als Entwurfsmuster betrachtet werden.

In der agentenorientierten Softwareentwicklung werden Systeme betrachtet, die deutlich dynamischer und flexibler als Plugin-Systeme sind. Diese Flexibilität geht so weit, dass wie in objektorientierten Systemen kaum Merkmale vorhanden sind, welche die Beziehungen zwischen Systemkomponenten in eine bestimmte Richtung prägen. Aller-

dings ist die AOSE in der Lage, die Möglichkeit der dynamischen Architekturänderung über die grundlegenden Eigenschaften des Agenten zu erklären, statt sie – wie in den anderen Paradigmen und Techniken nötig – als Hilfstechnik um das zentrale Abstraktionskonzept herum zu implementieren.

Keiner der vorgestellten Softwareentwicklungsansätze bietet also alleine ein architekturprägendes Konzept, welches die Eigenschaften der untersuchten Plugin-Systeme vollständig beschreibt. In jedem Paradigma und jeder Technik sind zusätzliche Vorgaben zu ergänzen oder einige Vorgaben zu lockern.

Die in der Untersuchung vorgefundene Vielfalt an Begriffen für Plugins und deren Beziehungen muss nun in ein einheitliches Plugin-Konzept zusammengeführt werden. Während der Festlegung und Definition zentraler Konzepte wird im folgenden Kapitel häufig auf die in diesem Kapitel angeführten Beispiele verwiesen werden.

# 4 Konzeptionelle Modellierung

Nach der Aufarbeitung verwandter Paradigmen und Techniken aus dem Bereich der Softwareentwicklung in Kapitel 2 und der Erarbeitung der Grundzüge der Softwarearchitekturen existierender Plugin-Systeme in Kapitel 3 kann nun die konzeptionelle Modellierung von Plugin-Systemen angegangen werden.

Die Modellierung ist ein mehrschrittiger Prozess. Zuerst werden in Abschnitt 4.1 aus der erfolgten Betrachtung existierender Plugin-Systeme Anforderungen an ein Plugin-System abgeleitet und zentrale Begriffe definiert. Anschließend wird in Abschnitt 4.2 die Wahl der verwendeten Modellierungstechnik Referenznetze begründet und die sich daraus ergebenden Möglichkeiten der Modellierung erläutert. Gegenstand des Unterabschnitts 4.2.6 sind einige bei der Modellierung vorgenommene Abstraktionen von in Kapitel 3 besprochenen Details implementierter Systeme.

Ein konzeptionelles Modell von Plugin-Systemen wird in Abschnitt 4.3 durch schrittweise Verfeinerung einer abstrakten Beschreibung entworfen, um die zentralen Modellierungsideen auf nachvollziehbare Weise zu entwickeln. Abschließend wird in Abschnitt 4.4 die Anwendbarkeit des Modells auf die unterschiedlichen in den existierenden Plugin-Systemen vorgefundenen Softwarearchitekturen, konkret deren Beziehungsarten und Lebenszyklen, überprüft.

## 4.1 Anforderungen und Definitionen

Der erste Schritt zur Modellierung eines Plugin-Konzeptes ist die Charakterisierung des zu modellierenden Konzeptes, indem elementare Anforderungen erarbeitet und zentrale Begriffe geschärft werden. Als Basis für die Charakterisierung eignen sich die erkannten Eigenschaften der in Kapitel 3 vorgestellten, höchst unterschiedlichen Plugin-Systeme. Aus Eigenschaften, die allen oder zumindest vielen untersuchten Systemen gemein sind, lassen sich essentielle Anforderungen an ein Plugin-System ableiten. Zugleich lässt sich die in der Untersuchung vorgefundene Begriffsvielfalt mittels der konkreten Definition grundlegender Begriffe bändigen, so dass in der weiteren Modellierung des Plugin-Konzepts präzise Ausdrucksmöglichkeiten bestehen. Insbesondere kann im Anschluss an diese Begriffsklärung der zentrale Begriff des Plugins definiert werden (siehe Unterabschnitt 4.1.9).

Zu Beginn der Anforderungen stehen die vom Anwender wahrnehmbaren Eigenschaften eines Plugin-Systems: Der Hauptnutzen liegt in der *dynamischen Änderbarkeit* der inneren Zusammensetzung eines Softwaresystems, um die vom Gesamtsystem gebotene Funktionalität während seines Einsatzes an individuelle Bedürfnisse anzupassen. Daher müssen zu einem (weitgehend) beliebigen Zeitpunkt Systemkomponenten hinzugefügt oder entfernt werden können (siehe Unterabschnitt 4.1.1). Alle untersuchten Plugin-Systeme erlauben es Anwendern, die Zusammensetzung des von ihnen

persönlich genutzten Softwaresystems auf *einfache Weise* zu konfigurieren (siehe Unterabschnitt 4.1.2). Diese Eigenschaften werden in den existierenden Plugin-Systemen durch mehrere Techniken aus dem Bereich der Softwareentwicklung erreicht. Jedes der Softwaresysteme definiert bereits im Vorwege *Möglichkeiten zur Erweiterung* durch andere Systemkomponenten (siehe Unterabschnitt 4.1.3). Die Softwarearchitekturen grenzen klar *gekapselte* Module ab, welche hinzugefügt und entfernt werden können (siehe Unterabschnitt 4.1.4). Das Hinzufügen und Entfernen von Systemkomponenten geschieht unter der Maßgabe eines ebenfalls im Vorwege definierten *Lebenszyklus* (siehe Unterabschnitt 4.1.5). Vorgesehene Erweiterbarkeit, Kapselung und Lebenszyklen werden in den untersuchten Systemen in der Regel auf zwei (oder mehr) *Ebenen* eingesetzt: Während eine zentrale Verwaltung das Hinzufügen und Entfernen von Systemkomponenten auf der Ebene der Softwarearchitektur vereinheitlicht, muss zu jeder vorgesehenen Erweiterungsmöglichkeit die Integration und Entkopplung fachlich definiert (oder implementiert) werden (siehe Unterabschnitt 4.1.6). Diese Liste der Anforderungen an Plugin-Systeme wird durch zwei Punkte ergänzt, welche allgemein der Modellierung moderner Softwaresysteme zuträglich sind und daher bei der konzeptionellen Modellierung von Plugin-Systemen zu berücksichtigen sind: die Berücksichtigung von *Nebenläufigkeit* im System (siehe Unterabschnitt 4.1.7) sowie die Verfügbarkeit einer *Modellierungssprache* mit präziser und dennoch anschaulicher Semantik (siehe Unterabschnitt 4.1.8).

### 4.1.1 Dynamische Änderbarkeit

Plugin-Systeme sollen es Anwendern ermöglichen, eine Anwendung an individuelle und wechselnde Bedürfnisse anzupassen. Insbesondere soll die Anwendung nicht nur dauerhaft, sondern auch vorübergehend um zusätzliche Funktionalität ergänzt werden können. Um die Anwendung schlank zu halten, wird nicht die gesamte mögliche Funktionalität fest integriert und durch kleine Änderungen an Konfigurationsparametern freigeschaltet, sondern der zugehörige Code tatsächlich erst bei Bedarf in das Softwaresystem integriert. Plugin-Systeme sind daher *offene, rekonfigurierbare Systeme.*

Viele Plugin-Systeme lassen solche Änderungen der Funktionalität nur zwischen Programmende und -neustart zu, so dass die Zusammenfügung der Systemteile während der Laufzeit des Programms statisch ist. Falls zusätzliche Funktionalität nur für einen sehr kurzen Zeitraum benötigt wird oder falls das Gesamtsystem eine sehr lange, unterbrechungsfreie Laufzeit bieten soll, muss eine Rekonfiguration auch dynamisch während der Laufzeit möglich sein. Einige existierende Plugin-Systeme haben diesen Bedarf erkannt und bemühen sich, dynamische Rekonfigurierbarkeit zu implementieren. Solche Plugin-Systeme sind daher *dynamisch rekonfigurierbare Systeme.*

Die Charakterisierung als offene und dynamisch rekonfigurierbare Systeme soll in den folgenden Definitionen präzisiert werden, um daraus Anforderungen an das zu entwickelnde Plugin-Konzept ableiten zu können.

**Definition 1 (System)** *Ein System ist eine organisierte Zusammenfügung von (System-)Komponenten mit dem Zweck, eine spezifische Funktionalität zu erbringen.*[1]

Ein System ist in einer Umgebung situiert, welche das System beeinflussen und weitere, mit dem System interagierende Systeme beinhalten kann. Die Umgebung grenzt den Wirkungsbereich des Systems gegen andere Systeme ab [vgl. ISO07, S. 4]. Der Begriff der Komponente soll an dieser Stelle noch nicht mit einer strengen Definition (z. B. im Sinne der CBSE, siehe Unterabschnitt 2.2.1) belegt, sondern im allgemeinen, intuitiven Sinne als Teil des Systems verstanden werden. Zur Abgrenzung gegen den strengen Komponentenbegriff wird im Rahmen dieser Arbeit die Bezeichnung „Systemkomponente“ verwendet. Die Allgemeinheit der Definition des System-Begriffs lässt es zu, einzelne Systemkomponenten wiederum als (Sub-)Systeme zu betrachten. Diese rekursive Anwendung des System-Begriffs ist sinnvoll und wird im Rahmen dieser Arbeit auch praktiziert. Im Kontext der Definition von Anforderungen an Plugin-Systeme soll diese Verwendung aber nach Möglichkeit vermieden werden, um Verwirrungen vorzubeugen.

Ein Softwaresystem ist grundsätzlich einem Produktlebenszyklus unterworfen, der im Rahmen der Softwaretechnik frei definiert werden kann. Wie der Produktlebenszyklus von Softwaresystemen im Bereich der CBSE aufgefasst wird, wurde bereits in Unterabschnitt 2.2.4 diskutiert. Für diese Arbeit soll in Anlehnung an die Einteilung von Lau und Wang in [LW07] zwischen einer *Entwicklungsphase*, in welcher Entwurf und Implementierung des Systems und seiner Komponenten stattfinden, einer *Einsatzphase*, in welcher ein System beim Kunden installiert ist, und der *Laufzeit*, während derer das System sich in Ausführung befindet, unterschieden werden. Jeder Produktlebenszyklus eines genutzten Softwaresystems wird ein oder mehrere Vorkommen jeder Phase beinhalten. Änderungen am System, die ein Neu-Übersetzen des Systems oder einiger Komponenten erfordern, sind nur in einer Entwicklungsphase möglich. In einer Einsatzphase können im Vergleich zu einer Entwicklungsphase die Entwickler, Administratoren oder Anwender des Systems häufig nicht oder nur in stark reduziertem Umfang in das System eingreifen, um die Zusammenfügung der Systemteile zu verändern. Während einer Laufzeit befindet sich mindestens eine Systemkomponente in Ausführung. Laufzeiten treten immer innerhalb von Einsatzphasen auf.

Basierend auf diesem Produktlebenszyklus können nun zwei Varianten von Rekonfigurierbarkeit unterschieden werden:

**Definition 2 (rekonfigurierbares System)** *In einem rekonfigurierbaren System kann die Zusammenfügung der Systemkomponenten* während seiner Einsatzphase *verändert werden.*

**Definition 3 (dynamisch rekonfigurierbares System)** *In einem dynamisch rekonfigurierbaren System kann die Zusammenfügung der Systemkomponenten* während seiner Laufzeit *verändert werden.*

Definition 3 weicht von dem in Unterabschnitt 2.2.4 beschriebenen Verständnis *dynamischer Änderungen* ab, da oftmals die dort als Bezugspunkt genannte *Laufphase* nicht

[1] Zitiert und übersetzt aus [ISO07, S. 3]: „*[A system is] a collection of components organized to accomplish a specific function or set of functions.*“

der in dieser Definition verwendeten *Laufzeit*, sondern der *Einsatzphase* entspricht. Eine Übereinstimmung besteht nur, falls Laufzeit und Einsatzphase zusammenfallen, wie z. B. bei einem dauerlaufenden Web-Server mit austauschbarer Anwendungsfunktionalität.

Im physikalischen Sinne (und im Sinne der Systemtheorie) ist ein *offenes System* eines, das im Gegensatz zum *geschlossenen System* Energie oder Materie mit seiner Umgebung austauschen kann. In den Disziplinen der verteilten Systeme und der Betriebssysteme wird mit dem Begriff *offenes System* eine unabhängig standardisierte Betriebsumgebung bezeichnet, welche die Mischung der Hardware, Software und Netzwerktechnik unterschiedlicher Hersteller zulässt [vgl. Sam97, S. 80]. Weit verbreitet ist darüber hinaus die Verwendung des Begriffs *offen* im Sinne von *frei zugänglich und verwendbar*, wie z. B. zur Bezeichnung von Software, die frei von Patenten oder lizenzrechtlichen Nutzungs- und Weiterentwicklungsbeschränkungen ist [vgl. FH08, S. 586]. Für das Thema dieser Arbeit ist es sinnvoll, sich an der physikalischen oder systemtheoretischen Definition zu orientieren. Übertragen auf Softwaresysteme bedeutet dies, dass ein *offenes System* Informationen oder Systemkomponenten mit der Umgebung austauschen kann. Da jedoch die Fähigkeit des Informationsaustausches mit der Umgebung für alle im Rahmen dieser Arbeit betrachteten Systeme aufgrund der Interaktion mit Anwendern grundsätzlich gegeben ist, eignet sie sich nicht als diskriminierende Eigenschaft. Somit bleibt die Diskriminierung über den Austausch von Systemkomponenten:

**Definition 4 (offenes System)** *Ein offenes System ist ein* rekonfigurierbares *System, dem Systemkomponenten* hinzugefügt *oder* entnommen *werden können.*

Maßgeblich ist also die Möglichkeit des Hinzufügens und Entfernens von Systemkomponenten. Ein offenes System kann, muss aber nicht, *dynamisch* rekonfigurierbar sein, so dass mit *offenen* und *dynamisch rekonfigurierbaren* Systemen zwei orthogonale Varianten von Rekonfigurierbarkeit definiert sind. Diese Definition eines offenen Systems deckt sich mit dem Begriff des *erweiterbaren Systems*, wie ihn viele (englischsprachige) Autoren verwenden. Da der Erweiterbarkeitsbegriff im Rahmen dieser Arbeit eine präzisere Definition benötigt (siehe Unterabschnitt 4.1.3), soll die Bezeichnung des offenen Systems Missverständnisse vermeiden helfen.

Das in dieser Arbeit entwickelte konzeptionelle Modell von Plugin-Systemen muss in der Lage sein, *rekonfigurierbare, offene Systeme* zu beschreiben. Konkret muss es Operationen zum Hinzufügen und Entfernen von Funktionalität während der Einsatzphase (in Form von Plugins) unterstützen. Da das konzeptionelle Modell auch die Variante *dynamisch rekonfigurierbarer* Plugin-Systeme abdecken soll, müssen diese Operationen sogar während der Laufzeit zur Verfügung stehen. Da Laufzeiten immer innerhalb von Einsatzphasen auftreten, stehen Operationen, die während der Laufzeit anwendbar sind, automatisch auch in einer Einsatzphase zur Verfügung. Es ist also nicht notwendig, im konzeptionellen Modell den Fall *nicht dynamisch rekonfigurierbarer* Systeme explizit zu behandeln.

Allerdings gibt es viele Varianten offener oder dynamisch rekonfigurierbarer Systeme. Beispielsweise ist ein Betriebssystem ein offenes, dynamisch rekonfigurierbares System, da dort während der Laufzeit Anwendungen installiert und wieder deinstalliert werden können. Manche komponentenbasierte Systeme sind dynamisch rekonfi-

gurierbar (siehe Unterabschnitt 2.2.4), manche Architekturbeschreibungssprachen ermöglichen die Beschreibung dynamisch rekonfigurierbarer Systeme (siehe Unterabschnitt 2.3.4), und Multiagentensysteme sind in der Regel offene Systeme. Zur stärkeren Abgrenzung von Plugin-Systemen gegen andere rekonfigurierbare Systeme wird im nächsten Unterabschnitt auf die Durchführung der Rekonfiguration eingegangen. Die folgenden Unterabschnitte ergänzen zudem einige Anforderungen, welche typische Eigenschaften von Plugin-Systemen im Hinblick auf die Softwarearchitektur festlegen.

### 4.1.2 Einfache Rekonfiguration

Plugin-Architekturen sollen ein Mittel sein, welches Anwendern das Zurechtschneidern (*tailoring*) der Funktionalität eines Softwaresystems auf ihre eigenen Bedürfnisse hin ermöglicht. Mørch hat in [Mør97] drei Ebenen herausgestellt, auf denen Anwender ein System auf ihre Bedürfnisse hin einrichten können: *Anpassung* (*Customization*) parametrisiert das Systemverhalten in engem Rahmen anhand der vom System zu diesem Zweck bereitgestellten Variablen. Ein Anwender kann *Integration* selber vornehmen, indem darauf ausgelegte Komponenten wahlweise in das System hineinkopiert (*Hard Integration*) oder mittels einfacher Skriptsprachen zu neuer Funktionalität kombiniert (*Soft Integration*) werden. *Erweiterung* (*Extension*) bezeichnet die Programmierung von Funktionalität durch den Anwender auf der Basis einer hierfür geeigneten (und im Softwaresystem bereitgestellten) Programmiersprache.

Übertragen auf die Definitionen des vorigen Unterabschnitts handelt es sich bei Softwaresystemen, welche *Integration* nach Mørch unterstützen, um *rekonfigurierbare Systeme*, da die Integration während der Einsatzphase oder gar während der Laufzeit durch den Anwender durchgeführt wird. Im Falle der *Hard Integration* kann von einem *offenen System* nach Definition 4 gesprochen werden. Die Möglichkeit der *Anpassung* von Softwaresystemen auf Basis angebotener Parametrisierungsvarianten stellt hingegen keine besonderen Anforderungen an die Architektur des Systems. Mørchs *Erweiterung*sbegriff geht über die im Rahmen dieser Arbeit diskutierten Möglichkeiten hinaus – Mørch beschreibt im Prinzip eine Idee, wie sie auch in der Eclipse-Community gepflegt wird, wenn aus Eclipse-Anwendern schrittweise Eclipse-Erweiterer werden (vgl. Unterabschnitt 3.2.1.8). Diese Idee ist meiner Meinung nach nur realisierbar, wenn die Anwender eines Softwaresystems auch zum weit gefassten Kreis der Softwareentwickler gehören. „Normalen“ Anwendern sollte das Erlernen einer Programmiersprache nicht zugemutet werden. Allenfalls im Rahmen der von Mørch als *Soft Integration* benannten Möglichkeit ist eine anwendungsorientierte Skriptsprache beispielsweise zur Automatisierung wiederkehrender Aufgaben zumutbar. Die Grenzen zur Softwareentwicklung verlaufen dabei allerdings fließend.

In der Praxis anzutreffende Plugin-Systeme sind bewusst darauf ausgelegt, vom Anwender als Monteur des Softwaresystems nicht mehr als einen Installations- (also Kopier-)vorgang zu verlangen. Diese Eigenschaft wird beispielsweise auch in [WDPM06] unter der Bezeichnung *Discovery* als Unterscheidungsmerkmal der Plugin-Systeme genannt (siehe Unterabschnitt 3.3.3). Techniken wie das XPInstall oder das Toolkit für Mozilla-Extensions (siehe Unterabschnitt 3.1.2.4) haben genau diese Vermeidung von Aufwand beim Anwender zum Ziel, indem die notwendige Konfigurationsprogrammierung in den Aufgabenbereich des Plugins oder des Rahmenwerkes verlagert wird.

Die „allereinfachste Integration“ als wesentliche Eigenschaft von Plugin-Systemen ist natürlich mit anderen Möglichkeiten des Zurechtschneiderns von Anwendungen kombinierbar, wie insbesondere die in Kapitel 3 vorgestellten Beispiele Eclipse und NetBeans zeigen, welche als Arbeitsumgebungen für Softwareentwickler vollen Zugriff auf die eigene Implementierung bieten und daher jede vom „Anwender“ programmierte Änderung und Erweiterung ermöglichen. Aber auch abseits der expliziten Softwareentwicklungsumgebungen erlauben viele gängige Bildbearbeitungs- oder Büroanwendungspakete die Automatisierung wiederkehrender Aufgaben mittels einer Makrosprache unter Rückgriff auf installierte Funktionalität. Nichtsdestotrotz ist die Erweiterung durch programmierende Anwender keine typische, notwendige Eigenschaft von Plugin-Systemen.

Das zu entwickelnde Plugin-Konzept muss einen Rahmen vorgeben, der eine einfache Integration von Komponenten durch Anwender in ein offenes, möglicherweise dynamisch rekonfigurierbares System ermöglicht bzw. unterstützt.

### 4.1.3 Erweiterbarkeit an vorgesehener Stelle

Offene Softwaresysteme sind gemäß Definition 4 grundsätzlich in einem allgemeinen Sinne erweiterbar, da in der Einsatzphase neue Komponenten hinzugefügt werden können, die zusätzliche Funktionalität bereitstellen. Intuitiver erscheint jedoch ein engeres Verständnis von Erweiterbarkeit: Demnach ist ein System erst dann erweiterbar zu nennen, wenn im System eine oder mehrere Möglichkeiten zu seiner Erweiterung explizit vorgesehen sind. Dieses Verständnis erzwingt beim Entwurf der Softwarearchitektur eines erweiterbaren Systems eine Festlegung, auf welche Weise zusätzliche Funktionalität in das System integriert werden soll und kann.

Im vorigen Unterabschnitt wurde bereits betont, dass der Erweiterungsbegriff, wie Mørch ihn in [Mør97] definiert, nicht zur Betrachtung von Plugin-Systemen geeignet ist, da er die Programmierung von Erweiterungen durch Anwender bezeichnet. Szyperskis Begriff des *erweiterbaren Systems*, den er in [Szy96] verwendet, ist sehr allgemein formuliert. Er deckt sich mit dem Begriff eines dynamisch rekonfigurierbaren, offenen Systems nach Definition 3 und Definition 4. Von einem *unabhängig erweiterbaren System* erwartet Szyperski, dass es das „späte“ Hinzufügen von Komponenten ohne die Notwendigkeit einer globalen Integritätsprüfung verkraftet. Mit dieser Bedingung geht er zwar einen Schritt über die Definitionen des Unterabschnitts 4.1.1 hinaus, konkretisiert die Integrationsweise aber nicht. Marquardt bezeichnet in [LM08] Plugins als Erweiterungen für Systeme, welche *vorgesehenerweise erweiterbar* sind.[2]

Meyer erläutert in [Mey97, Abschnitt 14.8], wie im objektorientierten Programmierparadigma mittels Verhaltensklassen (*behavior class*) oder aufgeschobener Klassen (siehe Unterabschnitt 2.4.5) so genannte „Programme mit Löchern“ [*programs with holes*, Mey97, S. 505] implementiert werden können. Anders als beim üblichen Bild von Komponenten als feste Bausteine, welche flexibel zu vielfältigen Strukturen kombiniert werden können, sei bei „Programmen mit Löchern“ die Struktur vorab festgelegt, während die Steine verändert werden könnten. Dieses Konzept schließt die Möglichkeit ein,

[2] „*A plug-in is an extension to an application that is intended to be extended.*“ [LM08, min. 2:35–2:41]

ein Code-Modul zunächst nur mit Platzhaltern zu übersetzen, so dass es seinen Nutzen nur nach Einsatz der tatsächlichen Komponenten innerhalb seiner Struktur entfaltet. Damit trifft Meyer die Kernidee von Plugin-Systemen, auch wenn er vermutlich weniger an dynamisch rekonfigurierbare Systeme nach Definition 3 als an die Nutzung von objektorientierten Rahmenwerken (was „Programme mit Löchern“ faktisch sind) in der Entwicklungsphase gedacht hat.[3]

Meyer bezeichnet „Programme mit Löchern“ als objektorientierte Variante der *Callback*-Technik, welche auch von Szyperski in [Szy02, Abschnitte 5.4–5.8] als Option zur Realisierung komponentenbasierter Systeme diskutiert wird. Dabei übergibt eine erweiternde Komponente eine Rückrufadresse an eine vom Rahmenwerk bereitgestellte Prozedur, um vom Rahmenwerk bei Bedarf über Veränderungen benachrichtigt oder zum Erbringen von Funktionalität aufgefordert werden zu können. In der objektorientierten Variante wird ein Rückrufobjekt übergeben, welches gegenüber der Rückrufprozedur sowohl eine höhere Flexibilität als auch eine stärkere Kapselung aufweisen kann. Szyperski betont, dass die durch die *Callback*-Technik eingeführten Möglichkeiten nur schwer in einem Schnittstellenvertrag zu bändigen sind, da die Zwischendurch-Ausführung von Code aus *Callback*-Prozeduren oder -Objekten die Atomizität von Operationen der aufrufenden Systemkomponente unterbricht. Wie in Abschnitt 2.4 bereits dargelegt, kommen in Plugin-Systemen häufig Entwurfsmuster zum Einsatz, die auf der *Callback*-Technik aufsetzen, wie z. B. beim Beziehungsaufbau im Magic-Beans-Rahmenwerk (siehe Unterabschnitt 3.3.4).

Von den in Unterabschnitt 3.4.2 diskutierten Beziehungsarten passt vor allem die Erweiterungsbeziehung gut zur *Callback*-Technik bzw. zur Idee der „Programme mit Löchern“. Im Unterschied zu allen anderen beobachteten Beziehungsarten werden sowohl die Beziehungsdauer als auch die Schnittstellenspezifikation in der Erweiterungsbeziehung von derselben Rolle festgelegt, nämlich dem erweiterbaren System (siehe Tabelle 3.3). Die zweite beteiligte Rolle, die Erweiterung, muss sich nach den Vorgaben des erweiterbaren Systems richten, kann durch diese Beschränkung aber leichter in das System integriert werden.

Der Begriff *erweiterbares System*, der jetzt zu definieren ist, wird in der Softwaretechnik (insbesondere in der CBSE), allgemein üblich in einer schwächeren Weise verwendet, welche nur grundlegende technische, aber keine fachlichen Vorgaben des erweiterbaren Systems zu seinen Erweiterungsmöglichkeiten vorsieht. Um die Bedeutung der Intention der Architekten des erweiterbaren Systems bei der Vorgabe von Erweiterungsmöglichkeiten zu betonen, soll in dieser Arbeit stattdessen folgende Definition verwendet werden:

> **Definition 5 (intendiert erweiterbares System)** *Ein intendiert erweiterbares System ist ein offenes System, welches technische und fachliche Vorgaben zur Integration von während einer Einsatzphase hinzufügbaren Systemkomponenten macht.*

[3] Wenn Meyer von wiederverwendbaren Komponenten (*reusable components*) spricht, bezieht er sich auf objektorientierte Elemente, die in der Entwicklungsphase zur Verfügung stehen, z. B. so genannte „abstrahierte Module“ (*abstracted module*). Ein Indikator für diese Sichtweise ist, dass er die Auslieferung von Komponenten in Quellcodeform zulässt [vgl. Mey97, S. 73].

**Definition 6 (Erweiterung)** *Eine Erweiterung eines intendiert erweiterbaren Systems ist eine dem System hinzufügbare Systemkomponente, deren Funktionalität entsprechend der Vorgaben des intendiert erweiterbaren Systems integriert werden kann bzw. muss.*

Da in dieser Arbeit bis auf wenige Ausnahmen nur intendiert erweiterbare Systeme betrachtet werden, wird der Zusatz „intendiert“ häufig nicht erwähnt. In Situationen, in denen der allgemeinere Begriff der Erweiterbarkeit benötigt wird, wird explizit darauf hingewiesen.

Ein intendiert erweiterbares System bietet erweiterungsunabhängige Kernfunktionalität, welche auf wohldefinierte Weise durch Erweiterungen beeinflusst oder um weitere Funktionalität ergänzt werden kann. Als offenes System stellt das intendiert erweiterbare System seiner Umgebung mindestens die Operationen *Hinzufügen* und *Entfernen* von Erweiterungen während der Einsatzphasen zur Verfügung. Definition 5 schließt nicht aus, dass die Definition von Vorgaben zur Integration von Erweiterungen innerhalb einzelner Systemkomponenten stattfindet, so dass die Eigenschaft *intendiert erweiterbar* auch auf diese Systemkomponenten zutrifft. Falls ein offenes System eine erweiterbare Systemkomponente enthält, sind sowohl die Komponente als auch das System entsprechend erweiterbar.

Der Begriff Erweiterung kann immer nur mit Bezug auf das zu erweiternde System bzw. die zu erweiternde Systemkomponente verwendet werden. Eine Erweiterung eines intendiert erweiterbaren Systems muss sich an die Vorgaben des Systems zu seiner Erweiterung halten. Das erweiterbare System ist nicht auf die Funktionalität konkreter Erweiterungen angewiesen, die Integration von Erweiterungen ist *optional* [vgl. Cha05, S. 36].

Ein intendiert erweiterbares System kann verschiedene Klassen von Erweiterungen unterscheiden und jeweils spezifische Vorgaben festlegen. In der Eclipse-Architektur wird diesem Umstand durch die Möglichkeit der Deklaration mehrerer Erweiterungspunkte (*extension points*, siehe Unterabschnitt 3.2.1.5) Rechnung getragen. Gleichermaßen können in der CAP.NET-Architektur mehrere *extension slots* definiert werden (siehe Unterabschnitt 3.3.3). Marquardt verwendet dieses Konzept noch in [MV03, S. 22] unter der Bezeichnung *Integrationspunkt*, verzichtet aber in [Mar06, S. 307 und 314] darauf, da er stattdessen einem Softwarearchitekten die Möglichkeit einräumt, mehrere Plugin-Arten (*plug-in-kind*) zu unterscheiden. Zu jeder Plugin-Art wird ein eigener Plugin-Vertrag innerhalb des Systems definiert (vgl. Unterabschnitt 2.4.3). Auch in dieser Arbeit soll anstelle der eher technisch orientierten Sicht auf einen Erweiterungspunkt die architekturorientierte Sicht auf eine Erweiterungsspezifikation eingenommen werden:

**Definition 7 (Erweiterungsspezifikation)** *Eine Erweiterungsspezifikation umfasst sämtliche Vorgaben, die ein intendiert erweiterbares System einer Klasse ähnlich gearteter Erweiterungen macht.*

Der Klassenbegriff entspricht hier dem Begriff der Plugin-Art bei Marquardt. Die Einteilung denkbarer Erweiterungen in unterschiedliche Klassen obliegt dem Architek-

ten des erweiterbaren Systems und findet in aller Regel nach funktionalen[4] Gesichtspunkten statt, also nach einer Unterscheidung der Funktionalität der vorgesehenen Erweiterungsmöglichkeiten. Die Vorgaben einer Erweiterungsspezifikation umfassen sämtliche Festlegungen von der Anwendungsdomäne über die nutzbare Infrastruktur des erweiterbaren Systems bis zur Deklaration und Verhaltensbeschreibung des technischen Interfaces, welches von einer Erweiterung zu implementieren ist. Marquardt und Völter weisen in [MV03, S. 23] darauf hin, dass ein Plugin-Vertrag (der einer Erweiterungsspezifikation vergleichbar ist) das globale Wissen über die gesamte Domäne umfasst. Einige dieser Vorgaben werden in den folgenden Unterabschnitten noch näher diskutiert.

In Tabelle 3.3 ist die Erweiterungsbeziehung als $1 : n$-Beziehung charakterisiert worden, die einem erweiterbaren System mehrere Erweiterungen zuordnen kann. Diese Einordnung kann nun so konkretisiert werden, dass beliebig viele Erweiterungen zu genau einer Erweiterungsschnittstelle zugeordnet werden. Von diesen Schnittstellen kann es aber mehrere im erweiterbaren System geben, so dass wir zu einer $1 : m : (\Sigma_{i=1}^{m} n_i)$-Beziehung kommen, welche einem System mehrere verschiedene Spezifikationen und jeder Spezifikation wieder mehrere passende Erweiterungen zuordnen kann.

#### 4.1.3.1 Diskussion

Um den Unterschied zu alternativen Architekturen offener Systeme zu verdeutlichen, sei hier im Vergleich die im Rahmen der (allgemeinen) Erweiterbarkeit offener Systeme ebenfalls zur Auswahl stehende Dienstbeziehung definiert.

**Definition 8 (dienstanbietendes System)** *Ein dienstanbietendes System stellt während einer Einsatzphase seine Funktionalität gemäß einer technischen und fachlichen Dienstspezifikation zur Integration in andere Systeme oder eigene Systemkomponenten bereit.*

**Definition 9 (dienstnutzendes System)** *Ein dienstnutzendes System integriert während einer Einsatzphase Funktionalität von beliebigen, zu einer gegebenen Dienstspezifikation passenden, dienstanbietenden Systemen, um seine eigene Funktionalität zu erbringen.*

**Definition 10 (Dienstspezifikation)** *Eine Dienstspezifikation umfasst sämtliche Vorgaben, die zur Integration der Funktionalität eines passenden dienstnutzenden Systems in die eines passenden dienstanbietenden Systems notwendig sind.*

Ein dienstanbietendes System erbringt eine Funktionalität, welche auf wohldefinierte Weise in die Funktionalität von dienstnutzenden Systemen integriert werden kann. Bei dieser Betrachtung wird die Dienstbeziehung außerhalb der Grenzen beider Systeme aufgebaut, ebenso ist die Dienstspezifikation nicht notwendigerweise Teil eines der beiden Systeme. Da Spezifikation sowie anbietendes und nutzendes System Teil eines größeren Ganzen sind, können sie auch einfacher als Systemkomponenten eines äußeren

[4] Das Adjektiv „*funktional*“ wird in weiten Teilen dieser Arbeit nicht im mathematischen Sinne, sondern in der Bedeutung „*funktionalitätsorientiert*“ verwendet.

rekonfigurierbaren Systems betrachtet werden. Die dienstanbietende Systemkomponente wird auch kurz *Dienstanbieter* genannt, die dienstnutzende Systemkomponente *Dienstnutzer*. Das äußere System wird nur dann dienstanbietend oder dienstnutzend genannt, wenn es selber an seiner Systemgrenze Funktionalität zur Integration anbietet bzw. einfordert, also dieses Anbieten oder Einfordern nicht erst aufgrund von hinzugefügten Systemkomponenten stattfindet.

Die Begriffe der Dienstbeziehung lassen sich noch allgemeiner definieren, wenn die Einschränkung auf Integration *während der Einsatzphase* fallen gelassen wird. Der Begriff des Dienstes wird traditionell auch beim Entwurf von Architekturen monolithischer Softwaresysteme oft als Metapher eingesetzt. Zudem wird gerade in der CBSE die Dienstbeziehung oft zur Integration von Komponenten herangezogen, die Komposition eines komponentenbasierten Systems durch einen System-Monteur ist aber in vielen Fällen der Entwicklungsphase zuzurechnen (vgl. Unterabschnitt 2.2.4). Somit kann im allgemeinen Sprachgebrauch ein System aus Dienstanbietern und -nutzern *dynamisch rekonfigurierbar* oder *offen* sein, muss es aber nicht. Im Rahmen dieser Arbeit sollen jedoch Dienstbeziehungen nur im Kontext offener Systeme diskutiert werden, um eine Vergleichbarkeit mit der eben definierten Erweiterungsbeziehung zu erreichen.

Beide Begriffe, sowohl Dienstanbieter als auch Dienstnutzer, können nur mit Bezug auf eine *Dienstspezifikation* verwendet werden. Die Dienstspezifikation legt die Art und Weise des Dienstangebots und der Dienstnutzung so weit fest, dass – wie in Definition 10 gefordert – alle Beteiligten zueinander passen. Nichtsdestotrotz können in einem System mehrere voneinander unabhängige Dienstspezifikationen existieren. Anhand der jeweils zugrundeliegenden Dienstspezifikation können dann Dienstanbieter und -nutzer in unterscheidbare Klassen eingeteilt werden, so dass alle Anbieter und Nutzer innerhalb einer Klasse zueinander passen. Die so erreichte Klassifizierung unterschiedlicher Dienstanbieter und -Nutzer funktioniert somit analog zur klassifizierenden Verwendung der Erweiterungsspezifikation in Definition 7.

Die Einteilung von Dienstanbietern und Nutzern in Klassen muss zwischen allen Architekten von Systemkomponenten des offenen Systems ausgehandelt werden und findet ebenso wie bei der Erweiterungsbeziehung nach funktionalen Gesichtspunkten statt. Falls eine Dienstspezifikation nicht zwischen mehreren Architekten ausgehandelt wird, kann auch der Architekt der ersten „passenden“ Systemkomponente einer Klasse die Dienstspezifikation festsetzen. In der CBSE (siehe Abschnitt 2.2) findet sich häufig das Szenario, dass der Entwickler einer generisch wiederverwendbaren Komponente quasi einen Dienstanbieter inklusive Dienstspezifikation erstellt. Das selbstspezifizierte Dienstangebot dieser Komponente kann sowohl von beliebig vielen, später implementierten, passenden Komponenten oder durch individuellen Code bei der Montage eines Systems genutzt werden.

Die Vorgaben einer Dienstspezifikation umfassen sämtliche Festlegungen von der Anwendungsdomäne über die nutzbare Funktionalität des anbietenden Systems bis zur Deklaration und Verhaltensbeschreibung des technischen Interfaces, welches von Nutzer und Anbieter zu implementieren ist. In Tabelle 3.3 ist die Dienstbeziehung als $n : 1 : m$-Beziehung mit beliebig vielen Anbietern und Nutzern, deren Gemeinsamkeit die einheitliche Dienstspezifikation ist, charakterisiert worden.

Im direkten Vergleich der Definitionen 5–7 mit den Definitionen 8–10 fallen sowohl Gemeinsamkeiten als auch Unterschiede auf. Zum Teil wurden diese schon in Unterabschnitt 3.4.2 erkannt oder in die vorangegangene Erläuterung der Definitionen eingeflochten.

Während die Vorgaben in der Erweiterungsbeziehung vom intendiert erweiterbaren System definiert und in der Erweiterungsspezifikation festgehalten werden, ist in der Dienstbeziehung die Dienstspezifikation als separates Element für die Festlegung der Vorgaben zuständig. Die jeweils anderen an der Beziehung Beteiligten (also im Falle des erweiterbaren Systems die passende Erweiterung und im Falle der Dienstspezifikation die passenden Anbieter und Nutzer) haben sich nach den Vorgaben zu richten. Allerdings kann die Separiertheit der Dienstspezifikation abgeschwächt werden, wenn ein konkurrenzloser Dienstanbieter oder -nutzer die Definition der Spezifikation übernimmt.

Das Verhältnis zwischen bereitstellenden und integrierenden Systemkomponenten weist bei den beiden Beziehungsarten verschiedene Richtungen auf, wenn man es mit dem jeweiligen Verhältnis zwischen spezifizierender und spezifikationskonformer Systemkomponente vergleicht. Das erweiterbare System (als Nutzer der Erweiterungsfunktionalität) integriert die Erweiterung und gibt gleichzeitig die Spezifikation vor. Die dienstnutzende Systemkomponente (als Nutzer der Funktionalität des Dienstanbieters) integriert zwar den Dienstanbieter, muss sich aber an eine in der Regel von anderer Seite vorgegebene Spezifikation halten. Bei der Erweiterungsbeziehung fallen also die Aufgaben der Spezifikation und der Integration derselben Systemkomponente zu, während bei der Dienstbeziehung Spezifikation und Integration Aufgaben verschiedener Komponenten sind. Im eben geschilderten Szenario der CBSE, in welchem der Dienstanbieter seine eigene Dienstspezifikation mitbringt, sind also die Aufgaben der Spezifikation und der Integration auf zwei einander gegenüber stehende Komponenten verteilt.

Ein weiterer Unterschied zwischen beiden Beziehungsarten lässt sich in der Richtung der Nutzung von Funktionalität ausmachen. In der Dienstbeziehung stellt ausschließlich der Dienstanbieter Funktionalität zur Integration bereit, er kann dabei nicht auf Funktionalität oder Eigenschaften des Dienstnutzers zurückgreifen. In der Erweiterungsbeziehung hingegen stellt nicht nur die Erweiterung Funktionalität zur Integration zu Verfügung, sondern sie darf auch Funktionalität des intendiert erweiterbaren Systems nutzen, sofern diese in der Erweiterungsspezifikation angeboten wird.

Erweiterungs- und Dienstbeziehung prägen zudem die Grobstruktur einer Softwarearchitektur auf unterschiedliche Weise. Die beteiligten Partner der Dienstbeziehung liegen häufig auf einer Ebene, wenn Systeme Dienste anderer Systeme nutzen oder eine Beziehung zwischen Systemkomponenten desselben Systems aufgebaut wird. Zwar lässt sich durch die Zuordnung von Dienstanbietern und -nutzern zu unterschiedlichen Schichten eine Ordnung in eine Softwarearchitektur einbringen, diese muss aber nicht notwendigerweise direkt mit den im System aufgebauten Dienstbeziehungen korrespondieren.

Die Erweiterungsbeziehung hingegen verlangt, dass eine Erweiterung als Systemkomponente in ein äußeres System eingefügt ist, so das automatisch eine hierarchisch organisierte Softwarearchitektur entsteht. Im Falle reiner Plugin-Systeme, welche jeder Systemkomponente die Möglichkeit zugestehen, selber wiederum durch ande-

re Systemkomponenten erweiterbar zu sein, könnte zwar eine flache Systemarchitektur aus gleichberechtigten Systemkomponenten vermutet werden. Dennoch ist auch in diesem Falle in der Natur der Erweiterungsbeziehung eine hierarchische Organisationsform angelegt. Falls eine Systemkomponente eine andere Systemkomponente desselben äußeren Systems erweitert, besteht zwischen den beiden Systemkomponenten eine asymmetrische Beziehung, da die Erweiterung zur Sub-Systemkomponente der erweiterbaren Systemkomponente wird und ohne jene die eigene Funktionalität gar nicht zur Geltung bringen könnte.

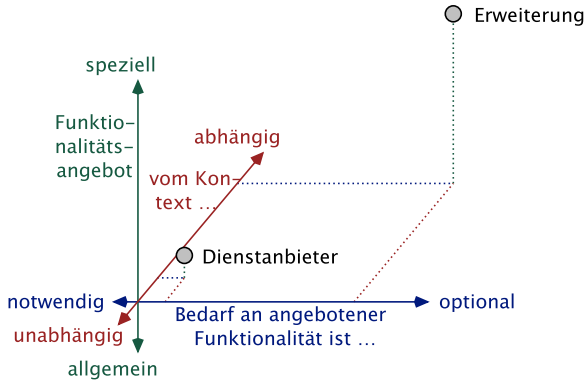



Abbildung 4.1: Einordnung von Erweiterungs- und Dienstbeziehung

Die unterschiedliche Verwendbarkeit der beiden Beziehungsarten wird in Abbildung 4.1 grafisch anhand der jeweils Funktionalität anbietenden Systemkomponenten (also Erweiterung bzw. Dienstanbieter) dargestellt. Auf jeder der Achsen kann die Einordnung der Systemkomponenten bei beiden Beziehungsarten variieren – die Graphik stellt nur eine Tendenz dar. Die Funktionalität von Dienstanbietern ist für den Dienstnutzer oftmals essentiell zur Erbringung seiner eigenen Funktionalität. Dementgegen steht die Möglichkeit erweiterbarer Systeme, auch ohne Erweiterungen bereits eine Kernfunktionalität erbringen zu können. Während die Erweiterung von ihrer Funktionalität her speziell auf ein erweiterbares System abgestimmt ist, werden Dienstanbieter in der Regel so konzipiert, dass ihre Funktionalität mehreren Dienstnutzern nutzen kann. Ein Dienstanbieter soll vom Dienstnutzer keine Funktionalität verlangen, sondern möglichst unabhängig zur Verfügung stehen. Dementgegen darf eine Erweiterung in starkem Umfang auf Funktionalität ihres Kontextes, also des erweiterbaren Systems, zurückzugreifen, wodurch sie vom Kontext abhängig wird.

Wie in Abbildung 4.1 zu erkennen ist, sind die Übergänge zwischen den auf jeder Achse zur Charakterisierung der Beziehungsarten aufgeführten Polen fließend. So ließe sich das Integrieren einer Erweiterung als Dienstleistung des erweiterbaren Systems gegenüber der Erweiterung verstehen, alternativ könnte ein intendiert erweiterbares System als Dienstnutzer aufgefasst werden, welcher Dienstangebote von Erweiterungen in Anspruch nähme. Zu beachten ist jedoch, dass die Beziehung zwischen intendiert erweiterbarem System und einer passenden Erweiterung als *doppelseitige* Dienstbe-

ziehung aufgefasst werden sollte. Das erweiterbare System agiert nicht nur als Nutzer der Erweiterungsfunktionalität, sondern auch als Dienstanbieter, da es als Kontext der Erweiterung Funktionalität bereitstellt.

Beide Beziehungsarten sind prinzipiell in der Lage, Systemkomponenten in beliebigen rekonfigurierbaren, offenen Systemen zu koppeln. Da keine Unterschiede in der Mächtigkeit der Beziehungsarten bestehen, ist es die Aufgabe der Architekten des Softwaresystems, die jeweils nach fachlichen oder technischen Gesichtspunkten passendere Beziehungsart im Entwurf der Softwarearchitektur eines offenen Systems auszuwählen. Die Wahl zwischen der Dienstbeziehung und der Erweiterungsbeziehung als Metapher zur Integration der Systemkomponenten eines offenen Systems hat Auswirkungen auf die resultierende Softwarearchitektur.

Der Umfang der zu erbringenden Funktionalität wird in der Dienstbeziehung in der Regel vom Dienstanbieter vorgegeben oder unabhängig von Anbieter und Nutzer in der Dienstspezifikation definiert. Funktionalitätsergänzungen im System sind möglich, indem hinzugefügte Systemkomponenten auf bereits angebotene Dienstleistungen zurückgreifen und diese individuell zu neuer Funktionalität kombinieren. Dies entspricht einem bottom-up-Vorgehen der Softwareentwicklung, Ergänzungen finden quasi „außen“ um das bestehende System herum statt.

Im Falle der intendierten Erweiterbarkeit wird der Umfang einer von Erweiterungen zu erbringenden Leistung vom Nutzer der Erweiterungen – also dem erweiterbaren System oder der erweiterbaren Systemkomponente – vorgegeben. Funktionalitätsergänzungen sind somit nur im vorgesehenen Rahmen möglich, dafür wird die Integration mit der bestehenden Funktionalität vereinheitlicht und vereinfacht. Dies entspricht einem top-down-Vorgehen der Softwareentwicklung, Ergänzungen finden also „im Inneren“ des Systems statt.

In der Praxis sind hybride offene Systeme anzutreffen, welche beide Varianten der Integration von Funktionalität beherrschen. Es ergibt sich meist eine flache Organisation – ein Rahmenwerk verbindet sämtliche Systemkomponenten, welche gleichberechtigt nebeneinander liegen und in beliebigen Paarungen Dienst- oder Erweiterungsbeziehungen eingehen. Die flache Organisation kommt der Natur der Dienstbeziehung entgegen, in der alle beteiligten Partner gleichberechtigte Systemkomponenten eines äußeren Systems sind. Die Erweiterungsbeziehung betont dagegen eine Asymmetrie zwischen erweiterbarem System und einer Erweiterung als Systemkomponente, welche eigentlich eine hierarchische Organisation nahelegen sollte. In den existierenden hybriden Systemen wird daher diese strukturell prägende und konzeptionell klärende Eigenschaft der Erweiterungsbeziehung nicht ausgenutzt und so eine Erosion der Softwarearchitektur begünstigt.

Das zu entwickelnde Plugin-Konzept soll in der Lage sein, hybride Systeme zu modellieren, dabei aber einen Schwerpunkt auf den hierarchischen Charakter der Erweiterungsbeziehung legen, der der Plugin-Metapher näher kommt und bisher in Software-Architekturen trotz seiner Relevanz eher vernachlässigt wird.

### 4.1.4 Kapselung

Offene Systeme benötigen klare Strukturen, um hinzufügbare oder entfernbare Systemkomponenten sauber vom restlichen System trennen zu können. Eine integrierbare

Systemkomponente muss so gekapselt sein, dass alle Voraussetzungen zur Integration über ihre Schnittstelle zum System und zu anderen Komponenten eindeutig beschrieben sind. Dies gilt natürlich auch für Plugin-Systeme.

Jedes der in den Kapiteln 2 und 3 betrachteten Softwareentwicklungsparadigmen und Plugin-Systeme verwendet ein Konzept zur Kapselung von Systemkomponenten, namentlich Komponenten, Agenten, Module, Plugins, Pakete oder Bundles (vgl. Tabellen 2.3 und 3.2). Das Konzept des *Moduls* kann hierbei als kleinster gemeinsamer Nenner betrachtet werden, auf welchem die anderen Konzepte aufbauen. Module bündeln mehrere softwaretechnische Elemente wie Klassen, Objekte, Funktionen, Prozeduren oder andere Ressourcen zu einer softwaretechnischen Einheit, einem (Sub-) System mit einer klar definierten Systemgrenze.

Szyperski definiert ein Modul als „Menge von Klassen oder anderen, nicht-objektorientierten Elementen wie Prozeduren oder Funktionen“ [Szy02, S. 420]. Zwischen Modulen können statische Abhängigkeiten bestehen, so dass ein Modul nicht ohne die Module, von denen es abhängt, einsetzbar ist (vgl. Unterabschnitt 3.4.2). Szyperski spart dabei so genannte „Ressourcen“, also Artefakte, die nicht von einem Compiler erzeugt wurden, aus dem Modulbegriff aus, um sie erst im Komponentenbegriff wieder einzubinden. Diese feine Unterscheidung nach der Herkunft der Elemente trägt nicht zur Klärung der Konzepte im Rahmen dieser Arbeit bei und wird daher nicht weiter verfolgt.

Zur Abgrenzung zwischen den Konzepten von Modulen und Komponenten eignen sich eher die in Unterabschnitt 2.2.1 diskutierten Eigenschaften der „Komposition durch Dritte“ [Szy02, S. 41] und der „Konformität zu einem Komponentenmodell“ [CH01, S. 7]. Streng genommen verfügen auch Module bereits über diese Eigenschaften: Sie können als Artefakte in Quellcode- oder Binärform zwischen Entwicklern ausgetauscht werden, sind also *handelbar*. Sie müssen zudem in Binärform zu den Schnittstellen des verwendeten Linkers kompatibel sein, um mit anderen Modulen zu einem vollständigen Programm verbunden werden zu können. Allerdings werden Komponierbarkeit und Konformität bei Modulen nicht als unabdingbar vorausgesetzt, da sie mitunter ausschließlich zur internen Verwendung durch dieselben Entwickler vorgesehen sind.

Parnas hat in [Par72] technische und fachliche Kriterien zur Abgrenzung von Modulen im Software-Entwurf definiert, die sich am Prinzip des *Information Hiding* orientieren. Diese Kriterien führten vom Basiskonzept des abstrakten Datentyps zur Entwicklung des objektorientierten Softwareentwicklungsparadigmas. Für offene Systeme sind diese Kriterien noch stärker maßgeblich als für andere Systeme. Die Funktionalität der hinzufüg- und entfernbaren Systemkomponenten muss klar umrissen sein. Dies ist nur möglich, wenn ihre Schnittstellen einen überschaubaren Umfang haben – ein Umstand, der wiederum durch das Information-Hiding-Prinzip befördert wird.

Das zu entwickelnde Plugin-Konzept muss ein modularisierendes Konzept enthalten, welches die eindeutige Abgrenzung von Systemkomponenten (Modulen) zulässt. Die Schnittstellen der Module müssen eindeutig und vollständig definierbar sein, so dass keine über die Schnittstellenspezifikation hinausgehende Information zur Integration der Module notwendig ist. Diese Anforderung verlangt nach einer weiträumigeren Auslegung des Begriffs *Schnittstelle* als er in vielen objektorientierten Programmiersprachen mit dem Konzept der *Interfaces* realisiert ist (siehe hierzu auch

Unterabschnitt 4.2.6). Insbesondere zählen zur Schnittstelle eines Moduls nach dieser Betrachtung nicht nur Signaturen öffentlicher Methoden, Prozeduren oder Funktionen, sondern auch jene Klassen, Objekte, Prozeduren oder Funktionen, die im Rahmen der Kommunikation mit anderen Systemkomponenten verwendet werden (sowohl eigene als auch diejenigen, die in anderen Modulen genutzt werden, also statische Abhängigkeiten) sowie die Protokolle und Verträge, welche Bedingungen und Abläufe in der Verwendung der Schnittstellen vorgeben.

### 4.1.5 Lebenszyklen

In Unterabschnitt 4.1.1 wurde bereits angesprochen, dass jedes Softwaresystem einem groben Produktlebenszyklus aus Entwicklungs-, Einsatz- und Laufzeiten unterliegt. In einem offenen System, dessen Systemkomponenten zu unterschiedlichen Zeitpunkten von verschiedenen Entwicklern hergestellt werden, verfügt notgedrungen jede einzelne Systemkomponente über einen eigenen Produktlebenszyklus. Für die Betrachtung von Plugin-Systemen sind in diesem Zusammenhang allerdings weniger die voneinander unabhängigen Entwicklungsphasen der einzelnen Plugins interessant, sondern vielmehr die Verflechtung der Produktlebenszyklen in gemeinsamen Einsatzphasen und Laufzeiten. Der Lebenszyklus von Plugins in einem Plugin-System muss klar definieren, wann ein Plugin zum System hinzugefügt und wieder aus dem System entfernt wird.

Marquardt unterscheidet in [Mar06, S. 322, siehe auch Unterabschnitt 2.4.3] zwischen den Lebenszyklen von Plugin-Implementierungen (*plug-in type*, $\hat{=}$ Modulen) und deren Instanzen. Der Lebenszyklus einer Plugin-Implementierung müsse mindestens die Registrierung umfassen. Minimale Zustandsübergänge einer Instanz sollten nach Marquardt Laden, Aktivieren, Deaktivieren und Entladen sein.

In den in Kapitel 3 untersuchten Systemen besteht eine 1:1-Beziehung zwischen Plugin-Implementierung und Instanz. Auch wenn – wie z. B. bei NPAPI-Plugins, siehe Unterabschnitt 3.1.1.3 – die Funktionalität in mehreren Instanzen genutzt werden kann, so gibt es doch in allen Systemen zur Laufzeit nur ein einziges Repräsentanten-Objekt für jedes Plugin. Dies entspricht der von Szyperski im Kontext der CBSE vertretenen Ansicht, dass Komponenten nicht als Ganzes instantiierbar seien (siehe Unterabschnitt 2.2.1).

Marquardt unterscheidet im Prinzip nur zwischen Zustandsübergängen, die auch in der Einsatzphase (Registrieren) oder nur zur Laufzeit (Laden, Aktivieren usw.) stattfinden können. Das im Rahmen dieser Arbeit zu erstellende konzeptionelle Modell beschreibt *dynamisch* rekonfigurierbare Systeme, da sich das Verhalten *nicht-dynamischer* Systeme als Spezialfall in diesem Modell abbilden lässt (siehe Unterabschnitt 4.1.1 und Unterabschnitt 4.2.6.1). Da in einem dynamisch rekonfigurierbaren System *sämtliche* Übergänge zur Laufzeit stattfinden, hat Marquardts Unterscheidung für diese Arbeit keine Relevanz.

Ein weiterer Ansatz zur Definition von Lebenszyklen ergibt sich aus dem im vorigen Unterabschnitt sehr umfassend eingeführten Konzept der Schnittstelle: Teil jeder Schnittstellenspezifikation müssen die zulässigen Abläufe in der Schnittstellennutzung sein. In einem offenen System gilt dies natürlich auch für die Schnittstellen, welche das Hinzufügen bzw. Entfernen von Komponenten erlauben. Die zulässigen Abläufe bei

diesen Operationen definieren wesentliche Übergänge im Lebenszyklus der hinzugefügten bzw. entfernten Systemkomponente. Ebenso erzwingen die in Unterabschnitt 4.1.2 geforderten Mechanismen zur einfachen Konfiguration sowie der Auf- und Abbau der in Unterabschnitt 4.1.3 besprochenen Erweiterungs- und Dienstbeziehungen jeweils bestimmte Abläufe, die als elementare Bestandteile eines Plugin-Lebenszyklus betrachtet werden können.

**Definition 11 (Lebenszyklus)** *Ein offenes und intendiert erweiterbares System muss seinen Systemkomponenten einen eindeutigen Lebenszyklus vorgeben, welcher die Reihenfolge sämtlicher notwendigen Zustände und Schritte beim Hinzufügen und Entfernen der Systemkomponente während der Einsatzphase spezifiziert.*

Die notwendigen Zustände und Schritte umfassen nicht nur das Überschreiten der Systemgrenze des offenen Systems durch eine Systemkomponente, sondern müssen – gerade im Falle eines intendiert erweiterbaren Systems – auch essentielle Konfigurationsschritte zur Integration der Komponente ins System berücksichtigen. Auf die Repräsentation des Wechsels zwischen Einsatzphasen und Laufzeiten kann bei der Definition eines Lebenszyklus verzichtet werden, wenn solche Wechsel für das offene System nicht wahrnehmbar sind (z. B. falls das System ausschließlich während der Laufzeit konfiguriert wird).

Das zu entwickelnde Konzept muss also eine klare Modellierungsmöglichkeit für den ablauforientierten Anteil von Schnittstellenspezifikationen, insbesondere den Lebenszyklus von Systemkomponenten im offenen System bieten. Dabei sind alle Lebenszyklusanteile, die sich aus den in den vorangegangenen Unterabschnitten besprochenen Konzepten wie einfacher Konfiguration, Erweiterungs- oder Dienstbeziehung und Kapselung ergeben, zu berücksichtigen.

### 4.1.6 Mehrere Verwaltungsebenen

In einem offenen System kann es keinen einzigen, für alle Systemkomponenten gleichermaßen gültigen Lebenszyklus geben, der sämtliche technischen und fachlichen Aspekte der Integration abdeckt. Dies ist allein schon dem Umstand geschuldet, dass jede hinzugefügte Systemkomponente weitere Integrationsmöglichkeiten für zuvor unbekannte Arten von Systemkomponenten schaffen kann. Daher ist die im vorigen Unterabschnitt erhobene Forderung nach einer Modellierungsmöglichkeit für den ablauforientierten Anteil einer Schnittstelle so zu verstehen, dass Lebenszyklen für jede im Systementwurf identifizierte Schnittstelle entsprechend individuell definiert werden können müssen.

Zur Identifikation von Schnittstellen bieten sich genügend Möglichkeiten entlang der bisher formulierten Gedanken zur Modellierung des Plugin-Konzepts. Schnittstellen können anhand fachlicher, technischer oder aus beiden Aspekten kombinierter Überlegungen voneinander abgegrenzt werden. Zudem ist eine Orientierung an verschiedenen Abstraktionsebenen der Systemarchitektur sinnvoll – die (technische) Schnittstelle zum Hinzufügen und Entfernen von Systemkomponenten eines offenen Systems liegt auf einer anderen Ebene als die (eher fachlichen) Dienst- oder Erweiterungsspezifikationen des Systems. Insbesondere, wenn intendiert erweiterbare Systemkomponenten (fachlich) unabhängig vom äußeren System eine eigene Schnittstelle zum Integrieren

von Erweiterungen enthalten, ergibt sich notwendigerweise eine Trennung der Schnittstellen nach Architekturebenen.

In „reinen“ Plugin-Systemen wie Eclipse oder NetBeans wird das Problem so gelöst, dass lediglich ein globaler, technisch inspirierter Lebenszyklus definiert wird, der für alle Systemkomponenten einheitlich gilt. Dieser Lebenszyklus berücksichtigt nur solche Zustände und Zustandsübergänge, die von der zentralen Plugin- bzw. Modulverwaltung zum Hinzufügen, Konfigurieren und Entfernen von Systemkomponenten benötigt werden. Lebenszyklen auf dieser Ebene werden auch in manchen Komponentenmodellen definiert wie z. B. in Enterprise Java Beans [vgl. JBC+07, S. 641ff.].

Die Definition der zulässigen Zustände und Operationen der Erweiterungs- oder Dienstspezifikationen überlassen diese Systeme den Entwicklern der individuellen Module. Häufig führt dies zu nur unzureichend klar definierten Lebenszyklen der konkreten Beziehungen, insbesondere wenn die jeweiligen Entwickler lediglich Methoden, Prozeduren oder Funktionen deklarieren und den ablauforientierten Anteil der Schnittstelle vernachlässigen. Unterstützt wird dieser Missstand dadurch, dass in nicht dynamisch rekonfigurierbaren Systemen für Module mit während der Laufzeit unveränderlichen, also quasi ewig andauernden Beziehungen kein expliziter Beziehungsabbau nötig zu sein scheint. Am Beispiel Eclipse zeigt sich, dass solche Annahmen auch mehrere Jahre nach der Umstellung auf ein prinzipiell dynamisch rekonfigurierbares System noch dazu führen, dass die dynamischen Eigenschaften vielen Plugins Schwierigkeiten bereiten (vgl. Unterabschnitt 3.2.1.4).

Eine einfache Lösung des Problems ist, in der Softwarearchitektur des offenen Systems klar die Notwendigkeit mehrerer Ebenen von Lebenszyklen festzuschreiben und möglichst viele dieser Ebenen fest vorzugeben. Soweit möglich, sollten auch die Lebenszyklen einzelner Ebenen zentral definiert werden. So geschieht es beispielsweise in der OSGi-Architektur mit der Trennung von Bundle- und Service-Lebenszyklen (siehe Unterabschnitt 3.2.1.6), welche beide im Rahmenwerk fest definiert sind. Aus der Einführung der Architekturebenen ergibt sich meist eine weitere Vorgabe – jede Ebene benötigt eine eigene Verwaltung, so dass sich die Definition mehrerer separater Verwaltungsinstanzen empfiehlt. Die Verwaltungen dürfen sich nicht gegenseitig beeinträchtigen, müssen sich aber untereinander koordinieren können, falls zwischen den definierten Lebenszyklen der einzelnen Ebenen ein Zusammenhang besteht. In der OSGi-Architektur kann beispielsweise ein Service nur angeboten werden, solange sich das den Service bereitstellende Bundle im Zustand *aktiv* befindet.

Das zu entwickelnde Plugin-Konzept muss diese Mehrfachheit an Verwaltungen und Lebenszyklusspezifikationen erlauben, klar zwischen den Ebenen unterscheiden und Anknüpfungspunkte zur Koordination definieren können.

### 4.1.7 Nebenläufigkeit und Threadsicherheit

Nur in sehr seltenen Fällen arbeiten heutige Softwaresysteme noch rein reaktiv und sequentiell. Selbst Anwendungen, die von nur einem Anwender bedient werden, beinhalten in der Regel mehrere Kontrollflüsse (*Threads*), um z. B. die grafische Benutzungsoberfläche trotz durchzuführender, länger andauernder Rechenoperationen nicht einzufrieren. Der aktuelle Trend, auch auf Arbeitsplatzrechnern oder sogar in tragbaren Geräten Prozessoren mit mehreren echt parallel arbeitenden Kernen zu verbauen,

dürfte das Vorkommen von Softwaresystemen mit mehreren Threads weiter fördern. Aus dem Vorhandensein von Nebenläufigkeit in einem Softwaresystem ergeben sich allerdings erhöhte Anforderungen an Systementwurf und Implementierung, um nicht die Daten- oder Ablaufkonsistenz zu verletzen.

Ein aktuelles Beispiel für die Umstellung von ursprünglich sequentiell implementierten Anwendungen auf mehrere Threads sind die Diskussionen um die im Jahr 2008 vorgestellten neuen Generationen von Webbrowsern verschiedener Hersteller. Die Hersteller betonen, dass die Unterteilung der laufenden Webbrowser-Anwendung in mehrere Prozesse sinnvoll ist [siehe M+08, Zei08], um beim gleichzeitigen Darstellen verschiedener voneinander unabhängiger Webseiten unerwünschte Wechselwirkungen zu vermeiden, die zu Beeinträchtigungen der Performanz, der Vertraulichkeit von Daten oder der Anwendungsstabilität führen können. In [M+08, S. 29ff.] erläutert das „Google Chrome“-Team, dass die Webbrowser-Plugins sich nicht so einfach in die neue Mehrprozess-Architektur integrieren lassen. Dieses Problem ist vermutlich der auf genau einen Browser-Thread ausgelegten NPAPI anzulasten (vgl. Unterabschnitt 3.1.1.3).

In Unterabschnitt 4.1.3 wurden mit der Dienstbeziehung und der intendierten Erweiterbarkeit zwei alternative Beziehungsarten zur Organisation der Komponenten eines offenen Systems vorgestellt. Beide begünstigen das Vorhandensein mehrerer Threads im System. Die $m : n$-Beziehung zwischen Dienstanbietern und -nutzern kann in der Dienstspezifikation keine Beschränkung der Nebenläufigkeit zwischen unabhängigen Nutzern desselben Anbieters vorschreiben, da die Nutzer keine direkte Beziehung zueinander haben. Es ist Sache der Dienstanbieter, während der Beziehungsaufnahme sicherzustellen, dass ihre individuell angebotene Dienstleistung nicht mehrfach zeitgleich genutzt werden kann, sofern eine solche Einschränkung nötig erscheint.

Beim Entwurf eines intendiert erweiterbaren Softwaresystems könnte zwar fest vorgegeben werden, dass das System nur über einen einzigen Kontrollfluss verfügen soll. Diese Vorgabe ist aber weder sinnvoll noch praktikabel durchsetzbar. Erstens ergibt sich aus heutigen Anforderungen an die Bedienbarkeit von Systemen, wie bereits erwähnt, ein Bedarf an mehreren Threads. Zweitens schränkt die Vorgabe, dass keine Erweiterung einen eigenen Kontrollfluss beinhalten darf, die Flexibilität zukünftiger Erweiterungsmöglichkeiten unnötigerweise ein. Drittens ergibt sich aus komplexen Lebenszyklusdefinitionen auf mehreren Ebenen (siehe Unterabschnitte 4.1.5 und 4.1.6) die Situation, dass sich eine Systemkomponente gleichzeitig in Zuständen verschiedener Lebenszyklen befindet, welche möglicherweise nicht voneinander unabhängig sind. In diesen Lebenszyklen können nebenläufige Übergänge möglich sein, deren vollständige Sequenzialisierung sehr aufwändig wäre. Und viertens schützt auch der Verzicht auf mehrere Threads nicht vor einer möglicherweise vom kausalen Zusammenhang abweichenden Reihenfolge von Ereignissen bzw. Methodenaufrufen. Gerade der für intendiert erweiterbare Systeme typische Einsatz von Callback-Mechanismen und Multicast-Kommunikation begünstigt Abweichungen in dieser Hinsicht [vgl. Szy02, Abschnitte 5.4–5.8, 10.6 und 22.1.1].

Es sollte daher in Plugin-Systemen davon ausgegangen werden, dass eine unbekannte Anzahl von Kontrollflüssen im System existiert und dass besondere Vorsicht zur Erhaltung von Daten- und Ablaufkonsistenz nötig ist.

Als Lösungsstrategien erwähnt Szyperski transaktionale Programmierung oder die zentral koordinierte Taktung und Ordnung von Nachrichten oder atomaren Aktionen,

vertieft jedoch keinen dieser Ansätze weitergehend [vgl. Szy02, Abschnitte 22.1.2–22.1.3, S. 458ff.]. Die englischsprachige Wikipedia nennt unter dem Stichwort Threadsicherheit (*thread safety*) vier unterschiedliche Lösungsansätze, namentlich die Sicherstellung von Reentranz oder wechselseitigem Ausschluss sowie die Verwendung von Thread-spezifischem Speicher oder atomaren Operationen [siehe Wik08]. Die konkrete Wahl einer oder mehrerer dieser Techniken muss den Entwicklern eines konkreten Softwaresystems überlassen bleiben, da sich jede Technik je nach Situation mal mehr oder weniger eignet.

Das zu entwickelnde Plugin-Konzept kann sicherlich keine allgemeingültige, einheitliche Lösung zur Vermeidung von inkonsistenten Zuständen liefern, die sich aus nebenläufigen oder asynchronen Ereignissen im Gesamtsystem oder einzelnen Systemkomponenten ergeben können. Es ist aber ein Ausdruckmittel nötig, welches Software-Entwickler in die Lage versetzt, diese Probleme auf der Basis intuitiv verständlicher, aber dennoch mächtiger und präzise definierter Modellierungsprimitive leichter zu bewältigen. Nur mit einer geeigneten Modellierungssprache können die Schnittstellen aller Systemkomponenten einfach und sicher auf nebenläufige Kommunikation oder deren Ausschluss hin ausgelegt werden. Ferner darf das Plugin-Konzept, sofern Schnittstellen zur Verwaltung der Beziehungen zwischen Systemkomponenten festgelegt oder implementiert werden, nicht selber zur Quelle von Inkonsistenzen werden.

### 4.1.8 Visuelle Modellierung mit präziser Semantik

Der Versuch, die Modellierung eines komplexen Konzeptes wie dem der Plugin-Systeme ausschließlich mit Worten zu beschreiben, wird zwangsläufig entweder unpräzise oder sehr lang und umständlich werden. Die Vielfalt der zu berücksichtigen Aspekte, die in den vorangegangenen Anforderungen und auch in den in Kapitel 3 untersuchten existierenden Plugin-Systemen deutlich geworden sind, erlaubt keine knappe und lückenlose textuelle Beschreibung. Eine geeignete grafische Aufbereitung von Konzepten hilft im Allgemeinen beim Verständnis der Grundprinzipien und auch komplexerer Zusammenhänge. Die einheitliche grafische Darstellung mehrerer Varianten eines Konzepts erlaubt zudem einen einfachen optischen Vergleich und erleichtert so das Erkennen von Unterschieden.

Viele existierende Plugin-Systeme definieren Plugins in der Regel nur informell: Schnittstellen werden zwar in der Semantik einer (objektorientierten) Programmiersprache spezifiziert, das einzuhaltende Protokoll sowie der Lebenszyklus werden jedoch nur als Konventionen ohne vollständige formale Beschreibung an Plugin-Entwickler übermittelt. Eine Ausnahme unter den in Kapitel 3 betrachteten Systemen stellen lediglich die als endlicher Automat definierten Lebenszyklen des OSGi-Rahmenwerkes und die atomar konfigurierten Plugins in MagicBeans dar. Eine Ursache für diese Schwäche im Bereich der Verhaltensbeschreibung dürfte in der mangelnden Eignung der verwendeten Programmiersprachen für die Spezifikation zustands-, daten- oder ablauforientierter Zusammenhänge liegen. Die Verwendung grafischer Formalismen mit klarer Semantik kann dem Entwickler einen Überblick über die möglichen Zustände und Prozesse oder den Fluss von Informationen im System verschaffen.

Bei der Modellierung des Plugin-Konzepts sollte daher eine visuelle Modellierungssprache unterstützend herangezogen werden. Hilfreich ist die Sprache aber nur, wenn

sie die in der Modellierung benötigten Aspekte eines Systems einfach und präzise darstellen kann. Aus den Anforderungen der vorangegangenen Unterabschnitte, welche das zu modellierende Konzept charakterisieren, ergeben sich auch die Anforderungen an die visuelle Modellierungssprache:

- Es müssen *dynamische Änderungen* am System darstellbar sein, so ist z. B. ein fest definierter Vorrat an Systemkomponenten mit statisch fixierten Verbindungen für ein Modell eines offenen Systems nicht akzeptabel.
- Es muss visualisierbar sein, an welchen Stellen im System die *Konfiguration* stattfindet und wie das System oder seine Komponenten diese Aufgabe (ohne Eingriff des Anwenders) übernehmen können.
- Es muss eine einfache Darstellung für die bei Plugin-Systemen relevante Beziehungsart der *intendierten Erweiterbarkeit* geben. Andere Beziehungsarten wie z. B. die Dienstbeziehung sollten ebenfalls grafisch modellierbar sein.
- Systemkomponenten müssen gegen beliebige Zugriffe anderer *gekapselt* werden können, so dass ihre Grenzen und *Schnittstellen* in der grafischen Darstellung eindeutig erkennbar sind.
- Der vorige Punkt gilt nicht nur für die Menge der extern verfügbaren und genutzten Operationen, sondern auch für den ablauforientierten Anteil der Schnittstellenspezifikationen, insbesondere die *Lebenszyklen* der Systemkomponenten und ihrer Beziehungen.
- Die Modellierungssprache muss in der Lage sein, die verschiedenen *Verwaltungsebenen* klar voneinander getrennt darzustellen und dennoch Verbindungen zwischen diesen aufzuzeigen.
- Die Sprache muss *Nebenläufigkeit* im System ermöglichen und einfache grafische Mittel zur Behandlung von Problemen der Konsistenz bieten.
- Die gewählte Modellierungssprache muss eine *präzise Semantik* besitzen, die z. B. formal oder operational definiert sein kann.

In Abschnitt 2.3 wurden bereits Architekturbeschreibungssprachen (ADLs) als visuelle Modellierungsmöglichkeit für Softwarearchitekturen vorgestellt. Zudem wurde in Abschnitt 2.1 die Modellierung mit Petrinetzen eingeführt. Die Wahl der *Netze-in-Netzen* als Modellierungssprache zur Unterstützung der Entwicklung des Plugin-Konzepts wird in Unterabschnitt 4.2.1 begründet.

### 4.1.9 Definitionen

Die in den vorangegangenen acht Unterabschnitten zusammengetragenen Anforderungen an ein Plugin-Konzept haben sich aus der Untersuchung existierender Plugin-Systeme in Kapitel 3 ergeben. Dabei sind allerdings die zentralen Begriffe *Plugin* und *Plugin-System* bisher noch nicht streng definiert, sondern in einer umgangssprachlichen Allgemeinheit verwendet worden. Diese Definition kann nun, nachdem im Rahmen der Anforderungsermittlung notwendige grundlegende Begriffe definiert wurden,

nachgeholt werden. Sie bezieht sich nur auf die in den ersten vier Unterabschnitten dieses Abschnitts zusammengetragenen Anforderungen, da diese den unmittelbaren Nutzen eines Plugin-Systems beschreiben.

**Definition 12 (Plugin-System)** *Ein Plugin-System ist ein* offenes *(siehe Definition 4),* einfach konfigurierbares *(siehe Unterabschnitt 4.1.2) und* intendiert erweiterbares *(siehe Definition 5) System.*

**Definition 13 (Plugin)** *Ein Plugin ist ein* Modul *(siehe Unterabschnitt 4.1.4), welches zu einem Plugin-System während einer Einsatzphase hinzugefügt und wieder daraus entfernt werden kann und zu mindestens einer im System definierten Erweiterungsspezifikation passend als Erweiterung implementiert ist.*

Diese Definitionen beschränken sich zwar auf ein absolutes Minimum von Anforderungen an die Softwarearchitektur des Systems, haben aber eine Reihe technischer Konsequenzen zur Folge. Um die Anforderungen zu erfüllen, müssen notwendige Bestandteile eines Plugin-Systems das Folgende umfassen:

- eine einfach verwendbare Schnittstelle zum *Hinzufügen* und *Entfernen* von Plugins während einer Einsatzphase,
- eine oder mehrere Erweiterungsspezifikationen (siehe Definition 7) bzw. die Möglichkeit zur Definition derselben durch Systemkomponenten (z. B. Plugins), so dass zur Erweiterungsspezifikation passende Plugins einfach integriert werden können, und
- je ein definierter Lebenszyklus (siehe Definition 11) per Schnittstelle bzw. Erweiterungsspezifikation, wobei die Lebenszyklen untereinander gekoppelt sein dürfen, so dass sich für jedes Plugin ein kombinierter Lebenszyklus bestimmen lässt.

Ein Plugin-System kann, muss aber nicht, *dynamisch* rekonfigurierbar sein.

Die weiteren in diesem Abschnitt aufgestellten Anforderungen beschreiben keine unverzichtbar notwendigen Eigenschaften eines Plugin-Systems. Stattdessen handelt es sich dabei um Anforderungen, welche in der konzeptionellen Modellierung unterstützt werden müssen. Entweder sind die Eigenschaften nötig, da sie in Plugin-Systemen auftreten *können*, oder sie sind nötig, um ein klares und praxistaugliches konzeptionelles Modell zu erhalten.

#### 4.1.9.1 Diskussion

Dass die Definition von Plugins auf dem *Modul*-Begriff (siehe Unterabschnitt 4.1.4) aufbaut, ergibt sich aus der Tatsache, dass Plugins – ebenso wie Module – separat von anderen Systemkomponenten und dem System selber, übersetzt werden und gekapselt sind. Nach den Vorarbeiten in Kapitel 2 stellt sich allerdings die Frage, ob nicht auch die dort vorgestellten zentralen Konzepte anderer Softwareentwicklungsparadigmen, namentlich das *Objekt* (siehe Unterabschnitt 2.1.2.1), die *Komponente* (siehe Unterabschnitt 2.2.1) und der *Agent* (siehe Unterabschnitt 2.5.1.1), statt des Moduls als

Basis hätten dienen können. Die Diskussion dieser Konzepte muss neben den Eigenschaften der betrachteten Systemkomponente auch die Eigenschaften der Umgebung bzw. Infrastruktur und ihrer Modellierung berücksichtigen. Insbesondere das Konzept des Agenten ist ohne seine Einbettung im Konzept des Multiagentensystems unvollständig.

**Kleinster gemeinsamer Nenner** Das Modulkonzept kann auf jeden Fall als kleinster gemeinsamer Nenner aller soeben angesprochenen Begriffe verstanden werden. In der obigen Plugin-Definition wird direkt auf den Modulbegriff Bezug genommen. Das Kapselungsprinzip der Objektorientierung schließt die Definition einer Modul-Grenze mit öffentlicher Schnittstelle ein. Szyperski definiert in seiner detaillierten Komponentendefinition [Szy02, S. 420] Komponenten direkt über einen Modulbegriff (vgl. Diskussion in Unterabschnitt 4.1.4). Die weit verbreitete Agentendefinition von Jennings und Wooldridge (siehe Unterabschnitt 2.5.1.1) umfasst ebenfalls die Aspekte der Kapselung und der klar definierten Schnittstellen. Somit sind auch in diesem Fall alle Anforderungen des Modulbegriffes erfüllt.

**Geschäftlicher Aspekt** Den Definitionen 12 und 13 fehlt der geschäftliche Aspekt, wie ihn viele Definitionen des Komponentenbegriffs (vgl. Unterabschnitt 2.2.1) mit Aussagen über „unabhängige Auslieferbarkeit“ oder „Komposition durch Dritte“ streifen. Aus der softwarearchitektonischen Anforderung, dass Plugins die Rolle einer Erweiterung in einem intendiert erweiterbaren System einnehmen sollen, ergibt sich aber auf natürliche Weise, dass Plugins von einem anderen Hersteller als dem des intendiert erweiterbaren Systems stammen können („unabhängige Auslieferbarkeit“). Voraussetzung hierfür ist lediglich die Veröffentlichung der Erweiterungsspezifikation(en) des erweiterbaren Systems. Der „Komposition durch Dritte“ entspricht beim Plugin-Begriff die Anforderung der einfachen Integrierbarkeit. Hier ist allerdings der feine Unterschied zu beachten, dass bei Plugins Komposition nur in Form der Integration eines Plugins in ein Gesamtsystem auftritt, während aus Komponenten eigenständige Komposita gebildet und diese wiederum als Komponenten separat gehandelt werden können (vgl. Diskussion zu Tabelle 2.1). Dieser Unterschied erklärt auch das bereits in Unterabschnitt 2.2.4 angesprochene Aussparen von Anwendern aus der Sichtweise der CBSE – schließlich werden Komponenten im Rahmen der Komposition durch Systemmonteure statt durch Endanwender „verwendet“.

Die in Unterabschnitt 2.5.1.1 vorgestellten Agentendefinitionen sind ausschließlich technischer Natur und enthalten daher wie die hier gegebene Plugin-Definition keinen geschäftlichen Aspekt. In vielen Multiagentensystemen (z. B. in Schwarmsimulationen) spielt der einzelne Agent eine untergeordnete Rolle und wird nicht individuell an andere Entwickler oder Anwender ausgeliefert. Nichtsdestotrotz können auch Agenten einen geschäftlichen Nutzen für Anwender entfalten, wenn sie z. B. als mobile Einkaufsagenten zwischen Plattformen migrieren und Angebote vergleichen. In einem solchen Szenario müssen Agenten Anwendern zur Verfügung gestellt und von diesen dem System hinzugefügt werden können, wobei sich ein Agent bei jeder Migration erneut in ein Softwaresystem (die jeweilige Agentenplattform) integriert. Diese Integration geschieht ohne explizite Einwirkung von Anwendern und ohne Programmierung durch

Entwickler während der Laufzeit des Systems, fällt also in die Kategorie „einfache Konfiguration“. Eine Komposition von Agenten zu komplexen Agenten wird gelegentlich betrachtet, so zielt z. B. das Konzept des hierarchischen MULAN (siehe Abschnitt 2.5.2) genau auf solche Komposition ab. Natürlich lassen sich die geschäftlichen Aspekte einer Komponentendefinition auf Agenten übertragen, um den Handel mit Agenten zwischen Herstellern von MAS zu beschreiben. Komponierbarkeit und Handelbarkeit sind aber keine notwendige oder zentrale Eigenschaft üblicher Agentendefinitionen.

Die tatsächliche Entwicklung von Plugins durch Drittanbieter ist (ebenso wie bei Komponenten und Agenten) keine notwendige Anforderung an Plugin-Systeme – beispielsweise werden innerhalb von Eclipse (siehe Abschnitt 3.2) alle Kern-Plugins, welche die grundlegenden Schichten der Plattform ausmachen, zentral von der Eclipse-Entwicklergemeinde erstellt und ausgeliefert. Diese Plugins implementieren Erweiterungspunkte (*Extension Points*) darunterliegender Architekturschichten, sind also Plugins gemäß Definition 13, obwohl sie nicht von einem Drittanbieter stammen. Ein ähnliches Vorgehen lässt sich auch bei der Auslieferung zentraler XPCOM-Komponenten im Mozilla-Rahmenwerk und zentraler Module der NetBeans-Plattform beobachten.

**Inhaltliche Abgeschlossenheit** Sowohl in der CBSE als auch bei der Charakterisierung von Plugins wird von verschiedenen Autoren der Aspekt der Granularität der betrachteten Systemkomponente bzw. des inhaltlichen Zusammenhangs der in einer Komponente oder einem Plugin enthalten Funktionalität angesprochen. Griffel macht diesen Aspekt sogar zur zentralen Grundlage seiner Definition des Komponentenbegriffs [siehe Gri98, S. 31]. Andererseits ignorieren z. B. Councill und Heineman in [CH01, S. 7] den Aspekt vollständig und erwähnen nicht einmal mehr die Anforderung der Eigenständigkeit oder In-sich-Abgeschlossenheit, wie sie z. B. bei Sametinger auftritt [siehe Sam97, S. 68, „*self-contained*“, siehe auch Unterabschnitt 2.2.1]. Wolfinger et al. heben in [WDPM06, S. 288, siehe auch Unterabschnitt 2.2.5] die Ähnlichkeit von Plugins mit kleinen Anwendungen hervor, während im Eclipse-Whitepaper [RW06, S. 4, siehe auch Unterabschnitt 3.2.1.1] der Granularitätsaspekt wiederum als alleiniges Kriterium zur konzeptionellen Definition von Plugins herangezogen wird.

In Definition 13 kommt dieser Aspekt nicht unmittelbar vor, eine entsprechende (etwas schwächere) Eigenschaft ergibt sich aber aus der Anforderung, die Integration auf Basis der intendierten Erweiterbarkeit vorzunehmen. Die Konformität zu einer Erweiterungsspezifikation erzwingt im fachlich von der Spezifikation vorgegebenen Rahmen einen inhaltlichen Zusammenhang. Der häufig in komplexen Plugin-Systemen wie z. B. Eclipse auftretende Fall, dass ein Plugin (aufgrund eines *gewünschten* inneren Zusammenhangs) als Erweiterung zu *mehreren* Erweiterungsspezifikationen passend implementiert wird, wird indirekt über die Anforderung der einfachen Konfigurierbarkeit abgedeckt. Sind Plugins zu kleinteilig entworfen, so führt dies beim Anwender trotz einfacher Integration jedes einzelnen Plugins zu einem hohen Konfigurationsaufwand, bis das gewünschte Funktionalitätsniveau erreicht ist.[5] Falls ein Plugin zu viel oder inhaltlich nicht zusammenhängende Funktionalität enthält, entsteht beim Anwender

[5] Szyperski bringt die Granularitätsproblematik im Kontext der Wiederverwendbarkeit von Komponenten auf den Punkt: „*Maximizing reuse minimizes use*“ [Szy02, Abschnitt 4.1.8, S. 45]. Im Allgemeinen erhöht der kleinteilige Entwurf die Wiederverwendbarkeit der Komponenten, da jede

womöglich ein erhöhter Aufwand, bis er diese Funktionalität beherrschen bzw. bändigen kann.

Zum Aspekt der In-sich-Abgeschlossenheit ist noch ein wesentlicher Unterschied zwischen Plugins und Komponenten anzumerken: Im Umfeld der CBSE empfehlen Autoren im Allgemeinen, Komponenten so in sich abgeschlossen zu entwickeln, dass sie möglichst wenig Abhängigkeiten zum Kontext aufweisen (der schließlich aus anderen Komponenten besteht). Dies gilt zwar im Prinzip auch für Plugins, wie z. B. die entsprechende Empfehlung der Mozilla-Entwickler belegt [vgl. MDC07e, siehe auch Unterabschnitt 3.1.2.3]. Allerdings geht ein Plugin notwendigerweise mit jeder von ihm implementierten Erweiterungsspezifikation eine Abhängigkeit zum Kontext ein, der vom intendiert erweiterbaren System vorgegeben wird (vgl. Unterabschnitt 4.1.3.1). Somit ist ein Plugin gewolltermaßen von einem relativ umfangreichen Kontext abhängig.

Auf den ersten Blick ähnlich gelagert ist der Begriff der Komponenteninfrastruktur (siehe Unterabschnitt 2.2.1), welcher eine Komposition aus Komponenten beschreibt, die das Grundgerüst einer Anwendung bilden. Im Unterschied zu einem intendiert erweiterbaren Plugin-System wird bei einer Komponenteninfrastruktur aber nicht davon ausgegangen, dass nur solche Komponenten hinzugefügt werden, die speziell auf diese Infrastruktur zugeschnitten, also von ihr abhängig sind. Bei der Montage der restlichen Anwendungsteile basierend auf einer Komponenteninfrastruktur können beliebige, also auch von dieser speziellen Infrastruktur unabhängige Komponenten, zu dem Gesamtsystem hinzugefügt werden.

Reine Agentendefinitionen, die auf der Mikro-Ebene angesiedelt sind (siehe Unterabschnitt 2.5.1.1), schweigen sich zum Aspekt der inhaltlichen Abgeschlossenheit einzelner Agenten aus. Im Kontext der Multiagentensysteme wird mitunter eine gegenläufige Formulierung aufgenommen, welche die Koordination aller beteiligten Agenten auf ein gemeinsames Systemziel hin empfiehlt. Erst mit der Betrachtung von Rollen und Organisationen in der Makro-Ebene (siehe Unterabschnitt 2.5.1.3) wird ein inhaltlicher Zusammenhang in den Systementwurf eingebracht, da nun ein Agent, der im MAS eine Rolle ausübt, mindestens all jene Interaktionen beherrschen, Rechte ausüben und Pflichten erfüllen können muss, die im Kontext dieser Rolle relevant sind.

**Technischer Aspekt** Reduziert man die in Unterabschnitt 2.2.1 gegebenen Definitionen auf ihren technischen Aspekt, verschwinden die Unterschiede zwischen Modul und Komponente so weit, dass der Komponentenbegriff sich als Grundlage für die Definition des Pluginbegriffs eignet. Aus einer rein technischen Sichtweise heraus ist eine Komponente ein *Modul* (siehe Unterabschnitt 4.1.4), dessen Schnittstellen zu einem *Komponentenmodell* konform spezifiziert sind, so dass es ohne Modifikationen komponiert bzw. integriert werden kann. Dieser Definition fehlt neben dem geschäftlichen Aspekt der unabhängigen Ausliefer- und Einsetzbarkeit auch der Aspekt der In-sich-Abgeschlossenheit. Letzterer ergibt sich allerdings aus der ebenfalls in Komponentendefinitionen regelmäßig geforderten klaren Kapselung, Spezifikation und Dokumentation. Diese Anforderungen sind im Verständnis des Autors bereits Teil des

einzelne weniger Abhängigkeiten aufweist, reduziert aber auch den unmittelbaren Nutzen jeder einzelnen Komponente.

Modulkonzepts und werden daher in dieser technischen Definition des Komponentenbegriffs nicht wiederholt.

Eine Reduktion der Agentenbegriffe, wie sie in Unterabschnitt 2.5.1.1 vorgestellt wurden, ist kaum nötig, da die genannten Definitionen bereits sehr technisch sind. Lediglich die Zweckbestimmtheit oder Zielorientiertheit der Agenten geht über den Rahmen technischer Anforderungen hinaus. Allerdings legen die Definitionen von Agenten und Multiagentensystemen eine starke Betonung auf „innere“ Details der Agenten wie z. B. ihr (unvollständiges) Wissen, ihre Fähigkeiten, ihre Adaptivität oder ihre Autonomie. In den Plugin- oder Komponentendefinitionen sind keine Vorgaben in Richtung solcher Aspekte enthalten. Entfernt man die inneren Aspekte aus der Agentendefinition, bleiben auf der Interaktions- oder Makro-Ebene neben modularen Eigenschaften wie Kapselung und Schnittstellen nur die Forderung nach Flexibilität der Systemorganisation übrig. Das Konzept der Rollen erlaubt ähnlich wie bei der Klassifizierung von Objekten eine Identifikation gemeinsamer Schnittstellen mehrerer Agenten, ergänzt um die Möglichkeit des dynamischen Rollenwechsels eines Agenten.

Im Vergleich zur Definition 13 (Plugin) fehlen im technischen Komponentenbegriff zwei Eigenschaften: die Festlegung auf die Erweiterungsrolle und die Anforderung der einfachen Integration. Eine Komponente muss lediglich ohne Modifikation integrierbar sein, aber der dabei anfallende Aufwand ist nicht Teil der Definition. Ebensowenig ist die der Integration einer Komponente zugrundeliegende Beziehungsart fest vorgegeben. Die Flexibilität der Komponenten in diesem Aspekt ermöglicht erst die explizite Modellierung von unterschiedlichen Verbindern im Rahmen der Architekturbeschreibungssprachen (vgl. Unterabschnitt 2.3.2). Die Schnittstellen einer Komponente sind allerdings ebenso wie die Schnittstellen eines Moduls oder die Zuordnung eines Objekts zu einer Klasse über die Lebenszeit der jeweiligen Systemkomponente unveränderlich.

Objekte und Agenten führen auf dem Modulbegriff aufbauend das Konzept der Instanzen ein, welches bei Modulen, Komponenten und Plugins nicht benötigt wird. Bei Objekten und Agenten können mehrere Instanzen basierend auf einem gemeinsamen Code der zugrunde liegenden Klasse bzw. Rolle erzeugt werden. Dennoch werden auch bei Modulen, Komponenten und Plugins Schnittstellen standardisiert – beispielsweise in Form von Dienst- oder Erweiterungsspezifikationen. Hier wird jedoch die Spezifikation direkt in der jeweiligen Systemkomponente implementiert, so dass kein gemeinsamer Code mehrerer Systemkomponenten existiert. In allen Konzepten ist zwar prinzipiell die Definition von Generalisierungs- oder Spezialisierungsbeziehungen auf Schnittstellenspezifikationen möglich, explizit im Entwurf eingesetzt werden sie aber in der Regel nur bei der objekt- oder agentenorientierten Softwareentwicklung, da hier auch die Implementierung auf der Basis gemeinsamen Codes möglich ist.

Das Konzept des Plugins schließt auf der technischen Ebene also alle Eigenschaften des Modulbegriffs und auch des Komponentenbegriffs ein. Im Vergleich zum Objektbegriff fehlt dem Plugin (ebenso wie der Komponente) das Konzept der Instantiierbarkeit mehrerer Exemplare einer Plugin-Klasse, da die mehrfache Integration derselben Plugin-Funktionalität in ein System keinen Gewinn bringt. Mit dem Agentenbegriff teilt das Konzept des dynamischen Plugins die Eigenschaft der flexiblen Systemorganisation. Die Möglichkeit des dynamischen Rollenwechsels ist allerdings nicht vorhanden, da ein Plugin während seiner Lebenszeit auf seine Rolle(n) als Erweiterung passend zu einer oder mehrerer Erweiterungsspezifikationen festgelegt ist.

**Integration** Der Rückgriff auf das Konzept des intendiert erweiterbaren Systems (siehe Definition 5) legt die für Plugins vorgesehene Integrationsweise in ein Softwaresystem eindeutig fest. Bei der Definition von Komponenten findet sich zu diesem Aspekt die allgemeinere Anforderung, dass Komponenten zu einem Komponentenmodell konform modelliert und implementiert werden. Das Komponentenmodell legt Interaktions- und Kompositionsstandards fest (siehe Unterabschnitt 2.2.1), welche somit die Integration von Komponenten in ein Softwaresystem beeinflussen. Die Agentendefinitionen wiederum beschränken sich auf die schwache „Festlegung" der Interaktion eines Agenten mit seiner Umgebung mittels Sensoren und Aktoren. Im Falle von Softwareagenten wird noch unspezifischer der Nachrichtenversand und -empfang als einzige Interaktionsmöglichkeit vorgegeben (siehe Unterabschnitt 2.5.1.1), womit der Agent dieselben Kommunikationsmöglichkeiten wie ein Objekt gemäß der „reinen Lehre" der Objektorientierung hat. Vorgaben zur Integration entstehen erst, wenn Kommunikationsstandards mit dem Ziel der Interoperabilität wie die KQML-Definitionen oder FIPA-Spezifikationen (siehe Unterabschnitt 2.5.1.2) nicht nur ein Nachrichtenformat, sondern mit dem Plattformkonzept auch eine Agenten- und Diensteverwaltung einführen.

Die Interaktions- und Kompositionsstandards eines Komponentenmodells könnten unter anderem auch die zur Integration vorgesehene Beziehungsart festlegen – tun dies aber selten. In der Regel wird nur die Spezifikation von nutzbarer und benötigter Funktionalität jeder Komponente vorgesehen, was am ehesten der in Unterabschnitt 4.1.3.1 diskutierten Dienstbeziehung entspricht. Dieses Vorgehen ist konsequent, da das Konzept einer nur im geeigneten Kontext (erweiterbares System) einsetzbaren Erweiterung der Idee der unabhängig (also ohne umfangreiche Voraussetzungen an den Kontext) einsetzbaren Komponente zuwiderläuft. Die Diensteverwaltung einer FIPA-konformen Agentenverwaltung legt ebenfalls eine Dienstbeziehung zur Integration von Agenten in ein Multiagentensystem nahe. Ein Dienst entspricht in diesem Fall einer Rollendefinition von Anbieter und Nutzer, so dass die Integration von Agenten in ein Multiagentensystem auch als das Einnehmen und Abgeben von bekannten Rollen betrachtet werden kann. Im Kontext der objektorientierten Softwareentwicklung existieren ähnliche Integrationsmöglichkeiten wie bei Agenten, allerdings ist aufgrund der statischen Zuordnung von Objekten zu Klassen und des nicht geforderten Verzeichnisdienstes die Flexibilität der Systemorganisation stark eingeschränkt.

Aus dem Unterschied zwischen Dienst- und Erweiterungsbeziehung ergeben sich weitere Charakteristika von Plugins, Komponenten und Agenten. Basiert die Integration auf der Dienstbeziehung, erfolgt die Integration durch Programmierung eines Dienstnutzers und dessen expliziter Kopplung an einen Anbieter zur Laufzeit. Eine Automatisierung der Integration ist nur möglich, wenn Dienstanbieter in einem Verzeichnis publiziert und von Dienstnutzern programmgesteuert ausgewählt werden – wie es bei Agenten, die autonome Entscheidungen treffen können, der Fall ist. Ist eine Komponente nicht autonom, so fehlt diese Möglichkeit und die Kopplung muss vom Entwickler/Monteur des kombinierten Systems manuell implementiert werden, falls nicht das Rahmenwerk wie bei MagicBeans (siehe Unterabschnitt 3.3.4) die Komponenten „zufällig" anhand passender Schnittstellenspezifikationen zusammenfügen soll. Basiert die Integration hingegen auf der Erweiterungsbeziehung, so wird die Kopp-

lung schon bei der Entwicklung der Erweiterung mit der Festlegung auf das passende intendiert erweiterbare System festgelegt.

Bei allen Integrationsvarianten sind klar spezifizierte und dokumentierte Schnittstellen der jeweiligen Systemkomponenten von großer Bedeutung, da sonst keine Kopplung möglich wäre. Aber auch hier ergeben sich Unterschiede aus der zugrunde gelegten Beziehungsart. Dienstbeziehungen können nur aufgebaut werden, wenn die Dienstspezifikation öffentlich verfügbar ist. Falls eine Komponente oder ein Agent einen neuen, zuvor unbekannten Dienst anbietet, muss die passende Dienstspezifikation präzise dokumentiert und veröffentlicht werden. Daraus ergibt sich die in manchen Definitionen des Komponentenbegriffs geforderte Schnittstellendokumentation [z. B. Sam97, S. 68, siehe auch Unterabschnitt 2.2.1]. Erweiterungsbeziehungen benötigen ebenfalls eine öffentliche Erweiterungsspezifikation, diese ist aber Teil des intendiert erweiterbaren Systems. Daraus ergibt sich, dass Plugins, falls sie ausschließlich als Erweiterungen passend zu bereits existenten Erweiterungsspezifikationen implementiert sind, keine zusätzliche Schnittstellendokumentation mehr bereitzustellen brauchen (gleiches gilt natürlich für Komponenten bzw. Agenten, die ausschließlich als Nutzer bereits spezifizierter Dienste agieren, solche eignen sich allerdings kaum noch zur Komposition neuer Funktionalität).

**Infrastruktur** Eine Gemeinsamkeit, die im Integrationsaspekt der Konzepte Modul, Komponente, Plugin und Agent notwendigerweise besteht, fällt aufgrund unterschiedlicher Namensgebung nicht sofort auf. Alle benötigen einen geeigneten Kontext, der die Integration in einer Einsatzphase erst möglich macht. Bei Modulen sorgt hierfür die Laufzeitumgebung aus Lader und Linker des Betriebssystems, bei Objekten die Laufzeitumgebung der objektorientierten Programmiersprache (z. B. die Java-VM), bei Komponenten die Komponentenmodellimplementierung, bei Plugins das Plugin-Management-System (PMS, der Begriff wird in Unterabschnitt 4.3.6 eingeführt), und bei Agenten die Agentenplattform inklusive Verwaltungsagenten.

Es bietet sich an, die im Bereich der CBSE recht ausgefeilte Unterscheidung der Begriffe Komponentenmodell und Komponentenmodellimplementierung auf die anderen Konzepte zu übertragen: Als „Komponentenmodell“ von Modulen und Objekten kann z. B. die hard- und softwarespezifische Binärschnittstelle (*Application Binary Interface, ABI*, vgl. Unterabschnitt 3.1.1.2) oder die Spezifikation des Java-Classloaders (vgl. Unterabschnitt 3.2.1.2) betrachtet werden, das „Komponentenmodell“ von Plugins wird durch das erweiterbare System vorgegeben und das „Komponentenmodell“ von Agenten wird z. B. durch die FIPA-Spezifikationen standardisiert. Der wesentliche Nutzen der Komponentenmodellimplementierungen ist die für rekonfigurierbare Systeme notwendige zeitliche Entkopplung des Übersetzungsvorgangs (in einer Entwicklungsphase) von der Systemintegration (in einer Einsatzphase bzw. zur Laufzeit). Diese Entkopplung wird von allen fünf diskutierten Konzepten benötigt, wenn auch die Integration auf unterschiedliche Weise stattfinden kann.

Alle Komponentenmodellimplementierungen definieren mit der grundlegenden Infrastruktur einen Laufzeit-Lebenszyklus (siehe Unterabschnitt 4.1.5) für die auf Basis des jeweiligen Komponentenmodells ausgeführten Komponenten. Der Lebenszyklus von Objekten ist durch Konstruktor und Destruktor (bzw. die automatische Destruk-

tion durch den Garbage Collector) gegeben. Objektorientierte Programmiersprachen mit dynamischem Klassenlader ergänzen hier noch einen Lebenszyklus für Klassendefinitionen, welcher durch das Laden und Entladen der Klasse bestimmt wird. Das Laden und Entladen beschreibt zugleich den Lebenszyklus eines Moduls vollständig, in der ursprünglichen Verwendung von Modulen deckt sich die Laufzeit aller Module zudem mit der des gesamten Softwaresystems. Gleiches gilt für Komponentenmodelle ohne dynamische Rekonfiguration zur Laufzeit. Der Lebenszyklus eines Agenten auf einer Agentenplattform ähnelt eher dem von Objekten, da er im Wesentlichen durch das Erzeugen und Terminieren des Agenten definiert wird. Die FIPA definiert einen etwas komplexeren Lebenszyklus, der unter anderem auch Zustandswechsel zur Migration von Agenten unterstützt. Die in den Plugin-Systemen in Kapitel 3 beobachteten Lebenszyklusdefinitionen kombinieren einige der zuvor genannten Eigenschaften, sie werden in Unterabschnitt 4.4.2.2 miteinander verglichen. Neben dem häufig aus dem Modul- oder Komponentenkonzept übernommenen Laden und Entladen ist bei Plugins vor allem das Integrieren und Entfernen des Plugins in bzw. aus dem erweiterbaren System relevant.

**Ergebnis** Ob zwischen den diskutierten Begriffen zur Charakterisierung von Systemkomponenten, namentlich Modul, Objekt, Komponente, Plugin und Agent, ein Unterschied besteht, und ob einer der Begriffe sich auf einem anderen aufbauend definieren lässt, hängt von den bei der Definition berücksichtigten Aspekten ab. In jedem Fall definiert der Modulbegriff als kleinster gemeinsamer Nenner grundlegende technische Eigenschaften, die allen diskutierten Begriffen gemein sind. Somit ist jede Komponente, jedes Objekt, jedes Plugin und jeder Agent eine spezielle Variante eines Moduls.

Beschränkt man sich auf technische Aspekte, so kann eine Hierarchie zwischen den Begriffen Modul, Komponente und Plugin aufgebaut werden: Der Modulbegriff beschränkt sich auf die Eigenschaften der Kapselung und der separaten Übersetzbarkeit, der Komponentenbegriff fügt die Komponier- bzw. Integrierbarkeit im Rahmen eines Komponentenmodells hinzu, und der Plugin-Begriff legt die Art und Weise der Integration auf eine Erweiterungbeziehung mit einfacher Konfiguration fest. Somit wäre jedes Plugin auch eine Komponente.

Diese Spezialisierungshierarchie wird allerdings bei einer detaillierteren Betrachtung des Integrationsaspekts schon in Frage gestellt, da die Forderung nach der Schnittstellendokumentation, welche oft in der Definition einer Komponente enthalten ist, beim Plugin-Konzept in den Verantwortungsbereich des intendiert erweiterbaren System verlagert wird. Eine Gemeinsamkeit besteht hier nur, falls die Erweiterungsspezifikationen ebenfalls von Systemkomponenten, die wiederum Plugins sind, bereitgestellt werden. In einem solchen System – als Beispiele lassen sich sämtliche „reinen“ Plugin-Systeme nennen, von den in Kapitel 3 untersuchten sind Eclipse, Mozilla und NetBeans die prominentesten – werden quasi „Plugin-Komponenten“ integriert. Eine Festlegung auf diese Begriffskombination würde allerdings die ursprünglichen, anwendungsspezifischen Plugin-Systeme (wie z. B. die NPAPI) aus der Betrachtung ausschließen.

Sobald man noch den geschäftlichen Aspekt oder den Aspekt der inhaltlichen Abgeschlossenheit hinzunimmt, treten offensichtliche Unterschiede zwischen dem Komponenten- und dem Pluginbegriff zu Tage. Plugins brauchen weder in anderen Kontexten

wiederverwendbar noch zwischen Softwareentwicklern geschäftlich handelbar zu sein, beides sind aber in vielen Komponentendefinitionen geforderte Eigenschaften. Während bei Komponenten Abhängigkeiten nach Möglichkeit vermieden werden sollen, sind sie bei Plugins inhärenter Bestandteil der Erweiterungsbeziehung.

Objekte und Agenten sind im Gegensatz zu Modulen, Komponenten und Plugins instantiierbar. Hauptzweck der letztgenannten drei Konzepte ist das reine Bereitstellen von Funktionalität – die Kapselung von Zuständen wie bei Objekten und Agenten ist hierzu nicht erforderlich.

Die inneren Eigenschaften des Agentenbegriffs liegen orthogonal zu den betrachteten Aspekten der anderen Begriffe, lassen sich aber auch mangels kollidierender Anforderungen jederzeit mit diesen zur Bildung neuer Konzepte zusammenführen. Gleiches gilt für die Flexibilität der Organisation, welche in Multiagentensystemen betont wird. Hier lassen sich die Vorgaben eines Komponentenmodells oder eines Plugin-Systems, die sich z. B. in der Festlegung von Dienst- oder Erweiterungsbeziehungen manifestieren, als eine konkrete, auf Flexibilität ausgelegte Organisationsform einordnen.

Ein Widerspruch, der die Definition von Plugin-Komponenten oder gar Plugin-Komponenten-Agenten verhindern würde, ergibt sich aus den eben genannten Unterschieden nicht. Ebenso ist eine Implementierung von Plugins auf Basis des objektorientierten Paradigmas nicht ausgeschlossen (im Gegenteil, sie ist in der Praxis sogar weit verbreitet). Somit ist der Weg frei, im Folgenden ein Plugin-Konzept zu modellieren, das Vorzüge der anderen Begriffe mit aufnimmt.

## 4.2 Modellierungstechnik

Die verwendete Modellierungssprache und -technik hat einen großen Einfluss auf das damit erstellte Modell. Dies gilt nicht nur im Hinblick auf fachlich nicht direkt relevante Aspekte wie die Größe, Übersichtlichkeit oder Klarheit des Modells, sondern auch im Hinblick auf Eigenschaften des modellierten Konzepts. Fehlende oder zusätzliche Ausdrucksmöglichkeiten einer Modellierungssprache können zu einer anderen Sichtweise und damit zu einem fachlich von der Sprache geprägten Modell führen. Daher kommt der Wahl einer geeigneten Modellierungssprache und der Charakteristik ihrer Verwendung eine große Bedeutung zu. Die Modellierungssprache sollte möglichst gut zum zu modellierenden Konzept passen.

Im Folgenden soll die in dieser Arbeit verwendete Modellierungssprache der Java-Referenznetze und das darauf abgestimmte Vorgehen bei der Modellierung diskutiert werden. In diesem Zusammenhang sind einige Eigenschaften der Sprache zu erwähnen, die sie von anderen, verbreiteten Modellierungs- oder Implementierungssprachen unterscheiden. Außerdem werden einige Modellierungsabstraktionen eingeführt. Diese Abstraktionen erhöhen die konzeptionelle Klarheit der Modellierung, indem sie Details ausblenden, die zwar bei der Implementierung eines konkreten Plugin-Systems anfallen, aber nicht konzeptprägend sind. Diese Abstraktionen werden zum Teil durch spezifische Ausdrucksmittel der Modellierungssprache unterstützt.

### 4.2.1 Wahl der Modellierungssprache

Die in Unterabschnitt 4.1.8 zusammengefassten Anforderungen an eine Modellierungssprache sind umfangreich und vielfältig: Sie benötigt eine grafische Repräsentierung mit präziser Semantik, muss sowohl Systemstruktur (Kapselung, Schnittstellen, Beziehungen) als auch Abläufe (Lebenszyklen) darstellen können, muss dabei dynamische Veränderungen an der Systemzusammensetzung zulassen und darf keine systeminhärente Nebenläufigkeit unterbinden. Idealerweise sollte zudem der Übergang vom Modell zu einer Implementierung ohne Bruch erfolgen können, d.h. ohne eine wiederholte Deklaration der bereits modellierten Strukturen und Abläufe in der Implementierungssprache zu erfordern. In den vorangegangenen Teilen dieser Arbeit wurden an verschiedenen Stellen Modellierungssprachen vorgestellt.

Die Architekturbeschreibungssprachen (ADL) aus Abschnitt 2.3 sind speziell für den Zweck der Modellierung von Softwarearchitekturen entwickelt worden und mit einer präzisen Semantik unterlegt. Allerdings hapert es in vielen Fällen an der Beherrschung dynamischer Änderungen in der Systemstruktur. Zudem wird zwar in einigen klassischen ADLs durchaus auf die Spezifikation von Verhaltensbeschreibungen für Komponentenschnittstellen oder Verbinder Wert gelegt, deren Spezifikation erfolgt aber – sofern überhaupt – nicht in visueller Form. Der Übergang von einer ADL-Spezifikation zu einer Systemimplementierung erfordert bei allen ADLs einen Wechsel zu einer Implementierungssprache. Es existieren allerdings in einigen Fällen Generatoren für Code-Skelette, um die bereits in der ADL entworfene Architektur nicht manuell nachbilden zu müssen.

Der prominenteste Vertreter der Modellierungssprachen mit umfangreichem Einsatz in der Praxis ist zweifelsohne die in Unterabschnitt 2.3.3 skizzierte Unified Modeling Language (UML). Da die UML viele verschiedene Diagrammarten für Strukturen und Abläufe umfasst, die auch in mehreren Sichten oder gar in einer Zeichnung kombinierbar sind, sollte sie sich eigentlich zur Modellierung aller Aspekte des Plugin-Konzepts eignen. Die Semantik der UML ist jedoch nur halbwegs formalisiert. Außerdem lassen sich dynamische Änderungen an der Systemstruktur gar nicht darstellen, und die Nebenläufigkeit von Teilen des Modells muss extra berücksichtigt werden, da viele ablauforientierte Diagrammarten den sequentiellen Ablauf als Standardfall ansehen. Ebenso wie bei den ADLs erfordert der Übergang zur Implementierung den Wechsel der Sprache, wenn auch vermehrt Code-Generatoren oder gar Roundtrip-Engineering-Werkzeuge existieren, welche Softwareentwicklern die manuelle Nachbildung und Pflege der Systemarchitektur in der Implementierungssprache ersparen.

In Abschnitt 2.1 wurde der Formalismus der Java-Referenznetze vorgestellt, der Eigenschaften von Petrinetzformalismen mit objektbasierten Konzepten vereint. Als Petrinetzformalismus verfügen die Java-Referenznetze über eine grafische Darstellung mit präziser Semantik und eignen sich besonders zur Modellierung von Abläufen und von nebenläufigen Zuständen und Aktionen. Das den Referenznetzen zugrunde liegende Paradigma der *Netze-in-Netzen* erlaubt aufgrund seiner Nähe zur objektorientierten Modellierung zudem die Spezifikation gekapselter Strukturen eines Softwaresystems (z. B. in einer Hierarchie), deren Zusammensetzung sich dynamisch verändern kann. Die Integration der umfangreichen Funktionalität der plattformunabhängigen Programmiersprache Java ermöglicht die Verwendung der Java-Referenznetze nicht

nur als Modellierungs-, sondern auch als Implementierungssprache, so dass kein Bruch beim Übergang vom Modell zur Implementierung auftritt.[6]

Die genannten Eigenschaften der Java-Referenznetze prädestinieren sie für den Einsatz als Modellierungssprache im Kontext dieser Arbeit. Im Vergleich zur Systemstrukturmodellierung mit ADLs oder den Struktur-Diagrammarten der UML und auch im Vergleich mit den Zustands-Diagrammarten der UML benötigt die Modellierung von Systemstruktur oder Zustandsübergängen mit Referenznetzen zwar gelegentlich einen höheren Aufwand in Bezug auf die Zahl der zu zeichnenden Elemente oder Anschriften. Dieser Nachteil wird aber durch eine insbesondere im Vergleich zur UML präziseren Semantik und durch die Möglichkeit der Zusammenfassung von Struktur- und Zustandsaspekten in einer Zeichnung wieder aufgewogen.

Bereits am Ende des Unterabschnitts 2.1.3 wurde angemerkt, dass für Referenznetze keine allgemeinen Analyseverfahren wie für viele andere Petrinetzformalismen existieren. Für die Modellierung im Rahmen dieser Arbeit stellt das zunächst keine Einschränkung dar, da im Interesse der intuitiven Nachvollziehbarkeit und Flexibilität des Modells ohnehin keine lückenlose Formalisierung des Plugin-Konzepts angestrebt wird. Eine formale Analyse der Modelle ist dennoch nicht ausgeschlossen, da grundsätzlich die Möglichkeit besteht, die Modelle im Hinblick auf Analyseeigenschaften einzuschränken.

### 4.2.2 Modellierung durch fortschreitende Verfeinerung

Da mit den Java-Referenznetzen eine Modellierungssprache zur Verfügung steht, die sich ebenso als Implementierungssprache eignet, kann der Übergang vom Modell zur Implementierung fließend erfolgen. Um vom zunächst abstrakten Modell zur konkreten Implementierung zu gelangen, werden Teile des Modells durch weitere Netzelemente verfeinert oder mit Implementierungsdetails z. B. in Form von Java-Anschriften an Transitionen angereichert. Dieses Vorgehen hat sich unter dem Schlüsselbegriff „Implementierung durch Spezifikation“ bereits bei diversen Arbeiten im Kontext des Arbeitsbereichs TGI bewährt, so auch bei der Studien- und Diplomarbeit des Autors (siehe [Duv01, Duv02]).

Im Kontext dieser Modellierungstechnik bietet es sich an, auch den Entwurf des abstrakten Modells bereits in mehreren Schritten vom einfachen Grundprinzip hin zu einem fortlaufend verfeinerten und komplexeren Modell zu gehen. Auf diese Weise können konzeptionelle Ideen, die im vollständigen Modell durch viele weitere Modellierungsdetails bereits verschleiert wären, separat erläutert und durchdacht werden. Im Rahmen dieser Arbeit wird die Technik der schrittweisen Verfeinerung vor allem im Abschnitt 4.3 eingesetzt.

[6] Beim Übergang von Spezifikationsmodellen zur Implementierung mit Java-Referenznetzen ist jedoch zu beachten, dass die vollständig formal definierte Semantik von P/T-Netzen oder gefärbten Petrinetzen durch eine operationale Semantik abgelöst wird. Sowohl die Programmiersprache Java als auch der von Kummer in seiner Dissertation [Kum02a] gegebene Formalismus der Referenznetze verfügen bisher lediglich über eine operationale Semantik.

### 4.2.3 Implizite Nebenläufigkeit

Wie in Unterabschnitt 4.1.7 bereits erläutert, existieren in modernen Softwaresystemen viele voneinander unabhängige Funktionen und Prozesse, die nebenläufig zueinander ausgeführt werden können. Eine unnötige Einschränkung dieser Nebenläufigkeit sollte vermieden werden, da moderne Rechensysteme in der Lage sind, solche Funktionen und Prozesse tatsächlich physisch parallel abzuarbeiten.

Die Wahl eines Petrinetzformalismus als Modellierungssprache bietet hierfür ideale Voraussetzungen: Eine Transition ohne Ein- und Ausgangsstellen (und ohne synchrone Kanäle) kann beliebig oft nebenläufig zu allen anderen Transitionen des Systems und sogar zu sich selbst schalten. Erst durch die Verbindung mit anderen Netzelementen wird die Nebenläufigkeit explizit eingeschränkt. Aber auch Transitionen mit Abhängigkeiten zu anderen Netzelementen können immer noch nebenläufig zu sich selbst schalten, falls ausreichend viele Marken zeitgleich zur Verfügung stehen.

Das Paradigma der *Netze-in-Netzen* bedient sich ebenfalls einer von Grund auf nebenläufigen Semantik. Die Zustände und Schaltvorgänge mehrerer Netzinstanzen sind grundsätzlich voneinander unabhängig, lediglich im Falle explizit modellierter synchroner Kanäle wird diese Unabhängigkeit aufgehoben. Anders als bei konventionellen Klassen bzw. deren Objekten in einer sequentiellen Programmiersprache, die nur bei einem Konstruktor- oder Methodenaufruf vorübergehend einen Kontrollfluss enthalten, können Netzinstanzen ab dem Moment ihrer Erzeugung nebenläufig zu allen anderen Netzinstanzen im System schalten.

Um diese Vorteile im Rahmen der Modellierung des Plugin-Konzepts so weit wie möglich ausnutzen zu können, sind bei der Modellierung einige Techniken empfehlenswert:

- Verschiedene Aspekte des Modells sollten vorwiegend unabhängig voneinander modelliert und nur im Falle tatsächlicher Abhängigkeiten miteinander verbunden werden. Diese Empfehlung gilt nicht nur bei der Modellierung getrennter Netze, sondern auch bei der Modellierung von Transitionen innerhalb eines Netzes. In letzterem Fall führt diese Technik häufig zu unzusammenhängenden Netzgraphen, die vielleicht beim untrainierten Betrachter den Eindruck entstehen lassen können, es handele sich um getrennte Netze. Falls tatsächlich mehrere Netze in einer Abbildung dargestellt werden, ist daher eine explizite Darstellung der Netzgrenzen erforderlich.

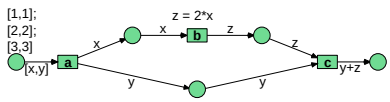
- Die inhärente Nebenläufigkeit von Transitionen zu sich selbst bei ausreichender Markierung kann, falls sie so oft wie möglich ausgenutzt wird, für sehr kompakte Graphen sorgen. Ähnlich der Reentranz-Problematik bei klassischen Prozeduren ist aber auch bei dieser Technik dafür Sorge zu tragen, dass die Daten (Marken) inhaltlich voneinander unabhängiger Schaltfolgen nicht zwischen zwei Schaltvorgängen untereinander vermischt werden (wie in Abbildung 4.2(a)).

  Zur Vermeidung dieses Problems innerhalb eines Netzes existieren zwei Techniken: Entweder werden Marken auf allen involvierten Stellen zu Tupeln mit einer den Zusammenhang identifizierenden Komponente erweitert (wie in Abbildung 4.2(b)), oder alle zusammengehörigen Daten werden in einer (komplexen) Marke in nur einer Stelle zwischengelagert (wie in Abbildung 4.2(c)), statt sie

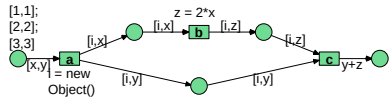
der Übersichtlichkeit halber auf mehrere Stellen aufzuteilen. Die letztere Variante eignet sich insbesondere im Falle eines sequentiellen Kontrollflusses, der in mehreren voneinander unabhängigen Instanzen auftritt - die komplexe Marke fungiert dann gleichzeitig als Zustandsindikator einer Instanz des Kontrollflusses. Alternativ bietet sich zur Vermeidung der Reentranz-Problematik der Ausweg an, jedem zusammengehörigen Datensatz eine eigene Netzinstanz zuzuordnen (wie in Abbildung 4.2(d)), so dass die einzelnen Daten ohne Vermischungsgefahr auf unterschiedliche Stellen verteilt werden können. Diese Variante ist am übersichtlichsten und schränkt die Nebenläufigkeit nicht ein, benötigt aber zusätzliche Netzinstanzen, welche im Modell zu berücksichtigen sind. Die Modellierungsvarianten – identifizierende Komponenten in Tupeln, Zusammenfassung in jeweils einer einzigen Marke oder Trennung in Netzinstanzen – haben jeweils eine eigene Charakteristik bezüglich des Systemverhaltens und der möglichen Nebenläufigkeit. Von diesen Varianten kann daher je nach Eigenschaften des zu modellierenden Systems eine passende gewählt werden.

- Falls eine Marke von einer Transition im Schaltvorgang wieder in dieselbe Stelle wie vorher zurückgelegt werden soll, ist zu prüfen, ob die Marke modifiziert werden soll oder nur als Nebenbedingung benötigt wird. Im letzteren Fall erlauben die im Referenznetzformalismus verfügbaren Testkanten die nebenläufige Konsultation der Nebenbedingung in mehreren voneinander unabhängigen Schaltvorgängen. Im ersteren Fall ist hingegen die nebenläufige Nutzung zu unterbinden, um Inkonsistenzen zu vermeiden – zu diesem Zweck sollte eine Reservierungskante als kombinierte Ein- und Ausgangskante die Marke für die Dauer des Schaltvorganges aus der Stelle abziehen, also exklusiv reservieren [vgl. KWD06b, Kapitel 3].

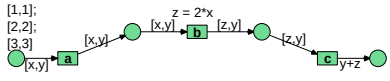
  Eine Ausnahme von dieser Regel stellen Marken dar, welche, dem Prinzip der *Netze-in-Netzen* folgend, wiederum Netzinstanzen repräsentieren. Die nebenläufige Veränderung des inneren Zustands der Netzinstanz (also: der Marke) ist in so einem Fall die Regel (sie kann jederzeit stattfinden). Da beim Einsatz synchroner



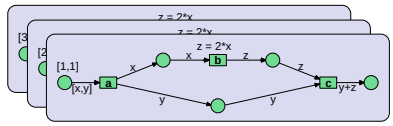

(a) Mischung unabhängiger Marken




(b) Separation mittels Identifier




(c) Bündelung zu einem Tupel




(d) Separation mittels Netzinstanzen

Abbildung 4.2: Vermeidung von Inkonsistenzen bei Reentranz bzw. Nebenläufigkeit

Kanäle die durch die Marke repräsentierte Netzinstanz selber den wechselseitigen Ausschluss nebenläufiger Zugriffe und damit die Entstehung von Inkonsistenzen kontrollieren kann, sind keine Reservierungskanten im äußeren Netz erforderlich.

### 4.2.4 Wert- oder Referenzsemantik

In der Modellierung sollen Systemkomponenten (also Plugins) durch Netze und das Beziehungsgeflecht zwischen den Systemkomponenten durch Beziehungen entsprechend des *Netze-in-Netzen*-Paradigmas repräsentiert werden. Wie in Unterabschnitt 2.1.2 bereits erläutert, stehen im Rahmen des *Netze-in-Netzen*-Paradigmas verschiedene Semantiken zur Interpretation der Netzmarken zur Verfügung. Der als Modellierungssprache gewählte Java-Referenznetzformalismus unterstützt namensgemäß die Referenzsemantik, welche der Semantik von Objektreferenzen in objektorientiert entwickelten Softwaresystemen entspricht. Insbesondere kann zwischen Netzinstanzen in Form der Netzreferenz-Marken ein beliebig komplexes Beziehungsgeflecht entstehen, welches weder asymmetrisch noch zyklenfrei sein muss. Die Referenzsemantik eignet sich daher gut zum Modellieren von Systemen, welche aus auf beliebige Weise untereinander verbundenen Systemkomponenten zusammengefügt sind.

Im Gegensatz dazu sieht die Wertsemantik für jede Netzmarke einen einzigen Platz im System vor. Falls Kopien einer Netzmarke erzeugt werden, so sind diese voneinander unabhängig und haben jeweils einen eigenen Zustand. Der Beziehungsgraph zwischen Systemkomponenten ist in einem unter Wertsemantik interpretierten Netzmodell grundsätzlich baumförmig, da jede Systemkomponente nur an einer einzigen Stelle in einer anderen Systemkomponente „enthalten“ sein kann. Diese Eigenschaft entspricht der intuitiven Sichtweise auf ein System, das aus hierarchisch zusammengefügten Systemkomponenten besteht.

Allerdings passt die Eigenschaft der Wertsemantik, Kopien von Netzmarken mit voneinander unabhängigen Zuständen erzeugen und solche Kopien wieder zu einer Netzmarke zusammenführen zu können, nicht auf die Modellierung von Plugin-Systemen. Wie bereits in Unterabschnitt 2.2.1 diskutiert, stellen Plugins ebenso wie Komponenten nicht-instantiierbare Elemente eines Softwaresystems dar. Ein Plugin soll Funktionalität in ein System integrieren. Es entsteht kein Vorteil, wenn mehrere unabhängige Kopien derselben Funktionalität im System vorhanden sind. Zudem ist bei Plugins nicht ersichtlich, wie das Aufteilen und Verschmelzen von Zuständen der Netzkopien im Modell sinnvoll interpretiert werden kann.

Um die Architektur eines aus mehreren Systemkomponenten zusammengesetzten Softwaresystems – also auch eines Plugin-Systems – anfangs abstrakt und leicht verständlich zu modellieren, ist zunächst eine streng hierarchische Organisation des Modells wünschenswert. Eine solche hierarchische Organisation wird durch Modellierung unter Wertsemantik erzwungen. In späteren Schritten muss aber eine Interpretation des Modells unter Referenzsemantik möglich sein, um komplexe Geflechte aus Erweiterungs- und Dienstbeziehungen darstellen zu können. Falls ein Plugin auf mehrere Weisen als Erweiterung oder Dienstanbieter anderer Systemkomponenten agiert, werden mehrere Referenzen auf ein- und dasselbe Plugin benötigt. Solche Referenzen ließen sich zwar auch unter Wertsemantik durch die Vergabe von Identifikatoren und eine explizit zu implementierende Kommunikationsschicht auf oberster Ebene des Sys-

tems realisieren, aber die Klarheit des Modells würde unter einer solchen Modellierung leiden.

Als Schnittmenge der beiden Netzsemantiken bietet sich in den ersten Modellierungsschritten eine Interpretation der Netzmarken an, welche weder von Referenzen noch von Kopien Gebrauch macht. So bilden die Modelle die Idee von physisch an genau einem Platz im Gesamtsystem positionierten Systemkomponenten unmittelbar ab. Diese Beschränkung in der Modellierung erhöht die Verständlichkeit und Nachvollziehbarkeit des Modells. Im Java-Referenznetzformalismus kann diese Beschränkung einfach eingehalten werden, wenn bei der Modellierung von Transitionen alle Variablen, welche Plugin-Netzmarken binden, stets in einfacher Multiplizität an maximal einer Ausgangskante einer Transition vorkommen.

Ab dem Moment, an dem mehrere Referenzen auf eine Netzmarke benötigt werden, können (aufgrund des Verzichts auf Netzkopien im Modell) dieselben Netzmuster verhaltensgleich unter Referenzsemantik interpretiert werden. Weiter verfeinernde Modellierungsschritte sind dann nicht mehr an die eben genannte Beschränkung gebunden und können so mehrere Netzreferenzen auf eine Netzinstanz erzeugen. Die für Menschen leichter nachvollziehbare hierarchische Systemarchitektur wird so ohne Änderung am Modell zu einem beliebigen Beziehungsgeflecht aufgelockert, welches sämtliche Freiheiten im Systementwurf bietet. Einzelne Ausschnitte aus diesem Beziehungsgeflecht (z. B. Paarbeziehungen zwischen Netzinstanzen) können aber weiterhin als hierarchische Beziehungen einfach nachvollzogen werden.

In der vorangegangenen Diskussion werden Netzmarken mit Netzinstanzen gleichgesetzt, da dies der Semantik von Java-Referenznetzen entspricht. Mehrere Netzinstanzen können – wie in Unterabschnitt 2.1.3 erläutert – auf einem gemeinsamen Netzmuster, also einer „Netzklasse“ basieren, verfügen aber über unabhängige Markierungen. Trotz der freiwilligen Beschränkung der Modellierung auf nur eine Referenz je Netzmarke besteht somit die Möglichkeit, mehrere Netzinstanzen zu einem Netzmuster im Modell zu erzeugen. Diese Netzinstanzen wären im Sinne der vorangegangenen Diskussion als verschiedene, voneinander unabhängige Netzmarken zu betrachten. Auf jede dieser Netzmarken dürfte somit in den frühen Modellierungsschritten genau eine Referenz im Gesamtsystem existieren. Diese Möglichkeit wird jedoch in der Modellierung nicht genutzt werden, da bei der Modellierung von Plugin-Systemen das Konzept mehrerer Instanzen eines Musters bzw. einer Klasse nicht benötigt wird. Eine strenge Unterscheidung der Begriffe Netz, Netzmuster, Netzinstanz und Netzmarke ist daher nicht nötig. Die Begriffe werden in der Modellierung nicht inhaltlich unterschieden werden, sondern allenfalls aufgrund technischer Aspekte der Petrinetzsimulation nebeneinander zum Einsatz kommen.

### 4.2.5 Schnittstellendefinition und Aufrufrichtung

Im Formalismus der Java-Referenznetze findet Kommunikation zwischen Netzinstanzen in Form synchroner Kanäle statt (siehe Unterabschnitt 2.1.3). Im Paradigma der *Netze-in-Netzen* entsprechen diese Kanäle vertikaler Kommunikation zwischen System- und Objektnetz, ergänzt um die Möglichkeit der Kommunikation eines Netzes mit sich selbst. Horizontale Kommunikation zwischen Objektnetzen, die innerhalb derselben Stelle liegen, ist im Referenznetzformalismus nicht unmittelbar möglich.

Ein Netz kann an jeder seiner Transitionen maximal einen Kanal-*Uplink* und beliebig viele -*Downlinks* deklarieren, wobei letztere die Angabe eines Kommunikationspartners in Form einer Netzreferenz benötigen, die z. B. aus den in einem Schaltvorgang gebundenen Marken bestimmt werden kann. In einer einfachen Analogie zur objektorientierten Programmierung entspräche ein Uplink einer Methodensignatur, während ein Downlink einem Methodenaufruf nahe käme, da letzterer ebenfalls eine Objektreferenz benötigt, die z. B. aus lokalen Variablen oder Instanzvariablen gewonnen werden kann. Gruppen aus im Netz deklarierten Kanal-Uplinks eignen sich daher zur Definition der Schnittstelle(n) des Netzes, so wie Gruppen von Methodensignaturen die Schnittstelle(n) einer Klasse bestimmen. Aufgrund der im Folgenden noch detaillierter besprochenen Bidirektionalität synchroner Kanäle eignen sich Gruppen von Downlinks ebenfalls zur Definition von Netzschnittstellen.

#### 4.2.5.1 Bidirektionalität

Die einleitend genannte Analogie zwischen synchronen Kanälen und Methodenaufrufen trifft jedoch nicht ganz, da Kanäle im Gegensatz zu Methoden eine bidirektionale, synchrone Kommunikation ermöglichen. Zwar haben beide Konzepte gemein, dass zwischen den beteiligten Objekten bzw. Netzinstanzen eine Asymmetrie besteht, da nur einer der Partner über eine Referenz auf das andere Objekt bzw. die andere Netzinstanz verfügen muss. Jedoch unterscheiden sie sich in der Richtung des Informations- und Kontrollflusses: Beim Methodenaufruf ist fest vorgegeben, dass beide Flüsse zu Beginn vom aufrufenden zum aufgerufenen Objekt hin und am Ende in Gegenrichtung wieder zurück fließen. Die am Kanal beteiligten Netzinstanzen sind hingegen während der gesamten (unteilbaren) Kommunikation gleichberechtigt am Informations- und Kontrollfluss beteiligt. Es können nicht nur beide Netzinstanzen zeitgleich Informationen bereitstellen, entgegennehmen oder zusammenführen, sondern es haben auch beide Instanzen Einfluss auf das Zustandekommen des Kanals. Jede Netzinstanz definiert eigenständig Bedingungen zur Aktivierung der Transitionen, welche jeweils mit Up- oder Downlink einen Kanalendpunkt bilden. Sind die Bedingungen an einem der Kanalendpunkte – egal, welchem – nicht erfüllt, kommt die Synchronisation nicht zustande.[7]

Diese Bidirektionalität führt zu einer grundsätzlich andersartigen Modellierung von Methoden und Schnittstellen bei Referenznetzen im Vergleich zu klassischen objektorientierten Sprachen. In der objektorientierten Softwareentwicklung ist es aufgrund des gerichteten Kontroll- und Informationsflusses in der Regel nötig, Schnittstellenpaare zu definieren, wenn asynchrone Ereignisse in beide Richtungen kommuniziert werden sollen. Beispielsweise definiert die Netscape-Plugin-API zwei Schnittstellen, von denen die eine Methodensignaturen, die durch den Webbrowser, und die andere solche, die durch das Plugin zu implementieren sind, vorgibt (siehe Tabelle 3.1). Grund dafür ist, dass sowohl auf Seite des Webbrowsers als auch auf Seite des Plugins Ereignisse auftreten können, die eine Kommunikation mit der Gegenseite auslösen und mitsamt Kontrollfluss durch die jeweilige unidirektionale Schnittstelle geleitet werden müssen.

[7] Das Prinzip des bidirektionalen Kontrollflusses wurde bereits beim Vergleich des Paradigmas der *Netze-in-Netzen* mit dem objektorientierten Paradigma erläutert (siehe Unterabschnitt 2.1.2.1).

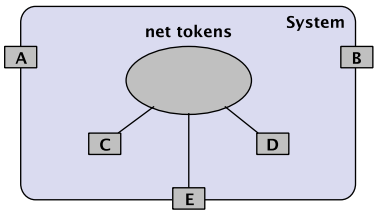
Eigentlich handelt es sich bei der NPAPI aber nur um *eine* Schnittstelle zwischen zwei Systemteilen, die teils asynchron miteinander kommunizieren. Die beiden auf konventionelle Weise getrennt und mit unterschiedlichem Umfang definierten Schnittstellen stellen zwei Seiten derselben Schnittstelle dar. Die Modellierung mit Referenznetzen bietet hier die Möglichkeit, die beiden Seiten der Schnittstelle mit jeweils identischem Umfang und spiegelbildlich definierten Up- und Downlinks zu formulieren. In der Modellierung mit Referenznetzen wäre die NPAPI als eine Schnittstelle realisierbar, die auf Seite des Webbrowsers ausschließlich aus Downlinks und auf Seite des Plugins ausschließlich aus Uplinks besteht, da auch Uplinks eine Kanalkommunikation von sich aus initiieren können.

#### 4.2.5.2 Innere und äußere Schnittstellen

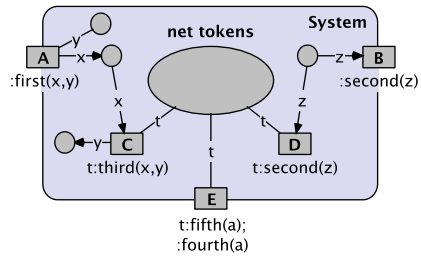
Die Spiegelbildlichkeit der Schnittstellendefinition in Referenznetzen geht einher mit der Möglichkeit, zwischen *inneren* und *äußeren* Schnittstellen eines Netzes unterscheiden zu können. Die äußere Schnittstelle wird durch Transitionen mit Uplinks definiert. Solche Kanäle stehen prinzipiell jeder anderen Netzinstanz offen, welche eine Referenz auf die betrachtete Netzinstanz verfügt. In der grafischen Darstellung in dieser Arbeit werden solche Transitionen auf dem Rand der durch ein abgerundetes Rechteck umschlossenen Netzinstanz platziert. Eine innere Schnittstelle wird durch Downlinks an Transitionen definiert, welche auf Netzmarken in bestimmten Stellen des Netzes zurückgreifen (üblicherweise alle Stellen, in denen sich gleichartige Marken eines Netztyps befinden können). Jede innere Schnittstelle eines Netzes muss zugleich äußere Schnittstelle eines anderen Netzes sein, um Kommunikation zwischen beiden Netzen zu ermöglichen. Die Begriffe innen und außen dienen also nur dem Benennen von Schnittstellen innerhalb eines Netzes, jede Schnittstelle wird in beiden Formen benötigt. In der klassischen Objektorientierung werden im Vergleich dazu lediglich äußere Schnittstellen explizit definiert. Deren Verwendung als innere Schnittstellen ist im Entwurf nicht unmittelbar sichtbar, sondern manifestiert sich in der Typisierung von Variablen, Feldern und Methodenparametern.

In Abbildung 4.3(a) zählen beispielsweise die Transitionen „A“, „B“ und „E“ zur äußeren Schnittstelle, während „C“, „D“ und „E“ eine innere Schnittstelle um die Stelle „net tokens“ definieren. Die Transition „E“ dient damit offensichtlich der Kommunikation über mehr als zwei Netzinstanzen hinweg.

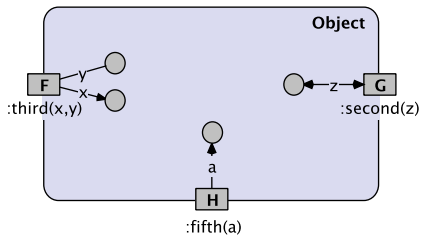
Die abstrakte Darstellung aus Abbildung 4.3(a) ist ohne textuelle Erläuterung oder eine konkrete Beschriftung mit Kanälen nicht vollständig. In Abbildung 4.3(b) sind exemplarisch die nötigen Netzanschriften und Stellen ergänzt worden, um Kanalsignaturen festzulegen. Der vorgesehene Kontroll- und Datenfluss kann im Netz nur teilweise festgelegt werden, weil sich das vollständige Verhalten erst im Zusammenspiel mit anderen Netzinstanzen ergibt. Beispielsweise könnte die Bindung der Marke „x“ im Kontext der Transition „C“ sowohl dem Informationstransfer in die Netzinstanz „t“ als auch in der Gegenrichtung (z. B. der Selektion aufgrund einer von der Netzinstanz „t“ vorgegebenen Bindung) dienen. In der grafischen Darstellung eines einzigen Netzes kann mittels Transitions- und Kantenanschriften die Daten- und Kontrollflussrichtung zum Teil eingeschränkt werden. Im Fall der Transition „C“ ist beispielsweise sichergestellt, dass keine Kanalsynchronisation zustande kommt, wenn keine Marke




(a) Abstrakte Darstellung




(b) Konkrete Darstellung mit Kanälen und Datenfluss




(c) Netzinstanz mit spiegelbildlicher äußerer Schnittstelle

Abbildung 4.3: Innere und äußere Schnittstellen eines Netzes (Beispiel)

zur Bindung an die Variable „x“ bereit liegt. Durch ständiges Bereithalten von passenden Marken lässt sich auch modellieren, dass eine Netzinstanz auf den Kontrollfluss einer anderen Netzinstanz wartet. Stellen, die solche Marken bereithalten, werden typischerweise über Testkanten an eine Transition angebunden, um die Nebenläufigkeit nicht einzuschränken.

Letztlich werden in jedem synchronen Kanal die individuellen Kontrollflüsse zweier Netzinstanzen gekoppelt – dies erfordert die Aktivierung der beteiligten Transitionen von *beiden* Seiten. Die Modellierung mit Referenznetzen lässt daher, solange nicht sämtliche beteiligten Netze bis ins Detail ausgearbeitet sind, während der Modellierung Freiheiten bzw. Unschärfen bezüglich des gewünschten Informationsflusses offen. Wünscht der Modellierer eine eindeutige Festlegung der im Netz offen gelassenen Details, ist im momentanen Entwicklungsstand des Referenznetzformalismus zusätzliche textuelle Dokumentation erforderlich. Eine Erweiterung des Formalismus, welche die Deklaration von Informationsflussrichtungen an synchronen Kanälen unterstützt, ist jedoch prinzipiell denkbar.

Abbildung 4.3(c) gibt ein exemplarisches Objektnetz wieder, welches mit einer zur inneren Schnittstelle aus Abbildung 4.3(b) spiegelbildlichen äußeren Schnittstelle versehen ist. Zu jenen Kanalparametern, an denen kein Informationsangebot vom Objektnetz erwartet wird, sind an den jeweiligen Transitionen lediglich Ausgangskanten

vorgesehen. Erst hier zeigt sich die intendierte Informationsflussrichtung der Variablen „x“ im Kanal „third“ auch in der grafischen Darstellung, da die Transition „F“ offensichtlich keine Vorgaben zur Bindung dieser Variablen macht.

Obwohl die vollständige Spezifikation von Kanalsignaturen erst mit detaillierten Anschriften und zusätzlicher Erläuterung gegeben ist, wird in den ersten Phasen der Modellierung des Plugin-Konzepts häufig auf die abstrakte Darstellungsvariante wie in Abbildung 4.3(a) zurückgegriffen werden. In diese werden lediglich nach Bedarf einige unter dem jeweiligen zu modellierenden Aspekt relevante Details zusätzlich eingearbeitet. Dieses Vorgehen stellt einen Kompromiss zwischen grafischer Klarheit und semantischer Vollständigkeit dar, der vertretbar scheint, da der intendierte Informationsfluss in der Regel aus dem Kontext des Modellierungsschrittes hervorgeht. Zudem bringt die dadurch entstehende Unschärfe Freiräume in die frühen Phasen des Modellierungsprozesses, die der Entwicklung eines einfachen Modells förderlich sind.

Gegenüber der klassischen Definition zweier unidirektionaler Schnittstellen ermöglicht es die spiegelbildliche Schnittstellendeklaration auf Basis bidirektionaler synchroner Kanäle, Informationen und Kontrollfluss in *beide* Richtungen auszutauschen, obwohl nur *eine* Seite über eine navigierbare Objektreferenz verfügt. Im Fall der NPAPI würde das Plugin keine Referenz auf das Objekt des Webbrowsers mehr benötigen, da seine bei Bedarf aktivierten Uplinks auch ohne direkte Kenntnis des Kommunikationspartners funktionieren.[8] Wenn aber die Kommunikationsrichtung kein Kriterium mehr für die Definition von Schnittstellen und das Vorhalten von Objektreferenzen ist, kann in der referenznetzbasierten Modellierung und Implementierung die Verteilung von *Uplink* und *Downlink* alleine nach den Kriterium der gewünschten Navigierbarkeit erfolgen. Hierzu zählen u.a. die gewünschte Struktur des Systems, die Kardinalität der Beziehung zwischen den an der Schnittstelle beteiligten Partnern ($1:1$, $1:n$ oder $n:m$) oder auch Aspekte der Kapselung oder Sicherheit. Solche Überlegungen werden bei Bedarf im Zuge der Modellierung näher ausgeführt.

#### 4.2.5.3 Abgrenzung einzelner Schnittstellen

Ergänzend sei an dieser Stelle auf die (möglicherweise) unterschiedliche Granularität von Systemgrenzen, Schnittstellen und Kanälen hingewiesen. Die Verwendung dieser drei Konzepte in der Modellierung kann, muss aber keineswegs entlang einer Eins-zu-Eins-Beziehung geschehen. In einer strengen Eins-zu-Eins-Beziehung verfügte jedes Netz über genau eine äußere Schnittstelle und jede Stelle für Netzmarken über genau eine innere Schnittstelle in Form eines einzigen Kanals. Der Modellierung in Abbildung 4.3 liegt aber der konzeptionelle Gedanke jeweils einer einzigen inneren und äußeren Schnittstelle je Netz zugrunde, so dass jeweils mehrere Kanäle zu einer Schnittstelle zusammengefasst werden. In einer weiteren Alternative könnten mehrere äußere Schnittstellen eines Netzes und auch mehrere innere Schnittstellen an einer Stelle unterschieden werden, die jeweils eine Teilmenge der angebotenen Kanäle umfassen. Die Wahl einer Option aus diesem Spektrum ist aus der Modellierung heraus als Entwurfsentscheidung zu treffen.

[8] Knapp gesagt: Durch die bidirektionale Kommunikation entfällt die leidige Diskussion, ob ein Ereignis mittels *Push*- oder *Pull*-Technik zu übertragen ist. Der Renew-Simulator implementiert in jedem Fall eine *Push*-Semantik.

Eine Eins-zu-Eins-Beziehung zwischen Schnittstellen und Kanälen hat – insbesondere während der frühen Modellierungsschritte – den Vorteil einer aufgeräumten, abstrakten Darstellung. Jede Schnittstelle lässt sich so durch eine einzige Transition mit einem Kanal repräsentieren, der bidirektionale Kommunikation zulässt. Diese Art der Modellierung stößt allerdings an ihre Grenzen, sobald Interesse an Details zu Zuständen und Abläufen im Rahmen der Kommunikation über Schnittstellen besteht. Die Modellierung von Prozessen oder Fallunterscheidungen wird durch das Unterscheiden mehrerer Transitionen (und Kanäle) erleichtert, welche aber fachlich zur selben Schnittstelle gehören sollen.[9] In konkreteren Modellierungsschritten soll daher im Rahmen dieser Arbeit eine Menge von Kanälen – bei Bedarf – zu einer Schnittstelle zusammengefasst betrachtet werden können.

Bei der Betrachtung von Downlinks in Nachbarschaft von Stellen mit Netzmarken bestehen mehrere Möglichkeiten, die angebotenen Kanäle einer Schnittstelle zuzuordnen. Aus der Modellierung heraus könnte es sich als wünschenswert erweisen, nicht sämtliche Kanäle im Kontext einer Stelle zu einer Schnittstelle zusammenzufassen, wenn beispielsweise die hinter den Kanälen stehende Funktionalität nach fachlichen Aspekten in verschiedene Gruppen eingeteilt werden kann. Dieses Vorgehen würde die Wiederverwendung der einzelnen Schnittstellen in anderen Kontexten erleichtern und zudem die Unterscheidung von Netzmarken nach der Menge der unterstützten Schnittstellen erlauben. So müsste nicht jede Netzmarke passende Uplinks zu sämtlichen Kanälen bereitstellen, welche um eine die Marke enthaltende Stelle herum angeboten werden, sondern könnte sich auf einen Teil der Kanäle entsprechend eines Teils der im Entwurf unterschiedenen Schnittstellen beschränken.

Konzeptionell klarer erscheint jedoch der Ansatz, alle im Kontext einer Stelle deklarierten Downlinks zu einer gemeinsamen Schnittstelle zusammengefasst zu betrachten. Unterschiedliche innere Schnittstellen können immer noch innerhalb eines Netzes verwendet werden, wenn die Netzmarken nach Schnittstellen getrennt in mehreren Stellen verwaltet werden. Dies entspricht der Idee der Typisierung von Stellen, wie es gefärbte Petrinetze (und auch Java-Referenznetze) zur Unterscheidung von Nicht-Netzmarken-Typen ohnehin schon vorsehen. Der Vorteil dieser Technik ist, dass die verschiedenen inneren Schnittstellen grafisch voneinander getrennt dargestellt werden und somit die Separation fachlicher Aspekte in der Netzgrafik ihre Entsprechung finden kann.

Falls eine Netzmarke zu mehreren inneren Schnittstellen eines äußeren Netzes passend ist, kann das durch die Marke repräsentierte Netz im Rahmen der Referenzsemantik von mehreren Stellen des äußeren Netzes aus referenziert werden. Somit stellt eine nach Typen getrennte Verwaltung von Netzmarken keine Einschränkung der Ausdrucksmächtigkeit im Vergleich zur gemischt-typisierten Verwaltung auf einer Stelle dar.

Schnittstellen können auch tolerant deklariert werden, so dass ein inneres Netz nicht zu sämtlichen Kanälen einer Schnittstelle passen muss. Letztlich entspricht ein vom

[9] Statt mehrere Kanäle zu unterscheiden, lassen sich diese auch zu einem Kanal zusammenfalten, der durch Setzen eines Parameters in verschiedenen Modi betrieben werden kann. Dieses Vorgehen verlagert allerdings die Information zur Zuordnung passender Kanäle von den in der Netzstruktur sichtbaren, fest kodierten Anschriften in erst zur Laufzeit evaluierbare Variablenbindungen auf der Basis von Marken. Ein Bedarf an mehreren Transitionen lässt sich bei detaillierter Modellierung von Prozessen und Zuständen nicht umgehen.

inneren Netz aufgrund des fehlenden passenden Uplinks *nicht* genutzter Kanal im Verhalten exakt einem Kanal, der aufgrund eines zwar vorhandenen, aber niemals aktivierten Uplinks nicht genutzt werden kann. Das *Nicht*-Implementieren eines Kanals durch Netzmarken kann daher zulässig sein, wenn der Schnittstellenvertrag das Nicht-Aktivieren dieses Kanals zulässt.

Mit Bezug auf die äußeren Schnittstellen eines Netzes bieten sich ähnliche Modellierungsmöglichkeiten bei der Zuordnung von Kanälen zu Schnittstellen. Für die Betrachtung aller Uplinks eines Netzes als *eine* gemeinsame Schnittstelle spricht der technische Umstand, dass Referenznetze bisher über keine objektorientierte Typhierarchie auf Basis von Schnittstellen verfügen. Die dynamische Bindung erlaubt sämtlichen Netzen, denen eine Referenz auf eine fragliche Netzinstanz zur Verfügung steht, die Kopplung mit sämtlichen in dieser Netzinstanz angebotenen Uplinks. Die Netzinstanz hat keine Möglichkeit, anstehende Kanalkommunikationen nach Herkunft zu unterscheiden und somit auf deren Zulässigkeit im Rahmen verschiedener Schnittstellenverträge zu achten.

Dennoch kann mit dem Ziel einer klareren Modellierung die Menge der nach außen angebotenen Kanäle konzeptionell in mehrere Schnittstellen unterteilt werden. Dieses Vorgehen erlaubt es, Kanäle nach unterschiedlichen fachlichen Aspekten separiert zu betrachten. Die konzeptionelle Aufteilung in mehrere Schnittstellen entspricht in etwa der beispielsweise in der objektorientierten Programmiersprache Java technisch vorhandenen Möglichkeit, ein Objekt als Implementierung mehrerer Interfaces auszuweisen. Referenzen auf dieses Objekt können dann jeweils mit einer der Schnittstellen typisiert und per „*Type Cast*“ von einer Schnittstelle zur anderen umgewandelt werden.

Zwar fehlen derzeit im Referenznetzformalismus die Möglichkeiten, die konzeptionelle Zuordnung von Uplinks zu verschiedenen Schnittstellen technisch zu fixieren, in der konzeptionellen Modellierung kann aber dennoch eine entsprechende Unterteilung vorgesehen werden. Zum einen ist die zukünftige Definition eines derartigen Typsystems für Referenznetze durchaus denkbar. Zum anderen lässt sich bereits mit den vorhandenen technischen Möglichkeiten unter Referenzsemantik eine Zwischenschicht in Form einer Hülle selektiver, schnittstellenkapselnder Netze um das eigentliche Netz herum implementieren. Eine schnittstellenkapselnde Netzinstanz besteht lediglich aus einer Transition je zulässigem Kanal, die zugleich mit Up- und Downlink zu diesem Kanal beschriftet ist und die Kanalparameter durchreicht. Anstelle von Referenzen auf die eigentliche Netzinstanz werden lediglich Referenzen auf selektiv kapselnde Netzinstanzen kommuniziert, so dass jedes Netz nur noch die in der kapselnden Netzinstanz vorgesehenen Kanäle nutzen kann. Dieses Vorgehen ist ein üblicher Weg, in nicht-objektorientierten Programmiersprachen Typsicherheit nachzurüsten, und entspricht in etwa der in Unterabschnitt 3.1.2.1 beschriebenen XPCOM-Architektur zur Simulation eines objektorientierten Typsystems innerhalb der Mozilla-Anwendungssuite.

Im Rahmen dieser Arbeit werden also mehrere äußere Schnittstellen je Netz unterschieden, wobei die Zuordnung von Kanälen zu Schnittstellen in der Regel textuell erläutert wird. Innere Schnittstellen werden jedoch nicht feiner unterteilt, als es die im Netz vorhandenen Stellen zur Verwaltung von Netzmarken erlauben. Jede Schnittstelle wird sowohl als innere Schnittstelle als auch als spiegelbildliche äußere Schnittstelle der zueinander passenden Netze betrachtet. Jede Schnittstelle kann mehrere Kanä-

le umfassen, von denen gegebenenfalls nicht alle in beiden Kommunikationspartnern aktivierbar sein müssen, falls die fachliche Bedeutung der Schnittstelle das Nicht-Aktivieren von Kanälen zulässt.

#### 4.2.5.4 Typen

Die Überlegungen des vorigen Unterabschnitts zur Unterteilung der Up- und Downlinks eines Netzes in mehrere äußere und innere Schnittstellen erlauben es, Netze (zumindest auf der konzeptionellen Ebene) nach Typen zu unterscheiden. Es sei hier nochmals angemerkt, dass der Java-Referenznetzformalismus bisher keine formale oder operationale Möglichkeit bietet, Netze zu typisieren, sondern alle Netze gleich behandelt. Die dynamische Bindung synchroner Kanäle führt aber dazu, dass nur „passende“ Netze sinnvoll miteinander kommunizieren können. Es ist daher für die Modellierung wünschenswert, sowohl Netzmarken als auch Variablen und Stellen nach passenden Typen unterscheiden zu können.

Im Folgenden sei die grundsätzliche Idee eines solchen Typsystems skizziert. Eine vollständige und formale Ausarbeitung des Typsystems ist jedoch die Aufgabe zukünftiger, weiterführender Arbeiten. Für die abstrakten Modelle in dieser Arbeit genügt die Möglichkeit, konzeptionell davon ausgehen zu können, dass nur passende Marken über Kanten und durch Stellen bewegt werden.

Eine Schnittstelle umfasst – wie zuvor beschrieben – eine Menge von Kanalsignaturen, wobei eine äußere Schnittstelle ausschließlich Uplinks und eine innere Schnittstelle ausschließlich Downlinks enthält. Jede Schnittstelle existiert sowohl als äußere als auch als innere Schnittstelle mit spiegelbildlichen Kanalsignaturen. Ein Netz wird durch seine äußeren Schnittstellen getypt, so dass jede Schnittstelle einem alternativen Typ des Netzes entspricht. Eine Variable, Kante oder Stelle wird durch eine innere Schnittstelle getypt, welche (bei sinnvoller Modellierung) die in der Nachbarschaft vorhandenen Downlinks beschreibt. Ein getyptes Netz passt zu einer getypten Variable, Kante oder Stelle, wenn es sich bei der durch den jeweiligen Typ benannten äußeren und inneren Schnittstelle um genau die beiden spiegelbildlichen Varianten ein- und derselben Schnittstelle handelt.

Das so skizzierte Typsystem kann eine Spezialisierungshierarchie beinhalten. Eine Schnittstelle ist grob gesagt spezieller als eine andere, wenn die Kanalsignaturen der ersten Schnittstelle eine Obermenge der Signaturen der zweiten Schnittstelle darstellen. Werden beim Vergleichen der Schnittstellen auch die Typen der Kanalparameter zueinander in Beziehung gesetzt, so können weitere Spezieller-als-Beziehungen zwischen Schnittstellen definiert werden.

In einer getypten Stelle können unter Berücksichtigung der Spezialisierungshierarchie sämtliche Netzmarken gelagert werden, deren Netztyp (äußere Schnittstelle) gleich dem oder spezieller als der Typ der Stelle (spiegelbildliche innere Schnittstelle) ist. Diese Beziehung gilt analog für getypte Variablen oder Kanten und die daran gebundenen Netzmarken. Die Schaltregel der Petrinetze kann darauf aufbauend so eingeschränkt werden, dass eine Transition nur schalten kann, wenn alle Typen der zu bindenden Netzmarken zu den jeweiligen Variablen, Kanten und Stellen unter Berücksichtigung der Spezialisierungshierarchie passen. Eine solche Schaltregel erlaubt

es, Nichtdeterminismus im Modell einzuschränken, indem z. B. Konflikte zwischen aktivierbaren Transitionen anhand des Typsystems aufgelöst werden.

Wie schon im vorigen Unterabschnitt gilt auch hier, dass im Java-Referenznetzformalismus Möglichkeiten bestehen, die aufgrund des nicht-vorhandenen Typsystems fehlenden Eigenschaften nachzubilden. Es ist beispielsweise möglich, die Typen von Netzen und Stellen durch Zeichenketten zu identifizieren und die Einschränkung der Schaltregel durch explizite Transitionsanschriften durchzusetzen. Die Spezialisierunghierarchie müsste bei dieser Technik allerdings explizit in den Zeichenketten oder Anschriften implementiert werden. Alternativ kann beispielsweise auf das in den Java-Referenznetzen ebenfalls verwendbare Java-Typsystem zur Unterscheidung von Netzmarken zurückgegriffen werden. Diese Technik erfordert allerdings die Definition einer eigenen Java-Klasse für jedes Netz.[10] Im Interesse der Klarheit des Modells werden solche technischen Details während der konzeptionellen Modellierung ausgeblendet werden. Sofern das Verhalten eines Netzmodells von der Typisierung der Variablen, Kanten oder Stellen abhängen soll, wird dieses im Text informell erläutert werden.

## 4.2.6 Grundlegende Abstraktionen

Während der Untersuchung existierender Plugin-Systeme in Kapitel 3 sind einige Eigenschaften besprochen worden, welche letztlich der konkreten technischen Implementierung und Funktionalität der Systeme in einem speziellen Kontext geschuldet sind. In der Dokumentation dieser in der Praxis eingesetzten Systeme wird entsprechenden Eigenschaften oftmals ein besonders großer Raum zuteil, da die beteiligten Programmierer auf genau diese technischen Details treffen und Lösungen für sich daraus ergebende Probleme benötigen. Im Interesse einer klaren konzeptionellen Modellierung erscheint es aber sinnvoll, von solchen kontextspezifischen Eigenheiten einzelner Systeme zu abstrahieren. Im Folgenden werden daher einige Abstraktionen erläutert, welche den Blick auf das Wesentliche und die Gemeinsamkeiten von Plugin-Systemen frei machen. Diese Abstraktionen werden von Beginn der konzeptionellen Modellierung an in Abschnitt 4.3 eingesetzt werden, um zu einem einheitlichen Modell zu kommen. Einige dieser Abstraktionen werden im Anschluss an die Modellierung in Abschnitt 4.4 erneut in der Diskussion aufgegriffen, um einen Abgleich des konzeptionellen Modells mit den untersuchten Systemen aus der Praxis zu ermöglichen.

### 4.2.6.1 Abstraktion von Lademechanismen und -zuständen

Nach Definition 12 sind Plugin-Systeme *offene* Systeme. Wie bereits im Zusammenhang der Definition 4 erläutert, können offene Systeme *dynamisch* rekonfigurierbar sein, müssen dies aber nicht. In der allgemeinen Modellierung des Plugin-Konzepts sollte daher das Hinzufügen und Entfernen von Plugins sowohl während als auch außerhalb der Laufzeit des Systems möglich sein.

Im Kontext existierender oder zu implementierender Plugin-Systeme führt die Unterscheidung der Rekonfigurationszeitpunkte dazu, dass die Mechanismen und Zu-

[10] Das Werkzeug Renew unterstützt die Definition von Java-Klassen, deren Funktionalität als Java-Referenznetz implementiert ist, durch das Konzept der „Stubs“ (siehe [KWD06b, Abschnitt 3.9]).

standsübergänge des zugrundeliegenden Modulladesystems[11] vollständig in die Modellierung und Implementierung des Plugin-Systems übernommen werden. Es wird relevant, ob ein Code-Modul nur als momentan ungenutztes Objekt im Dateisystem vorrätig gehalten oder ob es bereits in die Laufzeitumgebung der Anwendung integriert, also im Speicher, vorgehalten wird. Auf die fachliche Funktionalität der Anwendung hat diese Unterscheidung jedoch keine Auswirkung, da deren Funktionalität sich erst durch einen weiteren Zustandsübergang während der fachlichen Integration des Moduls in die Softwarearchitektur der Anwendung verändert.

Der Unterschied zwischen dynamischer und nicht-dynamischer Rekonfiguration ist in den Modellen vor allem an der Struktur der Lebenszyklusnetze zu erkennen, wie sie in Kapitel 3 für die untersuchten Plugin-Systeme herausgearbeitet wurden. Bei der Diskussion der unterschiedlichen Lebenszyklen in Unterabschnitt 4.4.2 wird auch dieser Unterschied detailliert besprochen werden, an dieser Stelle soll eine kurze Zusammenfassung genügen. Die meisten Plugin-Systeme haben sich auf eine der Varianten – dynamisch oder nicht – festgelegt. Lediglich der Lebenszyklus der NetBeans-Module berücksichtigt explizit beide Möglichkeiten des Rekonfigurationszeitpunkts, was sich in einer Verdopplung der „Install“- und „Uninstall“-Transitionen in Abbildung 3.15 bemerkbar macht.

Systeme mit nicht-dynamischer Rekonfiguration sind am Vorhandensein von mehreren unterscheidbaren Zuständen für inaktive Plugins (wie z. B. „installed“ und „known“) und von plattformweiten Zustandsübergängen (blassgrau dargestellte Netzelemente) in den Lebenszyklen der einzelnen Plugins zu erkennen. Diese Übergänge stellen ein Problem dar, weil sie die Lebenszyklen sämtlicher Plugins synchronisieren, selbst wenn weder fachlich noch technisch eine Abhängigkeit zwischen den einzelnen Plugins besteht. Eine Aufnahme dieser Synchronisation in ein Modell des Plugin-Konzepts würde das Modell auf eine nicht-dynamische Rekonfiguration festlegen. Umgekehrt stellt aber der Verzicht auf so eine Kopplung keine Einschränkung dar, da Plugins, die für dynamisch rekonfigurierbare Systeme entwickelt wurden, auch in einem nicht-dynamischen System mit einer einzigen Konfigurationsphase bei Systemstart integrierbar sind.

Im Interesse einer allgemeinen, einfachen Modellierung, welche sowohl dynamische als auch nicht-dynamische Plugin-Systeme erfasst, soll daher bei der folgenden Modellierung von Zuständen abstrahiert werden, die lediglich für den zugrundeliegenden Modulladememechanismus interessant sind. Diese Abstraktion reduziert die Komplexität von Lebenszyklusmodellen erheblich, ohne ihre Verwendbarkeit auf dynamisch oder nicht-dynamisch rekonfigurierbare Systeme einzuschränken. Zudem entspricht diese Abstraktion dem Trend hin zu dynamisch rekonfigurierbaren Systemen, dem viele der in Kapitel 3 untersuchten Systeme folgen.

Zu beachten ist allerdings, dass mit dieser Abstraktion die Modellierung einiger nichtfunktionaler Eigenschaften von Plugin-Architekturen eingeschränkt oder ausgeschlossen wird. So fehlt z. B. dem unter Eclipse zum Erhöhen der Performanz propagierten Lazy-Loading-Prinzip (siehe Unterabschnitt 3.2.1.4) ohne eine Unterscheidung zwischen installiertem und geladenem Modulcode jegliche Modellierungsgrundlage.

[11] Modulladesysteme wurden beispielhaft im Kontext einiger Plugin-Systeme vorgestellt: betriebssystemspezifisches Laden und Verknüpfen von Code-Modulen in Unterabschnitt 3.1.1.2, das dynamische Laden von Java-Klassen in Unterabschnitt 3.2.1.2 und einige Anmerkungen zu den Probleme darauf aufbauender Softwaresysteme in Unterabschnitt 3.3.8.

Der Schwerpunkt der Modellierung in dieser Arbeit liegt jedoch auf dem grundlegenden und korrekten Verhalten eines Plugin-Systems beim Hinzufügen und Entfernen von Plugins. Nichtfunktionale Eigenschaften können in späteren Arbeiten ergänzend modelliert werden.

#### 4.2.6.2 Abstraktion von deklarativ spezifizierten Eigenschaften

In einem intendiert erweiterbaren System müssen notwendigerweise die beim Entwurf zur Integration vorgesehenen und während der Einsatzphasen genutzten Schnittstellen nicht nur dem Entwickler, sondern auch dem in Ausführung befindlichen Softwaresystem bekannt sein. Dies bedeutet, dass jedes Plugin seine Schnittstellen in maschinenlesbarer Form mitführen muss.

Sämtliche in Kapitel 3 untersuchten Plugin-Systeme sehen zusätzlich in der globalen Plugin-Verwaltung oder in den fachlichen Dienst- und Erweiterungsspezifikationen Möglichkeiten vor, die Integration von Systemkomponenten zu parametrisieren. Die zur Ermittlung der Integrationsparameter verwendeten Techniken variieren vom imperativen Abfragen einer vereinbarten API (z. B. XPCOM) über die Extraktion von Typinformationen der verwendeten Programmiersprache (z. B. MagicBeans) bis zur Deklaration in separaten Dateien (z. B. Eclipse). Jede dieser Varianten bringt neben offensichtlichen Vorteilen auch individuelle Nachteile mit sich. Das Ermitteln von Parametern mittels einer API erfordert die wiederholte Implementierung fast identischen Codes in jeder beteiligten Systemkomponente. Die Extraktion von Informationen aus dem Typsystem der Programmiersprache bietet einen geringeren Informationsgehalt, da z. B. quantitative Parameter nicht in Form von Typdeklarationen ausgedrückt werden können. Die separate Deklaration zusätzlich zur Deklaration von Typen in der zugrundeliegenden Programmiersprache erfordert den fortlaufenden Abgleich der beiden Deklarationsebenen während der Entwicklung, um Inkonsistenzen auszuschließen.

Für das zu modellierende Plugin-Konzept ist die Unterscheidung der verschiedenen Spezifikationstechniken für Integrationsparameter allerdings nicht relevant – es zählt vielmehr die konzeptionelle Möglichkeit, überhaupt Integrationsparameter angeben zu können. Daher wird in der folgenden Modellierung auf eine Diskussion dieser Unterschiede verzichtet. Stattdessen wird zur Modellierung von Parametern im Interesse der Klarheit des Modells auf die einfachsten Möglichkeiten zurückgegriffen, die von der gewählten Modellierungssprache der Java-Referenznetze geboten werden.

Die dynamische Bindung ungetypter Referenznetze erlaubt das Betrachten der Kanalsignaturen eines Netzes als Schnittstellendeklaration. Werden auch Schnittstellenidentifizierende Bezeichner benötigt, so können diese als Marken vom Typ `String` in Stellen bereitgehalten und mittels synchroner Kanäle kommuniziert werden. Auf diese Weise sind alle drei zuvor angesprochenen Varianten im Modell abgedeckt: Das Kommunizieren der in den Plätzen bereitgehaltenen Zeichenketten lässt sich als Extraktion von Kanalsignaturen aus dem Plugin-Code heraus interpretieren. Es kann aber auch als Abfrage über eine in Form von Kanälen definierten API verstanden werden. Ebenso ist eine Interpretation als Bereitstellung deklarativer Informationen in einem separaten Datenpool (dem jeweiligen Platz) möglich.

#### 4.2.6.3 Abstraktion von konkreter Funktionalität

Anders als eine explizit auf den einzigen Zweck der Architekturmodellierung hin ausgelegte grafische Sprache kann die gewählte Sprache der Java-Referenznetze auch als vollwertige Implementierungssprache dienen (siehe Unterabschnitt 2.1.4). Gemäß der Technik der schrittweisen Verfeinerung (siehe Unterabschnitt 4.2.2) sollten Details, die der Implementierung konkreter Funktionalität dienen, während der Modellierung zunächst ausgeblendet bleiben. Sie wären dem Ziel dieser Arbeit, der Entwicklung eines einheitlichen und übersichtlichen Konzepts für Plugin-Systeme, nur hinderlich, da sie den Blick auf die wesentlichen Prozesse und Strukturen des Konzepts verstellten.

Dennoch müssen im Modell bereits Platzhalter für konkrete Funktionalität vorgesehen werden, um die Anknüpfungspunkte bei der Integration von Systemkomponenten zur Einsatzzeit eines Plugin-Systems beschreiben zu können. In der Erfahrung des Autors hat es sich bei der Modellierung mit Referenznetzen als sinnvoll herausgestellt, als Platzhalter für konkrete Funktionalität einzelne Transitionen zu verwenden. In der grafischen Darstellung werden diese Transitionen weiß repräsentiert werden.

Eine Transition steht in der Modellierung für ein aktives Element und ist daher geeignet, abstrahierte Funktionalität zu verkörpern. Mit der Technik der Verfeinerung von Petrinetzen lässt sich eine Transition beim Übergang zur konkreten Implementierung durch komplexe Netze ersetzen. Eine Transition kann zudem auf konkrete Funktionalität in einem anderen Teil des Netzes oder in anderen Netzinstanzen zugreifen, indem zur Schnittstelle des jeweiligen Netzes passende synchrone Kanäle verwendet werden.

Eine als Platzhalter für konkrete Funktionalität verwendete Transition steht in der Regel relativ isoliert im Netz. Nur die zur Modellierung des Konzeptes minimal notwendigen oder zur exemplarischen Veranschaulichung sinnvollen Kanten und Kanäle sind an die Transition angebunden. Nichtsdestotrotz kann bei fortschreitender Verfeinerung die Transition Teil eines komplexen Petrinetzes werden, welches zusätzliche Beziehungen zu anderen Platzhalter-Transitionen herstellt. Daher ist zu berücksichtigen, dass die fast uneingeschränkte Nebenläufigkeit, die solche Transitionen im groben Modell aufweisen, im konkreten System häufig stark eingeschränkt wird. Außerdem kann die Technik der Petrinetz-Verfeinerung dazu führen, dass die Atomizität des Schaltvorganges, über die eine Platzhalter-Transition natürlich verfügt, im konkreten Netz nicht mehr gegeben ist. Beide Effekte sind bei der Modellierung mit Platzhalter-Transitionen zu berücksichtigen, so dass weder die Atomizität noch die volle Unabhängigkeit der modellierten Funktionalität im umgebenden Teil des Netzes vorausgesetzt werden darf.

### 4.2.7 Verzahnung von Systemstruktur und Lebenszyklen

Wie bereits bei der Wahl der Modellierungssprache (siehe Unterabschnitt 4.2.1) argumentiert, eignen sich Referenznetze sowohl zur Modellierung von Abläufen (samt komplexer Zustände) als auch von Systemstruktur (vor allem der Schnittstellen und Infrastruktur zur Verbindung von Systemkomponenten). Zur Modellierung des Plugin-Konzepts ist die Integration beider Aspekte nötig, da das Konzept sowohl Struktur

als auch Abläufe im System umfasst. Offen ist dabei die Frage, wie diese Integration stattfinden kann.

In vorangegangenen Kapiteln dieser Arbeit finden sich bereits Beispiele für Netze, welche jeweils einen Aspekt modellieren: In Kapitel 3 wurden mit den Lebenszyklusnetzen bereits mehrfach die Zustandsmodellierungsmöglichkeiten genutzt.[12] Die in Abschnitt 2.5.2 vorgestellten äußeren Ebenen der Mulan-Architektur (vor allem System und Plattform, aber auch der Agent selber) modellieren Systemstruktur und Interaktionsmöglichkeiten. Zustands- und Strukturmodelle nutzen sehr unterschiedliche Techniken der Modellierung mit Petrinetzen. Während bei der Zustandsmodellierung in der Regel S/T-Netze zum Einsatz kommen, in denen Stellen und Kanten genutzt werden, um kausale oder zeitliche Abhängigkeiten zwischen den einzelnen Transitionen zu definieren, arbeiten Strukturmodelle stark mit dem Paradigma der *Netze-in-Netzen* und verwenden Stellen und Kanten – wie bereits in Unterabschnitt 4.2.5.2 skizziert – überwiegend zur Versorgung synchroner Kanäle mit passenden Netzmarken.

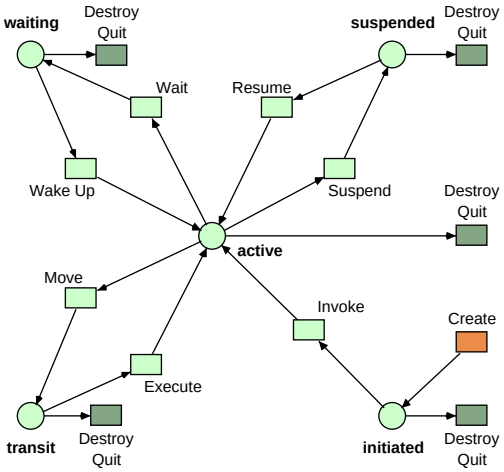
Zur Illustration sind in Abbildung 4.4(a) ein zustandsorientiertes Lebenszyklusnetz und in Abbildung 4.4(b) ein strukturorientiertes Plattformnetz aus dem Kontext der Multiagentensysteme (siehe Abschnitt 2.5.1.2 und Abschnitt 2.5.2) wiedergegeben. Beide Netze modellieren jeweils einen Aspekt von Agenten in einem Multiagentensystem. Im Lebenszyklusnetz werden die möglichen Zustandsübergänge des Agenten aus Sicht der FIPA-Agentenverwaltung modelliert. Ein Agent kann sich nur in einem Zustand zur Zeit befinden. Im Plattformnetz werden die grundlegenden Dienste, welche den Agenten während ihres Aufenthalts auf einer Plattform zur Verfügung stehen, modelliert. Sämtliche Agenten werden als Marken auf einer einzigen Stelle repräsentiert, von welcher aus die Dienste der umliegenden Transitionen in fast beliebiger Reihenfolge genutzt werden können.

Um beide Aspekte in einem Netz zu modellieren, kann das Lebenszyklusnetz mit dem Architekturnetz verzahnt werden. Das Netz in Abbildung 4.4(c) ist ein Resultat einer solchen Verzahnung.[13] Die Marken im verzahnten Netz sind Netzreferenzen auf Agenten, die zugleich durch ihren Aufenthalt in einer bestimmten Stelle den aktuellen Lebenszykluszustand des jeweiligen Agenten kodieren. Im verzahnten Netz gilt daher wie im Lebenszyklusnetz, dass jede Agenten-Netzinstanz aus maximal einer Stelle zur Zeit als Marke referenziert werden darf.

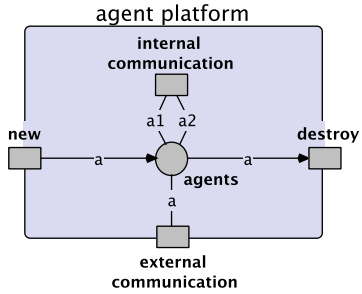
Das Grundprinzip der Verzahnung ist an der Stelle „active agents“ zu beobachten: Alle Plattformdienste (mit Ausnahme von „new“ und „destroy“, eine Begründung folgt im nächsten Absatz) werden an diese Stelle angebunden, so dass sie von Agenten im Lebenszykluszustand „active“ genutzt werden können. Vom Grundprinzip her ist diese Anbindung für jeden Zustand des Lebenszyklus' zu wiederholen, so dass die Transitionen zur Bereitstellung der Plattformdienste im kombinierten Netz mehrfach dupliziert auftreten müssten.

[12] Die Technik der Lebenszyklusmodellierung wird in Unterabschnitt 4.4.2.1 präziser erläutert werden.

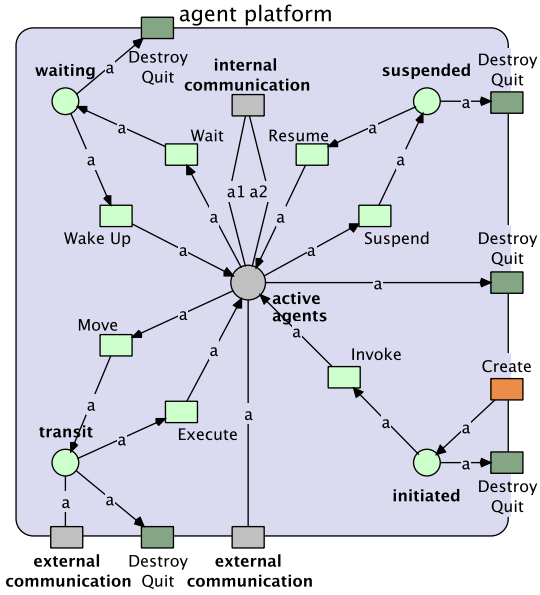
[13] Das Netz in Abbildung 4.4(c) ist nicht das reine Produkt einer Verzahnung. Es ist bereits (entsprechend den folgenden Erläuterungen im Text) verschlankt worden, indem Transitionen entfernt wurden, die aufgrund zusätzlichen Wissens über die Privilegien von Agenten in bestimmten Lebenszykluszuständen als unnütz identifiziert wurden.




(a) Zustandsmodell: FIPA-Lebenszyklus (siehe Abbildung 2.13)




(b) Strukturmodell: Plattformnetz (siehe Abbildung 2.14)




(c) Kombiniertes Modell

Abbildung 4.4: Mögliche Verzahnung von Zustands- und Strukturmodellierung am Beispiel der Mulan-Plattform

In der Regel kann die Duplikation von Diensttransitionen vermieden werden, wenn Wissen über die in bestimmtem Lebenszykluszuständen nicht benötigten Plattformdienste in die Verzahnung einfließt. In Abbildung 4.4(c) ist beispielsweise angenommen worden, dass lediglich zur Realisierung mobiler Agenten im Lebenszykluszustand „transit“ ebenfalls externe Kommunikation benötigt wird, in allen anderen Zuständen jedoch kein Kommunikationsbedarf besteht. Weitere Einsparungsmöglichkeiten bestehen, wenn bekannt ist, dass bestimmte Dienste nur exakt zum Zeitpunkt eines oder mehrerer Zustandsübergänge benötigt werden. Das Hinzufügen und Entfernen von Agenten zur bzw. von der Plattform (Transitionen „new“ und „destroy“) fallen mit Lebenszyklusübergängen „Create“ und „Destroy“ bzw. „Quit“ zusammen und treten daher nicht mehr getrennt im Modell auf.

Die Verzahnung von fachlich verwandten Zustands- und Strukturmodellen ist zwar grundsätzlich immer möglich, kann aber zu unübersichtlichen Ergebnissen führen, wenn viele Dienste in vielen Zuständen benötigt werden. In der folgenden Modellierung des Plugin-Konzepts werden daher zunächst die Strukturmodelle getrennt von den Zustandsmodellen besprochen, um die grafische Klarheit der Netze während der Modellierung zu erhalten. Die eigentliche Verzahnung wird erst später in Unterabschnitt 4.3.8.2 diskutiert und auch nur exemplarisch (in Unterabschnitt 4.4.2) durchgeführt werden, da ihr Ergebnis stark von den individuellen Eigenschaften konkreter Plugin-Systeme abhängt.

## 4.3 Schrittweise Modellierung des Plugin-Konzepts

In Abschnitt 4.1 wurden zentrale Anforderungen an ein einheitliches Konzept sowie eine grundlegende Definition für Plugin-Systeme aus den Ergebnissen der Untersuchung existierender Plugin-Systeme (siehe Abschnitt 3.4) herausgearbeitet. Auf Basis dieser Anforderungen soll im Folgenden ein generelles Modell für Plugin-Systeme unter Verwendung der im vorigen Abschnitt erläuterten Modellierungstechnik erstellt werden.

Gemäß dem in Unterabschnitt 4.2.1 vorgestellten Prinzip beginnt die Entwicklung des Modells mit einer abstrakten Modellierung des grundlegenden Konzepts. Nach Unterabschnitt 4.1.9 ist das zentrale Konzept eines Plugin-Systems das Konzept der intendierten Erweiterbarkeit, wie es in Definition 5 festgelegt wurde. Dieses Konzept wird in mehreren Schritten ausgearbeitet, um eine generelle Verwendbarkeit im Entwurf von Software-Architekturen zu erreichen und um weitere der in Abschnitt 4.1 besprochenen Anforderungen an Plugin-Systeme zu integrieren.

Das Ergebnis der Modellierung ist eine unter dem Paradigma der *Netze-in-Netzen* formulierte Referenzarchitektur eines Plugin-Systems, welches durch weitere Verfeinerung der Netze an die Gegebenheiten eines konkreten Plugin-Systems angepasst werden kann. Diese weiter konkretisierende Verfeinerung ist nicht Gegenstand dieses Kapitels, sie wird exemplarisch in Abschnitt 5.3 vorgenommen werden.

Das in diesem Abschnitt präsentierte Modell wurde vom Autor bereits in geringerer Detailtiefe in [CDMR05] skizziert.

### 4.3.1 Abstraktes Grundmodell

Von den drei in Definition 12 genannten Eigenschaften (offen, einfach konfigurierbar und intendiert erweiterbar) eines Plugin-Systems ist die *intendierte Erweiterbarkeit* diejenige, welche Struktur und Verhalten des Systems am stärksten prägt.[14] Daher ist es sinnvoll, im ersten abstrakten Modellierungsschritt das Konzept der Erweiterbarkeit als Grundlage zu wählen.

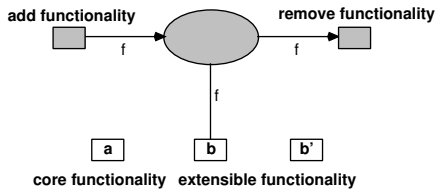



Abbildung 4.5: Grundmodell eines dynamisch erweiterbaren Systems

In Abbildung 4.5 ist das Grundprinzip eines intendiert erweiterbaren Systems auf abstrakte Weise als Petrinetz dargestellt. Dieses Netz kann zunächst ganz allgemein als Modell eines flexibel um Funktionalität erweiterbaren Systems betrachtet werden, ohne es bereits als Architekturbild mit scharfen Grenzen von Systemen oder Systemkomponenten diskutieren zu müssen. Die Elemente des Modells repräsentieren sowohl eine technische Infrastruktur zur Integration von Erweiterungen als auch – auf abstrakte Weise – fachliche Integrationsvorgaben. Die fachlichen Vorgaben existieren in Form von Schnittstellen zwischen vorhandener und hinzufügbarer Funktionalität.

Das Modell beinhaltet zunächst eigene Kernfunktionalität, welche nicht von Erweiterungen abhängt (exemplarisch modelliert durch Transition „a“). Daneben existiert auch solche Kernfunktionalität, die auf dynamisch integrierte Erweiterungen zurückgreifen kann (exemplarisch modelliert durch Transition „b“), im Folgenden als *erweiterbare* Funktionalität bezeichnet. Soll die erweiterbare Kernfunktionalität auch bei Abwesenheit von passenden Erweiterungen erbracht werden können, erfordert die Petrinetzsemantik eine Duplikation der entsprechenden Transitionen ohne den Rückgriff auf Erweiterungen (exemplarisch modelliert durch Transition „b'“).

Der obere, grau dargestellte Bereich des Netzes beschreibt die (technische) Verwaltung von Funktionalität, die als Erweiterung integriert werden kann. Die beiden Transitionen „add“ und „remove“ ermöglichen das Hinzufügen und Entfernen von Funktionalität, die zentrale Stelle des Netzes hält solche Funktionalität als Marken vor. Über Testkanten kann die Funktionalität im Kontext einer erweiterbaren Kernfunktionalität beliebig oft und auch nebenläufig genutzt werden.

Das abstrakte Modell aus Abbildung 4.5 ist nicht an ein konkretes Verständnis des Systems gebunden. Die Funktionalitätsverwaltung kann sowohl als Modell der Installation von Dateien auf Betriebssystemebene als auch als Modell der dynamischen

[14] Nach Definition 5 ist ein intendiert erweiterbares System immer auch ein offenes System. Die einfache Konfigurierbarkeit ist auf der im ersten Modellierungsschritt eingenommenen Abstraktionsebene noch nicht relevant, da noch keine komplexen Schnittstellen oder Protokolle modelliert werden.

Integration von Objekten während der Laufzeit einer Anwendung interpretiert werden. Diese vielseitige Interpretierbarkeit erlaubt die in Unterabschnitt 4.2.6.1 bereits diskutierte Abstraktion von Lade- und Modulverknüpfungsmechanismen, welche zwar in allen Plugin-Systemen benötigt, aber in der Regel vom Betriebssystem oder der programmiersprachenspezifischen Laufzeitumgebung bereitgestellt werden.

Obwohl das *Netze-in-Netzen*-Paradigma auf dieser abstrakten Ebene zum Verständnis des Netzes in Abbildung 4.5 nicht unbedingt notwendig ist, ist das Modell schon so angelegt, dass es auf dem Paradigma der *Netze-in-Netzen* aufbauend verfeinert werden kann. Die fehlenden Komponenten wie Netzmarken, Kanalendpunkte und Systemgrenzen lassen sich unmittelbar ergänzen. Die Präzisierung erfolgt mit dem im nächsten Unterabschnitt folgenden Verfeinerungsschritt.

Das *Netze-in-Netzen*-Paradigma verdeutlicht auch, warum dieses Netz tatsächlich *intendierte* Erweiterbarkeit und nicht irgendeine andere Form offener Systeme verkörpert. Betrachtet man die Marken auf der zentralen Stelle wiederum als Netze, welche mittels synchroner Kanäle an Transitionen vom Typ „b“ mit dem dargestellten Netz kommunizieren können, so wird klar, dass eine Erweiterung den Zustand des Netzes nur beeinflussen kann, wenn ihre Integration im Netz im Bereich der erweiterbaren Kernfunktionalität in Form eines synchronen Kanals vorgesehen ist. Diese erweiterbare Kernfunktionalität beschränkt wiederum exakt die Möglichkeiten der Einflussnahme seitens vorhandener Erweiterungen, indem sie die Beschaffenheit des synchronen Kanals definiert.

Das abstrakte Modell basiert zwar in erster Linie auf dem Konzept der intendierten Erweiterbarkeit, repräsentiert aber weitere Eigenschaften. Die Transitionen „add“ und „remove“ passen auf die in Unterabschnitt 4.1.1 geforderte Schnittstelle offener Systeme zum Hinzufügen und Entfernen von Systemkomponenten. Die Konfiguration des Systems geschieht, wie in Unterabschnitt 4.1.2 gefordert, auf einfachste Weise durch dieselben beiden Transitionen, da zur Integration einer Erweiterung lediglich eine Marke auf die zentrale Stelle gelegt werden muss. Das Netz definiert zudem einen sehr einfachen Lebenszyklus (wie in Unterabschnitt 4.1.5 gefordert) für Erweiterungen, der ebenfalls durch die Transitionen „add“ und „remove“ beschrieben wird.

Auf der abstrakten Ebene des Modells bietet das Netz zudem maximale Nebenläufigkeit wie in Unterabschnitt 4.1.7 gefordert. Aus der Petrinetzsemantik ergibt sich unmittelbar, dass lediglich die Transitionen „add“ und „remove“ bezogen auf jede *individuelle* Erweiterung nicht nebenläufig zueinander und zur Transition „b“ sind. Abhängigkeiten zwischen verschiedenen Erweiterungen bestehen nicht. Die Testkante zur zentralen Stelle schränkt auch die Nebenläufigkeit der Transition „b“ zu sich selber nicht ein.

### 4.3.2 Präzisiertes Modell der Erweiterbarkeit

Das abstrakte Modell des Konzeptes der intendierten Erweiterbarkeit kommt noch ohne den Systembegriff aus, da lediglich „Funktionalität“ verwaltet wird. Die Abgrenzung von Systemen und Systemkomponenten wird aber erst durch eine konkrete Semantik unter dem Paradigma der *Netze-in-Netzen* mit Spezifikation von äußerem Systemnetz, inneren Objektnetzen und synchronen Kanälen zur Kommunikation ermöglicht.

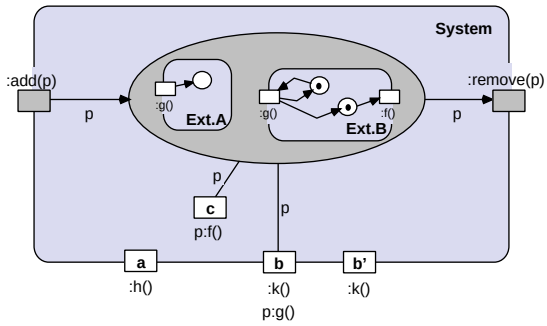



Abbildung 4.6: Modell der Erweiterungsbeziehung

Abbildung 4.6 präsentiert eine Verfeinerung des Netzes aus Abbildung 4.5, welche die soeben angesprochenen Details zum vorhergehenden Modellierungsschritt nachreicht. Jedes der eingefärbten abgerundeten Rechtecke repräsentiert jeweils ein Netz[15]. Zur Veranschaulichung des *Netze-in-Netzen*-Paradigmas sind Netzmarken grafisch innerhalb der sie enthaltenden Stelle dargestellt, in diesen Fällen stellt jeder hellblau eingefärbte Bereich zugleich eine Marke dar.

In Abbildung 4.6 sind drei Netze dargestellt: Das äußere Netz (in *Netze-in-Netzen*-Terminologie auch „*Systemnetz*“ genannt) repräsentiert das intendiert erweiterbare System, die beiden inneren Netze (in *Netze-in-Netzen*-Terminologie auch „*Objektnetze*“ genannt) repräsentieren exemplarisch zwei individuelle Erweiterungen „Ext.A“ und „Ext.B“, die momentan ins System integriert sind. Die Kanten und Stellen innerhalb der Objektnetze symbolisieren lediglich die mögliche Verschiedenheit konkreter Erweiterungen und haben keine konzeptionell relevante Bedeutung.

Die Grundstruktur des Netzes aus Abbildung 4.5 mit grau eingefärbten Verwaltungselementen und den ungefärbten Transitionen zur abstrakten Repräsentation konkreter Kernfunktionalität ist im Systemnetz in Abbildung 4.6 unmittelbar wieder zu erkennen. Ergänzt wurden zum einen Kantenanschriften mit der Variablen „p“ (für „Plugin“), um die Bindung von Objektnetzmarken im Kontext schaltender Transitionen zu ermöglichen. Zum anderen sind die abstrakten Transitionsbezeichnungen konkreten Kanalanschriften gewichen, so dass anhand der Up- und Downlinks sowie der exemplarisch vergebenen Kanalnamen die Synchronisationsmöglichkeiten zwischen den Netzen erkennbar werden.

Entsprechend der in Unterabschnitt 4.2.6.3 erläuterten Abstraktion von konkreter Funktionalität des Systems im Modell sind nur einige exemplarische Transitionen ohne ihren Kontext im Netz dargestellt. Das Kriterium für die Aufnahme dieser exemplarischen Transitionen in die Darstellung ist die Beschreibung von Systemschnittstellen

[15] Wie in Unterabschnitt 4.2.4 bereits besprochen, kommt die hier verwendete Modellierungstechnik mit einer Netzinstanz je Netzmuster aus, so dass der Begriff „Netz“ unterschiedslos zum Begriff der „Netzinstanz“ genutzt werden kann. Die grafische Darstellung mit eingefärbtem Hintergrund stellt dennoch eine deutliche Nähe zum Konzept der Netzinstanz her, wie es im Werkzeug Renew während der Ausführung einer mit Referenznetzen implementierten Anwendung dargestellt wird.

wie in Unterabschnitt 4.2.5.2 erläutert: Die vorhandenen Transitionen symbolisieren stellvertretend jene Teile des Netzes, die zur Definition der inneren oder der äußeren Schnittstelle(n) des Systems beitragen. Die Transitionen „a“, „b“ und „b′“ symbolisieren Funktionalität, welche über die Außengrenze des Systems hinweg arbeitet oder genutzt werden kann. Dagegen symbolisieren die Transitionen „b“ und „c“ Funktionalität, welche über die innere Schnittstelle mit verwalteten Netzmarken in Verbindung steht (die Funktionalität der Transition „b“ ist also über beide Schnittstellen erreichbar). Rein interne Funktionalität ohne Bezug zu Erweiterungen kann zwar im System vorhanden sein, ist aber nicht dargestellt, da sie weder zur äußeren noch zur inneren Schnittstellendefinition beiträgt.

Die innere Schnittstelle des Systemnetzes wird durch die Downlinks der Transitionen „b“, „b′“ und „c“ definiert, sie umfassen die Kanäle `:f()` und `:g()`. Genau diese beiden Kanäle bilden auch die äußere Schnittstelle des Netzes „Ext.B“, während das Netz „Ext.A“ nur den Kanal `:g()` in Form eines Uplinks unterstützt. Die Entscheidung, alle im Kontext einer erweiterungsverwaltenden Stelle definierten Kanäle zu einer *gemeinsamen* Schnittstellendefinition zusammenzufassen, wurde bereits in Unterabschnitt 4.2.5.3 diskutiert.

In jedem Fall kann die innere Schnittstelle des Systemnetzes als Erweiterungsspezifikation im Sinne von Definition 7 betrachtet werden. Ausschließlich solche Objektnetze, die zu dieser Schnittstelle passen, können erfolgreich in das System integriert werden. Die im Systemnetz vorgegebenen Kanalsignaturen bestimmen Möglichkeiten und Beschränkungen der Funktionalität potentieller Erweiterungen.

Die Kanäle verfügen im Beispiel nicht über Parameter, da die Darstellung sonst zu textlastig wäre. Aufgrund der in Unterabschnitt 4.2.5 bereits diskutierten Bidirektionalität der Kanäle würde die Angabe konkreter Parameter ohnehin kaum zur weiteren Präzisierung des Modells beitragen, da intendierte Daten- und Kontrollflussrichtung erst im Zusammenhang eines weiter verfeinerten Netzes mit Kanten und Stellen zur Variablenbindung erkennbar würden.

Die Duplikation der Transition „b“ in eine Variante mit und ohne Kommunikation mit Erweiterungen wurde bereits im vorigen Unterabschnitt erläutert. Eine derartige Duplikation ist natürlich auch für die Transition „c“ denkbar, falls ihre Funktionalität auch ohne das Vorhandensein von Erweiterungen verfügbar sein soll. Eine entsprechende Transition „c′“ hätte aber weder Kontakt zur inneren noch zur äußeren Schnittstelle des Systemnetzes und ist daher nicht dargestellt.

Mit Einführung der äußeren Systemgrenze besteht die Möglichkeit, die im Systemnetz angebotene Kernfunktionalität danach zu unterteilen, ob sie ausschließlich intern genutzt oder auch extern angeboten wird, also Teil einer äußeren Schnittstelle des Systems ist.[16] [17] In Abbildung 4.6 liegen z. B. die Transitionen „a“, „b“ und „b′“ auf der Systemgrenze und bieten Uplinks zur Nutzung von außerhalb an, während die Tran-

[16] Gemäß der Semantik der Referenznetze können die Uplinks der Transitionen auf der äußeren Systemgrenze auch lokal aufgerufen werden, da – wie in diversen objektorientierten Programmiersprachen – das reservierte Wort `this` zur Selbstreferenzierung in Downlinks zur Verfügung steht. Damit ist sämtliche im Netz auf der Systemgrenze dargestellte Funktionalität nicht nur von extern, sondern auch innerhalb des Netzes nutzbar.

[17] Die Entscheidung, die Menge aller Transitionen auf der äußeren Systemgrenze (also alle jene mit Uplink) z. B. nach fachlichen Aspekten in mehrere Schnittstellen zu unterteilen, wurde bereits in Unterabschnitt 4.2.5.3 diskutiert.

sition „c“ nicht direkt mit der Umgebung des Systems kommunizieren kann, sondern nur eine interne Wirkung auf den Zustand des Systemnetzes entfaltet. Die Transitionen „add“ und „remove“ stellen ebenfalls eine äußere Schnittstelle dar, da sie die in einem offenen System notwendige Schnittstelle zum Hinzufügen und Entfernen von Systemkomponenten definieren.

Die bereits im vorigen Unterabschnitt diskutierten Eigenschaften des Netzmodells gelten weiterhin: Es modelliert ein offenes und intendiert erweiterbares System mit einem einfachen Lebenszyklus für Erweiterungen. Das System in Abbildung 4.6 ist immer noch einfach konfigurierbar, da die Kooperation zwischen System- und Objektnetz lediglich durch die Transitionen „add“ und „remove“ gesteuert wird. Durch die Verwendung von Testkanten ist wie im Grundmodell sichergestellt, dass sämtliche Kommunikation zwischen System- und Objektnetzen nebenläufig stattfinden kann, solange die Objektnetze als Marke im Systemnetz enthalten sind.

In Ergänzung zum Grundmodell ist mit den Systemgrenzen ein Konzept zur Spezifikation der Kapselung von Systemkomponenten, wie in Unterabschnitt 4.1.4 gefordert, in das Modell eingeführt worden. Die Präzisierung der exemplarischen Transitionen und Anschriften verdeutlicht zudem die Möglichkeiten der Referenznetze als Modellierungssprache, der in Unterabschnitt 4.1.8 formulierten Anforderung einer intuitiven Modellierung mit präziser Semantik gerecht zu werden.

### 4.3.3 Unabhängige Erweiterbarkeit

Es ist mit dem bis jetzt erarbeiteten Modell ohne Weiteres möglich, unabhängige Erweiterbarkeit (*independent extensibility*) in mehreren Dimensionen, wie sie von Szyperski in [Szy96] definiert wurde, zu realisieren. Um mehrere, fachlich nicht unmittelbar zusammenhängende Erweiterungsspezifikationen im Systemnetz vorzusehen, können mehrere Erweiterungsverwaltungssegmente ins Netz integriert werden. Abbildung 4.7 illustriert dieses Vorgehen exemplarisch für zwei Dimensionen. Es entspricht der bereits in Unterabschnitt 4.2.5.3 besprochenen Idee, jede Stelle auf genau eine innere Schnittstelle festzulegen und mehrere innere Schnittstellen über mehrere Stellen zu verteilen.

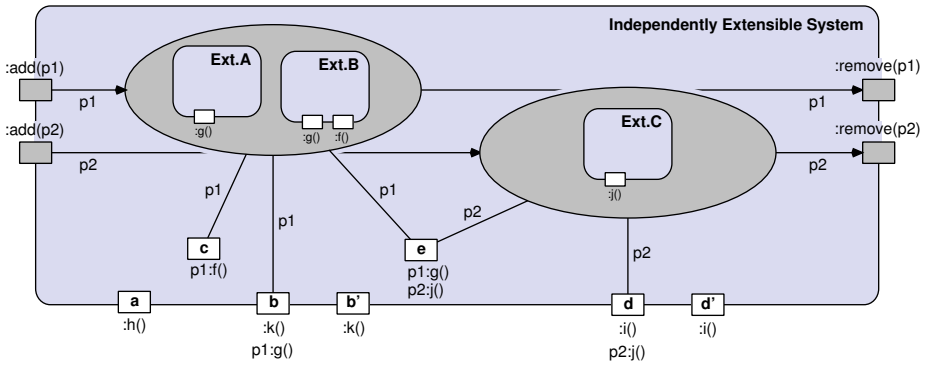



Abbildung 4.7: Mehrere Dimensionen unabhängiger Erweiterbarkeit

Jede der Erweiterbarkeitsdimensionen ist durch eine eigene Verwaltungsstelle mit zugehörigen „add“- und „remove“-Transitionen repräsentiert. Die mehrfach angebotenen Uplinks `:add(pi)` bzw. `:remove(pi)` stehen allerdings nach Java-Referenznetzsemantik nichtdeterministisch im Konflikt. Dieser Nichtdeterminismus lässt sich durch die in Unterabschnitt 4.2.5.4 vorgeschlagene Typisierung von Variablen, Kanten und Stellen auflösen. Jede „add“-Transition akzeptiert dann nur den passenden Erweiterungstyp als Kanalparameter, ebenso sind die Verwaltungsstellen und alle benachbarten Kanten auf einen Erweiterungstyp festgelegt.

Die exemplarisch als Erweiterungen integrierten Objektnetze sind in dieser Abbildung nur noch mit ihren äußeren Schnittstellen dargestellt, um das Modell übersichtlich zu halten. Neben die bereits aus Abbildung 4.6 bekannte Erweiterungsspezifikation mit den Kanälen `:f()` und `:g()` ist nun eine weitere Erweiterungsspezifikation mit dem Kanal `:j()` hinzugestoßen. Das Systemnetz kann auf Erweiterungen, die zu einer der beiden Spezifikationen passen, sowohl im Rahmen extern angebotener als auch intern existierender Kernfunktionalität in beliebiger Kombination zurückgreifen. In Abbildung 4.7 sind nur einige der möglichen Kombinationen dargestellt. Beispielsweise integriert die Transition „e“ Erweiterungsfunktionalität aus beiden Erweiterungsdimensionen.

Die Integration mehrerer unabhängiger Erweiterbarkeitsdimensionen erfordert in Petrinetz-typischer Weise lediglich die Addition der individuellen Erweiterungsverwaltungselemente. Da diese Weise des Ausbaus der Modelle grundsätzlich immer möglich ist, wird auf diese Möglichkeit in den folgenden Modellierungsschritten nicht mehr gesondert hingewiesen.

### 4.3.4 Rekursion in der Erweiterungsbeziehung

Das präzisierte Modell des Konzeptes der intendierten Erweiterbarkeit beschreibt ein System, das genau „auf einer Ebene erweiterbar“ ist. Auch die eben vorgestellte Möglichkeit der Erweiterung in mehreren unabhängigen Dimensionen behandelt noch nicht den Aspekt, dass in vielen Plugin-Systemen Systemkomponenten ins System integriert werden können, die wiederum erweiterbar, also rekursiv erweiterbare Erweiterungen, sind.

Das bisher entwickelte Modell ist bereits in der Lage, rekursive Erweiterbarkeit auszudrücken, wenn eine Erweiterung intern wiederum eine Erweiterungsverwaltung implementiert, also eine Systemkomponente wiederum als erweiterbares System auftritt. In Abbildung 4.8 ist das Netz „Extensible Extension B“ eine solche erweiterbare Systemkomponente, da es sowohl in der Rolle eines Objektnetzes als auch der eines Systemnetzes auftritt. Die innere Schnittstelle des Netzes gibt den Kanal `:m()` als Teil der Erweiterungsspezifikation vor, zu der im Beispiel das Netz „Ext.C“ passt. Die äußere Schnittstelle des Netzes „Extensible Extension B“ passt mit den Kanälen `:g()` und `:f()` genau zur Erweiterungsspezifikation des äußeren Systemnetzes.

Es ergibt sich eine Verkettung der Erweiterungsbeziehungen: Wird die im Kanal `:k()` angebotene Funktionalität des äußeren erweiterbaren Systems verlangt, so kann dieses System auf erweiternde Funktionalität einer Systemkomponente zurückgreifen, welche über den Kanal `:g()` integriert wird. Handelt es sich dabei um die Erweiterung „B“, so greift diese in der exemplarisch modellierten Systemkonfiguration wiederum

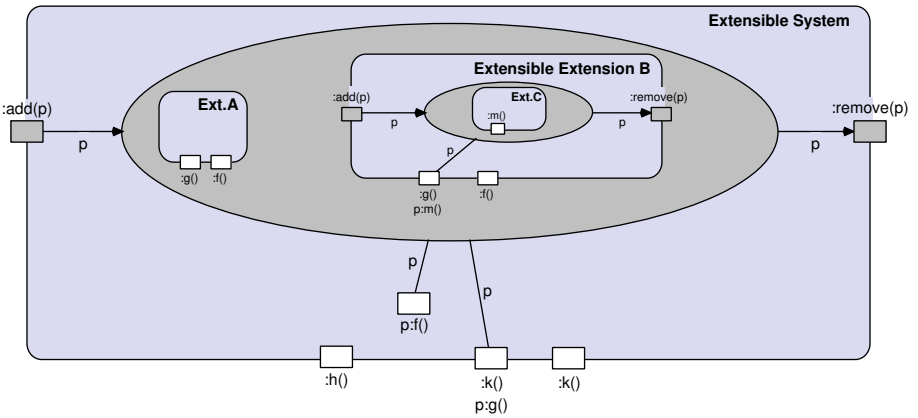



Abbildung 4.8: Erstes Modell der rekursiven Erweiterungsbeziehung

auf erweiternde Funktionalität der Systemkomponente „Ext.C“ über den Kanal `:m()` zurück.

Das Netz „Extensible Extension B“ verfügt allerdings über eine weitere äußere Schnittstelle, welche die Kanäle `:add(p)` und `:remove(p)` umfasst. Gemäß der Semantik synchroner Kanäle in Referenznetzen können (neben dem anbietenden Netz selber) ausschließlich jene Netze, welche eine Referenz auf das Kanal-Uplink-anbietende Netz zur Verfügung haben, diese Kanäle nutzen. In der vorliegenden Modellierung hat nur ein einziges Netz diese Kommunikationsmöglichkeit, nämlich das äußere erweiterbare System. Die Schnittstelle zum Hinzufügen und Entfernen von Erweiterungen des inneren erweiterbaren Systems ist somit von außerhalb des Gesamtsystems nicht zugänglich, so dass ein Anwender keine Möglichkeit hat, Erweiterungen wie z. B. „Ext.C“ zu konfigurieren. Um die Konfigurationsschnittstelle des inneren offenen Systems allgemein verfügbar zu machen, muss das äußere erweiterbare System die Existenz von erweiterbaren Erweiterungen *vorsehen*.

Ein Durchleiten der Konfigurationsschnittstelle des inneren offenen Systems nach außen kann in der gleichen Weise mit verketteten Kanälen geschehen, wie es bereits beim zuvor beobachteten Verketten von erweiternder Funktionalität geschieht. In Abbildung 4.9 ist das äußere erweiterbare System um eine weitere äußere Schnittstelle mit den zwei Kanälen `:subadd(q)` und `:subremove(q)` ergänzt worden.[18] Die „Funktionalität“ hinter dieser Schnittstelle umfasst im Wesentlichen die vom inneren System bereitgestellte Funktionalität zum Hinzufügen und Entfernen von Systemkomponenten. Das äußere erweiterbare System integriert also die Erweiterbarkeit des inneren erweiterbaren Systems in seine eigene Funktionalität. Die Erweiterung von Subsystemen ist natürlich nur möglich, falls solche Systeme im System integriert sind. Die

[18] Der Variablenname `q` trägt keine besondere Semantik zur Unterscheidung von der sonst verwendeten Variablen `p`. Im Kontext der Transitionen mit den Uplinks `:subadd(q)` und `:subremove(q)` ist lediglich die Variable `p` bereits an die zu konfigurierende Erweiterung gebunden, so dass ein anderer Variablenname notwendig wird.

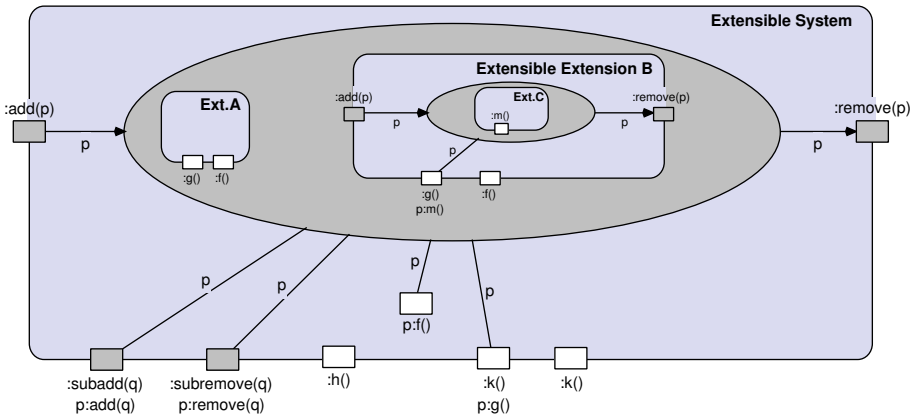



Abbildung 4.9: Kanalkette zur Konfiguration innerer Erweiterungen

Integration trägt somit die typischen Merkmale der intendierten Erweiterbarkeit, so dass von einer im äußeren System *vorgesehenen* Funktionalitätserweiterung gesprochen werden kann.

Es ist zu beachten, dass das Bereitstellen der inneren Konfigurationsschnittstelle nicht zwingend als bloße, ungefilterte Weiterleitung im äußeren System realisiert werden muss. Vielmehr sieht die Betrachtungsweise der inneren Konfigurationsmöglichkeit als *intendierte Erweiterbarkeit* der Funktionalität des äußeren Systems explizit vor, dass das äußere System auf die Kommunikation Einfluss nimmt, um eine Erweiterung nur im Rahmen der vom äußeren System gesetzten Vorgaben zuzulassen.

Auf die gleiche Weise lässt sich auch eine Rekursion über mehrere Ebenen realisieren. Beispielsweise könnte das Systemnetz wieder als Erweiterung innerhalb eines weiteren erweiterbaren Systems integriert werden, sofern die Kanäle der äußeren Schnittstelle `:h()` und `:k()` auf dessen Erweiterungsspezifikation passen. Oder die Erweiterung der Erweiterung ist wiederum erweiterbar – wie in Abbildung 4.10 exemplarisch durch das Netz „E.E.C“ (kurz für „Extensible Extension C“) visualisiert.

In diesem Beispiel ist nun auch im Netz „Extensible Extension B“ die Erweiterbarkeit innerer Erweiterungen Teil der intendiert erweiterbaren Funktionalität der Systemkomponente selber. Die Integration wird wie im vorigen Beispiel erreicht, indem die Konfigurationskanäle `:add(p)` und `:remove(p)` zum Bestandteil der inneren Erweiterungsschnittstelle der Systemkomponente „B“ erklärt werden. Die Konfiguration innerer Systemkomponenten wird in die äußere Schnittstelle des Netzes in Form der Kanäle `:subadd(q)` und `:subremove(q)` integriert.

Die „Extensible Extension B“ benötigt nun neben den bereits zuvor besprochenen Kanälen `:g()`, `:f()` `:add(p)` zur vollständigen Integration in ein äußeres Systemnetz eine Erweiterungsspezifikation, welche die Kanäle `:subadd(q)` und `:subremove(q)` mit einschließt. Das äußere erweiterbare System ist daher ein weiteres Mal anzupassen, um auch diese Erweiterung seiner Funktionalität über seine äußere Schnittstelle

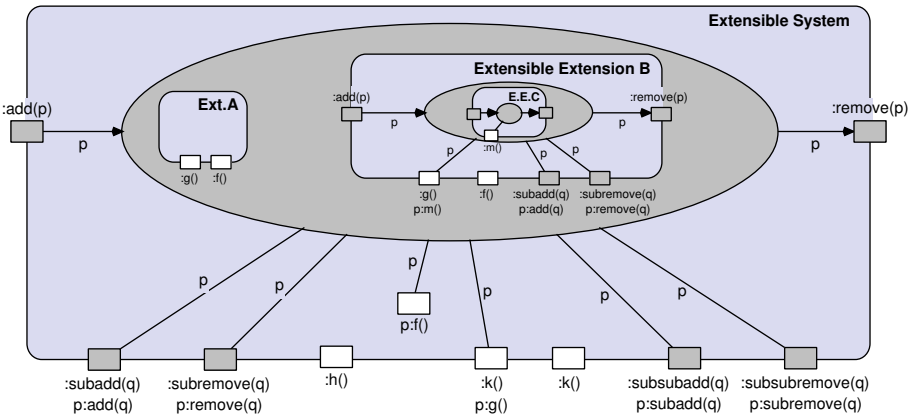



Abbildung 4.10: Rekursive Konfiguration über drei Ebenen

zugänglich zu machen. Die Anpassung geschieht mittels des in Abbildung 4.10 rechts unten dargestellten Transitionspaares.

Wie inzwischen klar werden dürfte, entstünde bei naiver Anwendung dieses Schemas der Einrichtung einer zusätzlichen Kanalkette für jede weitere Ebene offener Systeme im Inneren ein neues Transitionspaar auf allen äußeren Systemschnittstellen bis einschließlich der äußersten Systemgrenze. Neben diesen hohen Aufwand träte noch das Problem, dass dieses Schema nur eine feste Anzahl an Rekursionsebenen zuließe, da nur eine begrenzte Zahl an Kanalpaaren im Voraus im äußeren erweiterbaren System vorgesehen werden kann.

Unter Ausnutzung der dynamischen Bindung synchroner Kanäle, die relevant wird, sobald mehrere Transitionen Uplinks mit passender Kanalsignatur anbieten, kann die Rekursionstiefe beliebig gehalten und der Aufwand auf exakt zwei Transitionspaare auf jeder Rekursionsebene erweiterbar erweiterbarer Systeme beschränkt werden. In Abbildung 4.11 wird diese Technik eingesetzt. Sowohl die Konfigurationsschnittstelle des äußersten erweiterbaren Systems als auch die über mehrere Ebenen verketteten Konfigurationskanäle aller rekursiv eingeschlossenen offenen Systemkomponenten werden unter dem Uplink `:add(q)` bzw. `:remove(q)` angeboten. Der dabei entstehende Nichtdeterminismus kann wiederum auf Basis der in Unterabschnitt 4.2.5.4 vorgeschlagenen Technik durch die Typisierung der als Kanalparameter verwendeten Variablen aufgelöst werden. Während die allgemeine Kanalsignatur der rekursiven Kanäle `:add(q)` und `:remove(q)` beliebige Plugins abdeckt, sind die speziellen Uplinks `:add(p)` und `:remove(p)` eines jeden erweiterbaren Systems auf genau den jeweils passenden Erweiterungstypen hin eingeschränkt.

Falls alle erweiterbaren Systemkomponenten die Verkettung der Kanäle `:add(q)` und `:remove(q)` in ihre äußeren und inneren Schnittstellen (zusätzlich zur eigenen Konfigurationsschnittstelle unter denselben Kanalsignaturen) aufnehmen, ist eine beliebig tiefe Schachtelung erweiterbarer Systeme möglich. Das in Abbildung 4.11 unten rechts dargestellte Transitionspaar leitet die Konfigurationskommunikation mit unver-

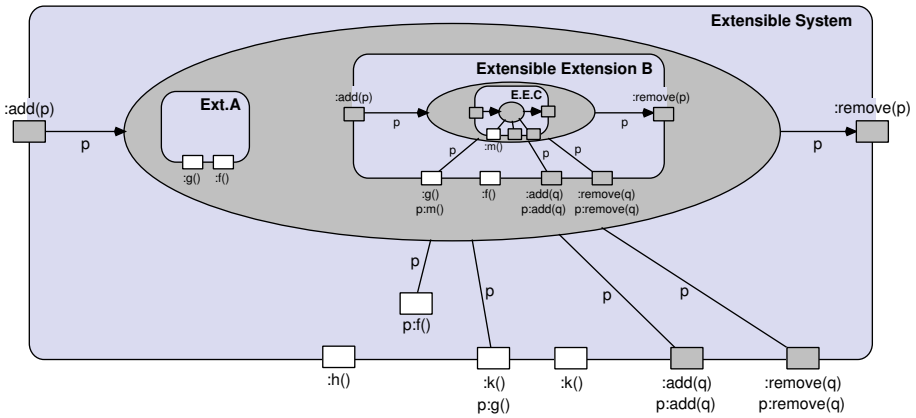



Abbildung 4.11: Vereinheitlichte rekursive Konfiguration

änderter Signatur von außen nach innen (bzw. umgekehrt, da bidirektional) weiter. Erst auf der Ebene des zu konfigurierenden offenen Systems übernimmt die am linken bzw. rechten Rand der passenden erweiterbaren Systemkomponente dargestellte Transition mit `:add(p)`- bzw. `:remove(p)`-Kanal die Kommunikation und führt die eigentliche Konfiguration durch. Damit entfällt auch die nach außen wahrnehmbare Sonderbehandlung der Konfiguration von Erweiterungen auf der obersten Ebene (`:add(p)`) im Vergleich zur Konfiguration von Erweiterungen auf allen anderen Ebenen (`:subadd(q)`), da sie nun über dieselbe Kanalsignatur angeboten wird.

Das um rekursive Erweiterbarkeit ergänzte Modell des Plugin-Konzepts dieses Modellierungsschrittes erfüllt weiterhin alle Anforderungen, die in den vorangegangenen Schritten bereits erfüllt sind, da keine grundlegend neue Technik eingeführt wird. Der Modellierungsschritt nutzt die bereits aus den vorigen Modellierungsschritten bekannte Technik zur Integration von Erweiterungen, um das erweiterbare System um eine erweiterbare Konfigurationsschnittstelle zu ergänzen. Das offene, intendiert erweiterbare System ist weiterhin einfach konfigurierbar, da zur Integration einer Erweiterung lediglich der `:add()`-Kanal aktiviert werden muss. Die Systemkomponenten erkennen selbständig, zu welchem der vorhandenen erweiterbaren Subsysteme die Erweiterung passt. Das Entfernen von Systemkomponenten geschieht gleichermaßen einfach. Da im Vergleich zu den vorigen Modellierungsschritten den Netzen lediglich Testkanten hinzugefügt werden, bleibt die Nebenläufigkeit im System weiterhin maximal.

Das streng hierarchisch organisierte Modell erfüllt darüber hinaus zum Teil die in Unterabschnitt 4.1.6 erarbeitete Anforderung mehrerer Verwaltungsebenen. Die geschachtelten erweiterbaren Systeme werden anhand fachlicher Kriterien entwickelt, so dass jedes erweiterbare System seine eigene(n) Erweiterungsspezifikation(en) definiert. Integration und Verwaltung der jeweils passenden Erweiterungen finden in derselben Systemkomponente statt, welche die Erweiterungsspezifikation definiert. Jede erweiterbare Systemkomponente kann daher ihren jeweiligen Erweiterungen einen individuellen, fachlich oder technisch motivierten Lebenszyklus vorgeben. Eine (einfa-

che) Koordination über Verwaltungsebenen hinweg ist in Form der vereinheitlichten Kanalketten zur Konfiguration von Erweiterungen vorhanden.

Allerdings findet zwischen den erweiterbaren Systemen keine über die Verkettung der Konfigurationskanäle hinausgehende Koordination statt. Jedes erweiterbare Subsystem muss neben der fachlichen auch die technische Integration von Erweiterungen alleine bewältigen. Zudem muss jede erweiterbare Systemkomponente explizit Infrastruktur zur Konfiguration innenliegender Systemkomponenten bereithalten, die zwar – wie in diesem Modellierungsschritt verdeutlicht – zur eigenen Funktionalität gehören soll, aber eigentlich nichts mit den fachlichen Aufgaben der Systemkomponente gemein hat.

### 4.3.5 Nicht-hierarchische Strukturen

Das in den bisherigen Modellierungsschritten entwickelte Modell eines Plugin-Systems unterstützt die Konzepte der intendierten Erweiterbarkeit, der unabhängigen Erweiterbarkeit in mehreren Dimensionen und der rekursiven Erweiterbarkeit in beliebiger Schachtelungstiefe. Es ist ein streng hierarchisches Modell, in welchem jede Systemkomponente an genau einer Position in der Hierarchie integriert ist.

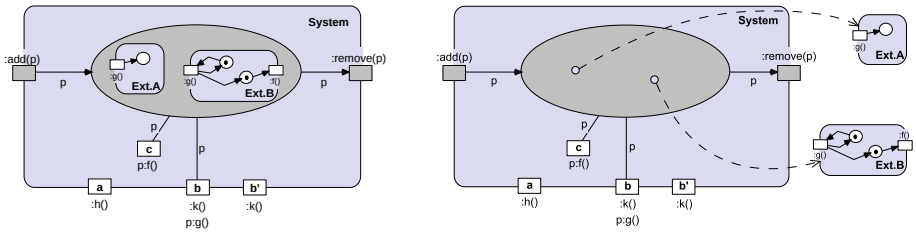
Die Annahme, dass jedes Plugin isoliert an genau einer Erweiterungsschnittstelle der Softwarearchitektur integriert werden kann, geht jedoch an der Realität vorbei. Wie bereits in Abschnitt 4.1.9.1 ab S. 213 diskutiert, sollen Plugins eine inhaltlich in sich abgeschlossene Funktionalität bieten, welche häufig erst durch die Nutzung *mehrerer* fachlich oder technisch spezialisierter Erweiterungsschnittstellen sinnvoll in ein System integriert werden kann.

Das bisherige Modell sieht in seiner streng hierarchischen Interpretation keine Möglichkeit vor, *eine* Erweiterung gleichzeitig in *mehrere* erweiterbare Systemkomponenten zu integrieren. Nicht einmal die Nutzung mehrerer Erweiterungsspezifikationen innerhalb derselben Komponente ist möglich, sobald die in Unterabschnitt 4.3.3 vorgestellte Technik zur Modellierung unabhängiger Erweiterbarkeit eingesetzt wird.

Wollte man das Modell weiterhin streng hierarchisch interpretieren, stünden verschiedene Alternativen zur Auswahl, um die Integration eines Plugins an mehreren Erweiterungsschnittstellen zu ermöglichen. Nötig würde die Einrichtung von Kommunikationswegen entlang der hierarchischen Beziehungen, so dass ein an einer Position in der Hierarchie verankertes Plugin entlang der Kommunikationswege baumauf- und -abwärts mit Erweiterungsschnittstellen an anderen Positionen kommunizieren könnte. Sämtliche erweiterbaren Systemkomponenten müssten ihre inneren und äußeren Schnittstellen um die nötige Infrastruktur ergänzen (analog der schon in der rekursiven Erweiterbarkeit eingeführten Kanalketten). Um die Kommunikationswege kurz zu halten und den Infrastrukturaufwand in erweiterbaren Systemkomponenten zu vermeiden, böte sich die Alternative an, die Hierarchie zu einem flachen System kollabieren zu lassen, in welchem ein einziges äußeres erweiterbares System zwischen sämtlichen Systemkomponenten in ihren Rollen als erweiterbare Systeme oder als Erweiterungen vermittelte. Damit entstünde eine grundsätzlich andere Systemarchitektur, die der einer dienstorientierten Architektur, wie später in Unterabschnitt 4.3.8.1 beschrieben, stark ähnelte. Sowohl unter Beibehaltung der Hierarchie als auch ohne sie hätte die Systemarchitektur kaum noch etwas mit dem grundlegenden Modell der intendierten

Erweiterbarkeit gemein, da die naheliegende Modellierung des Im-erweiterten-System-enthalten-Seins für viele oder alle Erweiterungen nicht mehr gelten würde.

Die elegantere Lösung ist, die Vorstellung eines streng hierarchisch organisierten Systems aufzuweichen. Die Realität existierender Plugin-Systeme verlangt nach einem Beziehungsgeflecht, das einem azyklischen Graphen entspricht. Das Netzmodell aus dem vorigen Unterabschnitt kann dieser Vorstellung gerecht werden, wenn es – wie bereits in Unterabschnitt 4.2.4 besprochen – unter Referenzsemantik interpretiert wird. Diese Änderung der Interpretation kommt ohne jegliche Modifikation an Struktur oder Anschriften aus, da jede individuelle Erweiterungsbeziehung auf die gleiche Weise wie bisher als Enthaltensein einer Netzmarke betrachtet werden kann.




(a) Das Netzsystem aus Abbildung 4.6 (b) Dasselbe Netzsystem in Referenzdarstellung

Abbildung 4.12: Gegenüberstellung der geschachtelten und der Referenzdarstellung

Zur Visualisierung können statt der grafischen Schachtelung der Netze der Referenzsemantik entsprechend Pfeile von Referenzmarken zur Netzinstanz eingesetzt werden. Abbildung 4.12(b) illustriert diese Technik am präzisierten Grundmodell aus Abbildung 4.6. Da unter der Darstellung von Referenzen in Form von Pfeilen in einigen Fällen die Übersichtlichkeit des Bildes leidet, wird im Folgenden eine gemischte Darstellung verwendet. Falls die Referenz das logische oder physische Enthaltensein einer Systemkomponente in einer anderen modelliert, kann alternativ zur Referenzdarstellung mittels Pfeil auch die bereits bewährte geschachtelte Darstellung gewählt werden. Netzinstanzen, die in mehreren Systemkomponenten *enthalten* sein sollen, können – ohne einen semantischen Unterschied darstellen zu wollen – an einer Position eingeschachtelt und gleichzeitig von anderen Positionen referenziert dargestellt werden.

In Abbildung 4.13 wird exemplarisch von dieser Technik Gebrauch gemacht. Im Beispiel ist „Ext. D“ sowohl in „Extensible Extension A“ als auch „Extensible Extension B“ integriert. Zu diesem Zweck bietet „Ext. D“ zwei äußere Schnittstellen an: Die eine mit dem Kanal `:m()` passt zur inneren Erweiterungsschnittstelle der Systemkomponente „A“, die andere mit dem Kanal `:n()` zur Erweiterungsspezifikation der Systemkomponente „B“.

Das Zulassen mehrfach geschachtelter Erweiterungsbeziehungen flexibilisiert die Architektur des Plugin-Systems. Die meisten vom Modell zuvor erfüllten Anforderungen an ein Plugin-Konzept bleiben erhalten, da sich die Struktur und die Anschriften der einzelnen Netze nicht verändert haben. Allerdings verkompliziert die Möglichkeit, ein Plugin an verschiedenen Positionen als Erweiterung zu integrieren, die Konfiguration

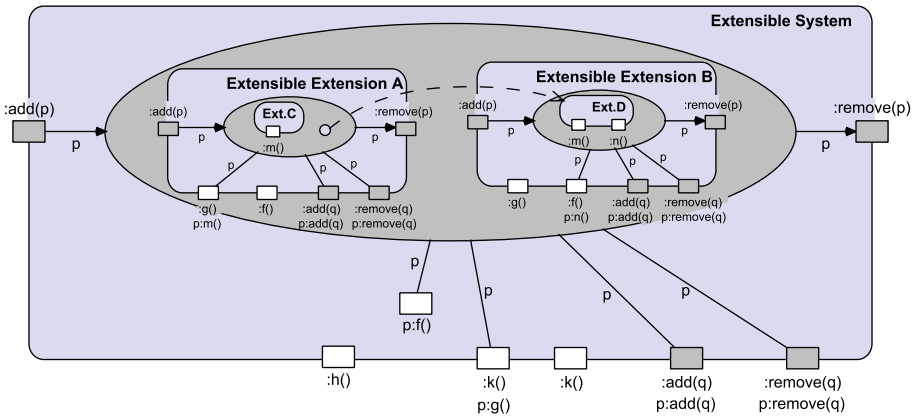



Abbildung 4.13: Beispiel für eine nicht-hierarchische rekursive Erweiterungsbeziehung

des Systems. Es ist für jede Erweiterungsbeziehung, die das hinzuzufügende Plugin eingehen soll, ein eigener Aufruf der `:add(p)`- Kanalkette erforderlich. Das Entfernen des Plugins aus der Systemkonfiguration benötigt analog viele Aufrufe der `:remove(p)`-Kanalkette. Somit sind nun das Knüpfen oder Lösen aller Beziehungen mehrschrittige Prozesse, die ohne Wissen über die konkrete Menge der Beziehungen eines Plugins nicht korrekt durchführbar sind.

### 4.3.6 Zentralisierung der Erweiterungsverwaltung

Um die Konfiguration eines rekursiv erweiterbaren Systems zu ermöglichen, wurde in Unterabschnitt 4.3.4 jeder erweiterbaren Systemkomponente mit den Konfigurationskanalketten ein Stück Verwaltungsfunktionalität aufgezwungen, welche nicht zur fachlichen Kernfunktionalität der Systemkomponenten gehört. Nachdem im vorigen Unterabschnitt durch einen Interpretationswechsel die Möglichkeit geschaffen wurde, Systemkomponenten an mehreren Positionen in der Systemstruktur zu integrieren, kann nun eine dedizierte erweiterbare Systemkomponente mit genau dieser Funktionalität der erweiterbaren Konfiguration eingeführt werden. Diese Systemkomponente entbindet alle anderen Systemkomponenten von der Notwendigkeit, selber für die technische Infrastruktur zur Verwaltung von Plugins zu sorgen, und soll daher Pluginverwaltungssystem (*Plugin Management System, PMS*) genannt werden.

In Abbildung 4.14 ist das „PMS“ auf der linken Seite dargestellt. Alle Plugins auf der rechten Seite sind nicht nur Erweiterungen ihres fachlich passenden erweiterbaren Systems, sondern auch Erweiterungen des Pluginverwaltungssystems PMS. Diese Separation erspart die aufwendige Implementierung der Verwaltungskanalketten in *allen* anderen Plugins. Stattdessen verfügt das PMS über eine Referenz zu jedem Plugin und bietet mittels dieser Referenzen eine einstufige Kanalkette zu den Konfigurationsschnittstellen der erweiterbaren Systeme. Da die Konfiguration von Systemkomponenten als „Erweiterungen“ des PMS eine deutlich von der Konfiguration als Erweite-

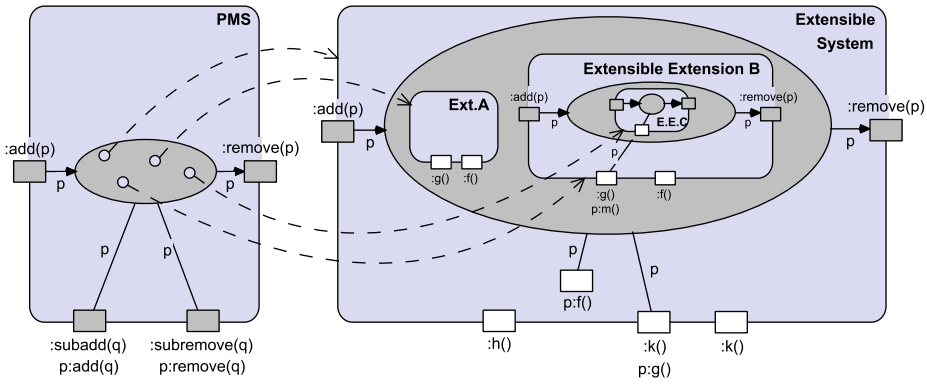



Abbildung 4.14: Plugin-Management-System (PMS) mit einstufiger Kanalkette

rungen anderer erweiterbarer Systemkomponenten zu unterscheidende Funktionalität ist, tragen die Konfigurationskanäle der Kanalkette nun wieder die unterscheidbaren Namen `:subadd(q)` bzw. `:subremove(q)`.

Da die Konfigurationskanalketten die einzige erweiterbare Funktionalität des PMS sind, handelt es sich hierbei um dessen Kernfunktionalität. Die *fachliche* Motivation der Erweiterungsspezifikation des PMS ist also eine *technische*, nämlich die Konfiguration erweiterbarer Systemkomponenten. Nach außen präsentiert sich ein PMS mit einer Konfigurationsschnittstelle wie jede andere erweiterbare Systemkomponente auch. Daher ist es vorstellbar, in einer Systemarchitektur mehrere Plugin-Verwaltungssystemkomponenten vorzusehen, von denen jede für einen Teilbereich der Architektur verantwortlich zeichnet. Diese Sichtweise erlaubt eine einfache Lösung für das in Unterabschnitt 3.3.6 angesprochene Problem der Integration unterschiedlicher Pluginverwaltungssysteme, da ein fremdes PMS nach einfacher Anpassung der Konfigurationskanalsignaturen in die Verwaltung einer bestehenden Plugin-Architektur integriert werden kann.

Die Darstellung von PMS und der eigentlichen Struktur erweiterbarer Systeme in Abbildung 4.14 als zwei nebeneinander stehende Systeme lässt bewusst die Frage offen, welches der beiden Systeme die eigentliche Grenze zur Außenwelt darstellt. Es ist auch denkbar, dass die äußerste Systemgrenze durch Vereinigung der beiden Systemgrenzen entsteht. Für die weitere Modellierung soll diese Frage offen bleiben, damit das Modell sämtliche Varianten des Plugin-Konzepts abdecken kann. Es wird also weiterhin wie in Abbildung 4.14 das PMS als separates System *neben* der eigentlichen Systemarchitektur dargestellt.

Im aktuellen Modellierungsschritt hat das Modell – wie in Unterabschnitt 4.1.6 gefordert – eine Separation der technischen von der fachlichen Verwaltung von Erweiterungen erfahren. Jede Systemkomponente kann sich in ihrer Konfigurationsschnittstelle auf die fachliche Integration von Erweiterungen beschränken, da das PMS die systemweit einheitliche technische Integration von Systemkomponenten übernimmt. Alle im vorhergehenden Modellierungsschritt erreichten Eigenschaften gelten weiterhin, so

dass das Modell die Anforderungen an ein offenes, intendiert erweiterbares, maximal nebenläufiges System erfüllt. Allerdings ist das am Ende des vorigen Unterabschnitts angesprochene Problem weiterhin vorhanden, dass Plugins, welche mehrere Systemkomponenten erweitern, mehrere Konfigurationsschritte erfordern. Das Problem tritt sogar regelmäßig auf, da nun jedes Plugin in mindestens zwei Erweiterungsbeziehungen (einer fachlichen und einer zur Konfiguration via PMS) als Erweiterung agiert.

### 4.3.7 Konfiguration in einem Schritt

Wie bereits nach Abschluss der vorigen beiden Modellierungsschritte angesprochen, hat das Modell durch die Möglichkeit bzw. Verpflichtung, Plugins unter mehreren erweiterbaren Systemen als Erweiterung zu verankern, die in Unterabschnitt 4.1.2 geforderte Eigenschaft der einfachen Konfigurierbarkeit eingebüßt. Um den Konfigurationsaufwand für den Anwender wieder auf das ursprüngliche Maß zu reduzieren, ist es notwendig, die Aktionen `:add(p)` und `:remove(p)` so mit weiteren `:subadd(p)`- bzw. `:subremove(p)`-Aktionen zu koppeln, dass sie alle durch eine einzige Aktion des Anwenders angestoßen werden können. Es gibt grundsätzlich zwei Möglichkeiten, dieses Ziel zu erreichen: Entweder übernimmt das PMS die Aufgabe, sämtliche Konfigurationsschritte zu koordinieren, oder das zu konfigurierende Plugin kümmert sich selber darum. In diesem Unterabschnitt soll erstere Variante vorgestellt werden, da für letztere noch einige Voraussetzungen fehlen, die erst im nachfolgenden Unterabschnitt geschaffen werden.

Soll das PMS sämtliche Konfigurationsschritte zur Integration eines Plugins im System koordinieren, so bietet sich die Technik an, die notwendigen Kanalinstanzen innerhalb des PMS zu verschmelzen. In Abbildung 4.15 ist eine Implementierung der Verkettung aller Konfigurationsaktionen zum Hinzufügen bzw. Entfernen mittels rekursiver Verschmelzung synchroner Kanäle dargestellt.[19] Das PMS erfragt im Moment des Hinzufügens bzw. Entfernens eines Plugins eine Liste der erweiterbaren Systeme, die vom Plugin erweitert werden, mittels des Kanals `p:getExtendedSpecs(el)`. Über eine rekursive Verschmelzung des Kanals `:subadd(el, q)` bzw. `:subremove(el, q)` wird je Listenelement von `el` eine Instanz der mittleren Transitionen im selben Schaltvorgang aktiviert. Jede Transitionsinstanz baut einen synchronen Kanal zum `:add(p)`- bzw. `:remove(p)`-Uplink des passenden erweiterbaren Systems auf. Um das passende erweiterbare System zu ermitteln, erfragt das PMS von den verfügbaren erweiterbaren Systemen mittels des Kanals `:getExtensionSpec(e)`, ob die passende Erweiterungsspezifikation `e` angeboten wird.

Um eine Liste erforderlicher erweiterbarer Systeme angeben zu können, muss die bisher implizite Typisierung der Konfigurationskanäle (zur Auflösung des Nichtdeterminismus innerhalb der Konfigurationskanalketten) *explizit* modelliert werden. Die explizite Angabe der Spezifikationen wird mittels Zeichenketten realisiert, die durch zusätzliche Kanäle in der äußeren Schnittstelle jedes Plugins ausgelesen werden. Wie bereits in Unterabschnitt 4.2.6.2 besprochen, kann diese Modellierung sowohl deklarative als auch imperative Implementierungen der Konfigurationsdaten repräsentieren.

[19] Die rekursive Verschmelzung synchroner Kanäle wird auch in Abbildung 3.6 angewandt, um die Wirkung der Mozilla-XPInstall-Technik zu visualisieren. Erläuterungen zur Technik finden sich im Text zur Abbildung auf S. 133.

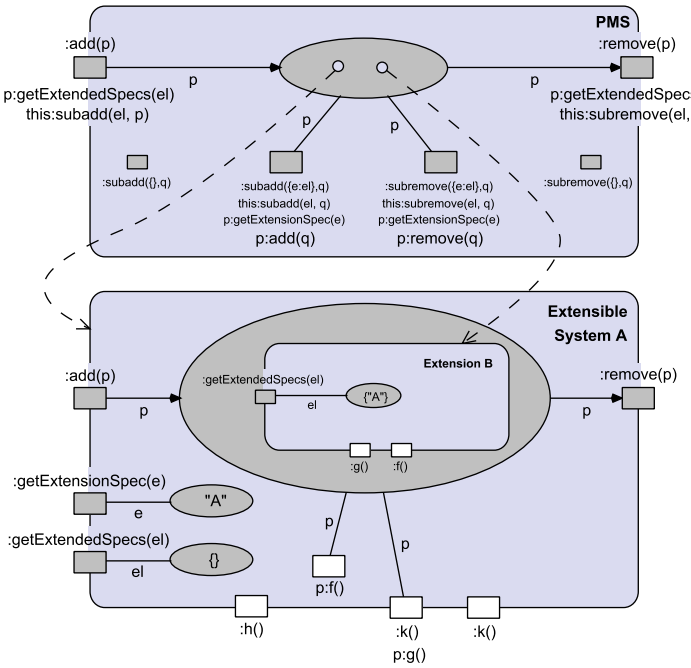



Abbildung 4.15: Atomare Konfiguration durch das PMS

Zu beachten ist jedoch, dass eine Repräsentation für Erweiterungsspezifikationen benötigt wird, die auch ohne das Vorhandensein des sie definierenden erweiterbaren Systems interpretierbar ist (im Modell in Form von Zeichenketten, z. B. „A“).

Die in Abbildung 4.15 vorgestellte Lösung wirkt atomar. Die Erweiterungsbeziehungen jedes Plugins werden beim Hinzufügen vollständig konfiguriert. Steht zu auch nur einer einzigen genutzten Erweiterungsspezifikation kein passendes erweiterbares System zur Verfügung, so scheitert der gesamte Schaltvorgang, da die rekursive Verschmelzung der Kanäle verhindert wird. Die Konfiguration von Systemkomponenten, die nicht zueinander in Beziehung stehen (sollen), kann dennoch nebenläufig erfolgen. Zur Integration eines Plugins werden nur Marken der passenden erweiterbaren Systeme im Gesamtsystem herangezogen, so dass alle anderen Systemkomponenten zeitgleich hinzugefügt oder entfernt werden können. Jedes erweiterbare System kann wie zuvor mehrere Erweiterungen zeitgleich konfigurieren, da es im PMS lediglich über Testkanten gebunden wird.

Die Kanäle `:subadd(q)` und `:subremove(q)` sind nicht mehr Teil der äußeren Schnittstelle des PMS, da sie nur noch intern im Rahmen der atomaren Konfiguration verwendet werden. Die innere Schnittstelle des PMS zu seinen Erweiterungen (also sämtlichen Plugins im System) ist um zwei Kanäle `:getExtensionSpec(e)` und `:getExtendedSpecs(el)` vergrößert worden. Ersterer kann von einer Systemkomponente vernachlässigt werden, falls sie kein erweiterbares System ist. Letzterer ist keines-

wegs optional, da er während der Rekonfiguration des Systems in jedem Fall abgefragt wird. Auch erweiterbare Systeme, die nicht als Erweiterung eines anderen erweiterbaren Systems agieren, müssen diesen Kanal anbieten (und eine leere Liste angeben).

Das in Abbildung 4.15 modellierte System schränkt die Flexibilität der Systemkonfiguration im Vergleich zu den vorangegangenen Modellierungsschritten geringfügig ein. Um die mehrfachen Konfigurationsschritte zur Integration eines Plugins mit mehrfacher Erweiterungsbeziehung nach außen hin zu einem einfachen Schritt zusammenzufassen, werden sämtliche Konfigurationsoperationen in einem atomaren Schritt durchgeführt. Die atomare Konfiguration bedingt, dass sich die Beziehungen zu erweiterten Systemkomponenten während der Aufenthaltszeit eines als Erweiterung integrierten Plugins im System nicht ändern können und dürfen. Die Informationen, die durch die beiden Kanäle `:getExtensionSpec(e)` und `:getExtendedSpecs(el)` von den Systemkomponenten zum PMS fließen, dürfen sich während der Aufenthaltszeit der Systemkomponente im System nicht ändern, da sonst die Konfiguration beim Entfernen der Komponente scheitert. Die Plugins haben keine Möglichkeit, von sich aus Konfigurationsänderungen zu veranlassen. Die Dauer jeder lokalen Erweiterungsbeziehung stimmt exakt mit der Dauer der globalen Beziehung des erweiternden Plugins zum Gesamtsystem überein.

Trotz dieser Einschränkungen handelt es sich bei dem in diesem Schritt modellierten System um ein dynamisch rekonfigurierbares System. Es ist jederzeit möglich, weitere Plugins zum System hinzuzufügen und auch wieder daraus zu entfernen. Die Einschränkung auf atomare Konfiguration führt zu einem nicht nur für den Anwender, sondern auch für den Entwickler eines Plugins sehr einfach konfigurierbaren System. Es müssen lediglich die Erweiterungsbeziehungen des Plugins deklariert und die fachlichen Verwaltungen für Erweiterungen implementiert werden. Das hier modellierte System ist als konzeptionell fast vollständiges Plugin-System verwendbar, da bis auf das Definieren eines Lebenszyklus für Plugins alle Anforderungen aus Kapitel 4 erfüllt sind.

Das Fehlen eines forcierten Lebenszyklus äußert sich in diesem Modell in einem möglicherweise unerwünschten Verhalten. Falls erweiterbare Systemkomponenten aus dem PMS mittels `:remove(p)` entfernt werden, während noch Erweiterungen konfiguriert sind, so bleiben die Erweiterungsbeziehungen erhalten, da die entfernte Systemkomponente sämtliche konfigurierten Netzreferenzen in sich behält. Diese Eigenschaft könnte unter Umständen erwünscht sein, falls neben dem vom PMS verwalteten Teil des Systems noch andere Bereiche existieren, in denen ein Fortdauern der konfigurierten Beziehungen sinnvoll ist. Unter der Annahme, dass das PMS die alleinige Kontrolle über die gesamte Systemkonfiguration haben soll, ist das Verhalten jedoch als Fehler zu betrachten. Der Fehler lässt sich leicht beheben, indem das PMS über sämtliche konfigurierten Erweiterungsbeziehungen Buch führt und diese Informationen beim Entfernen von Systemkomponenten berücksichtigt, indem z. B. das Entfernen der angeforderten Systemkomponente verweigert oder sämtliche erweiternden Plugins ebenfalls entfernt werden. Die Korrektur dieses Verhaltens soll hier nicht detailliert ausgeführt werden, da eine atomare Implementierung mehrere zusätzliche, der Verständlichkeit des Modells kaum zuträgliche Kanalrekursionen erforderte.

Im Unterabschnitt 4.3.8.2 wird stattdessen auf Basis der dann definierten Lebenszyklen eine Alternative vorgestellt, in welcher jedes Plugin selber seine Integration

koordiniert. Da Plugins auch die Möglichkeit erhalten sollen, ihre Beziehungen untereinander während ihres Aufenthaltes im System flexibel zu konfigurieren, wird diese Integration nicht mehr atomar stattfinden. Zuvor muss das Modell jedoch so angepasst werden, dass Plugins überhaupt eine Schnittstelle zur Konfiguration zur Verfügung steht. Das bisherige Modell hat Erweiterungen nur als passive Objekte im Konfigurationsprozess betrachtet.

### 4.3.8 Kommunikation und Koordination

Wie bereits zu Beginn des vorigen Unterabschnitts erläutert, besteht die zweite grundsätzliche Möglichkeit, dem externen Anwender die Koordination von Konfigurationsschritten zu ersparen, darin, die Konfigurationskoordination den individuellen Systemkomponenten aufzubürden. Die Umsetzung dieser Idee erfordert allerdings einige Änderungen an der Systemarchitektur:

1. Alle Systemkomponenten benötigen eine Referenz auf die zentrale Verwaltung, um deren Konfigurationsschnittstelle nutzen zu können.

2. Alle Systemkomponenten müssen über den Zeitpunkt ihrer eigenen Integration ins bzw. Entfernung aus dem Gesamtsystem informiert werden, um notwendige Beziehungen rechtzeitig knüpfen bzw. auflösen zu können.

3. Der direkte Zugriff auf Konfigurations- und andere Schnittstellen beliebiger Systemkomponenten neben dem PMS lässt sich leichter durch eine flach organisierte Systemarchitektur neben der hierarchischen bzw. azyklischen Struktur der rekursiven Erweiterbarkeit realisieren.

Die ersten beiden Punkte lassen sich durch eine Lebenszyklusverwaltung lösen, der dritte Punkt empfiehlt die Unterstützung der Dienstbeziehung im System. Beide Lösungsansätze sind unabhängig voneinander einsetzbar, ergänzen sich jedoch in Kombination besonders gut.

Im Folgenden werden beide Lösungsansätze grob skizziert. Sowohl bei der Modellierung von Lebenszyklen als auch bei der Modellierung einer dienstbasierten Systemarchitektur stehen verschiedene Optionen zur Wahl. Als Beispiele für Wahlmöglichkeiten seien die Referenzierungsrichtung zwischen Systemkomponenten, die Abgrenzung einzelner Schnittstellen oder die Ausgestaltung des Konfigurationsprozesses genannt. Eine detaillierte Ausarbeitung der verschiedenen Optionen wäre ähnlich umfangreich wie die bereits durchgeführte Diskussion von Modellierungsmöglichkeiten zur Modellierung der Beziehung der intendierten Erweiterbarkeit. Im Rahmen dieser Arbeit kann diese Diskussion nicht umfassend geführt werden, daher beschränken sich die folgenden Unterabschnitte auf die grobe Präsentation ausgewählter Modellierungsoptionen. In Unterabschnitt 4.3.8.4 werden die offen gelassenen Diskussionspunkte kurz umrissen.

#### 4.3.8.1 Dienste

Plugin-Systeme basieren zwar – wie in Unterabschnitt 4.1.3.1 diskutiert – in erster Linie auf dem Konzept der intendierten Erweiterbarkeit, unterstützen aber in der Regel als hybride Systeme auch die Erweiterbarkeit durch Angebot und Nutzung von

Diensten. Oft benötigen viele Plugins eines Systems einheitliche oder zentrale Dienstleistungen, welche aus Gründen der Wartbarkeit nicht unmittelbar vom äußersten erweiterbaren System als Teil des Erweiterungskontextes bereitgestellt werden sollen.

Die Einführung von Dienstbeziehungen erleichtert im Modell die Verlagerung der Konfigurationskontrolle vom externen Anwender ins Innere des Systems, ohne eine unflexible atomare Konfiguration von Plugins bei ihrer Integration zu erzwingen, wie es in Unterabschnitt 4.3.7 geschehen ist. Die äußeren Konfigurationsschnittstellen aller Systemkomponenten, also deren Kanäle `:add(p)` und `:remove(p)`, können als Dienst betrachtet werden, der allen Plugins im System angeboten wird, welche ihren Konfigurationsprozess selber steuern können sollen. Die bisher als konform zur Erweiterungsspezifikation des PMS definierten Kanäle lassen sich als konform zu einer Dienstspezifikation betrachten, da die Plugins keinen expliziten Gebrauch von der im Kontext der Erweiterungsspezifikation geboten Funktionalität machen.

Das in den bisherigen Schritten entwickelte Modell kann so ergänzt werden, dass es auch die Kopplung von Systemkomponenten anhand der Dienstbeziehung unterstützt. Zu diesem Zweck wird das PMS um die Verwaltung von dienstbasierten Beziehungen ergänzt. Die Ausführlichkeit der Diskussion der Modellierung dienstbasierter Systeme bleibt dabei hinter der intendiert erweiterbarer Systeme zurück, da die gleichrangig detaillierte Betrachtung dienstbasierter Systeme genug Material für eine weitere Dissertation böte. Die Betrachtung konzentriert sich auf die Modellierung der zentralen Diensteverwaltung, die Anpassung der verwalteten Systemkomponenten wird erst am Ende der Modellierung besprochen.

Essentiell für dienstbasiert gekoppelte Systeme ist die Existenz eines zentralen Diensteverzeichnisses (*Service Registry, SR*), welches sämtlichen Dienstanbietern und -nutzern bekannt ist, da hierüber die dynamische Kopplung der Systemkomponenten stattfindet. Jede dienstnutzende Dienstkomponente kann über das Diensteverzeichnis passende Dienstanbieter erfragen. Solange das Diensteverzeichnis keine über die Funktion als Auskunftei hinausgehenden Aufgaben wie z. B. eine Kontrolle der Lebensdauer von Dienstbeziehungen übernimmt, wird es nach erfolgreicher Kopplung weder vom Dienstanbieter noch vom Dienstnutzer mehr benötigt.

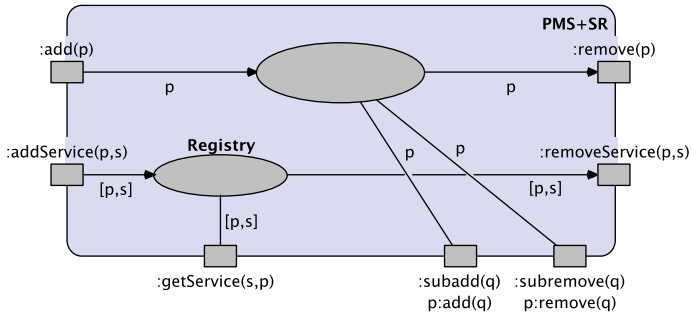



Abbildung 4.16: PMS mit Dienstverzeichnis

Eine Zusammenführung der dienstorientierten SR mit dem erweiterungsorientierten PMS aus Unterabschnitt 4.3.7 ist unproblematisch. In Abbildung 4.16 ist das grobe Prinzip exemplarisch dargestellt, das einfach die beiden Verwaltungselemente nebeneinander stellt.

Obwohl die grafische Struktur der SR der der Verwaltung eines erweiterbaren Systems stark ähnelt, ist die Semantik eine andere. Statt Systemkomponenten zu „enthalten“, verfügt die SR über eine Tabelle mit Dienstanbietern zu jeder Dienstspezifikation, im Netz als zweistellige Tupel aus Spezifikationsbezeichner `s` und Netzreferenz `p` auf den Dienstanbieter repräsentiert. Wie bei der Modellierung der atomaren Konfiguration durch das PMS in Unterabschnitt 4.3.7 werden *explizit* kommunizierbare Spezifikationsbezeichner benötigt. Bietet eine Systemkomponente mehrere Dienste an, so taucht sie in mehreren Tupeln mit unterschiedlichen Spezifikationszuordnungen in der SR auf.

Dienstanbieter können durch Aktivierung eines der Kanäle `:addService(p,s)` und `:removeService(p,s)` im Diensteverzeichnis an- und abgemeldet werden. Dienstnutzer können über den Kanal `:getService(s,p)` einen zur Dienstspezifikation `s` passenden Anbieter `p` als Netzreferenz erhalten.

In Abbildung 4.16 sind viele Fragen offen gelassen, die sich aus der Interaktion zwischen Dienstanbietern, -nutzern, erweiterbaren Systemen und Erweiterungen mit den jeweils zugehörigen Verwaltungsschnittstellen ergeben. Die Kopplung der Systemkomponenten mittels der dargestellten Schnittstellen ist auf viele verschiedene Weisen möglich. Beispielsweise könnte, wie in Abbildung 4.17 dargestellt, die im vorigen Unterabschnitt durchgeführte Synchronisierung aller Konfigurationsschritte auch auf die Registrierung von Dienstanbietern ausgedehnt werden. Jede Systemkomponente müsste dann eine Liste von Registry-Einträgen über den Kanal `:getServices(sds)` bereitstellen. Der Kanal wird jeweils über Downlinks an den Transitionen „add“ und „remove“ aktiviert, so dass die über den Kanal kommunizierten Registry-Einträge entlang der flexiblen Kante in das Diensteverzeichnis aufgenommen bzw. wieder daraus entfernt werden können.

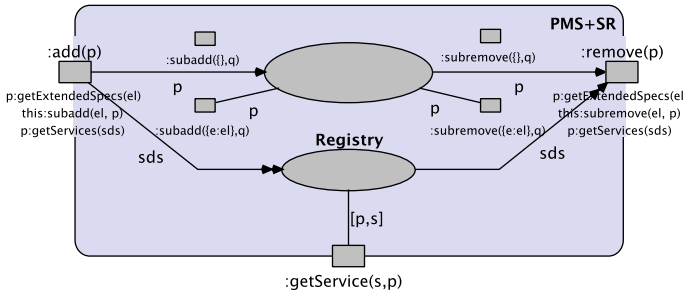



Abbildung 4.17: PMS mit Dienstverzeichnis und atomarer Konfiguration

In Abbildung 4.17 ist lediglich die Abfrage passender Dienstanbieter durch Nutzer nicht in die atomare Konfiguration aufgenommen worden. Auch dies ließe sich durch eine weitere synchrone Kanalverschmelzung und eine von jeder Systemkomponente

bereitzustellende Liste an benötigen Dienstspezifikationen einfach realisieren. Im Ergebnis bestünde dann eine Verwaltung, welche die automatische Auflösung *funktionaler Abhängigkeiten* (siehe Unterabschnitt 3.4.2) der verwalteten Systemkomponenten übernähme.

Die Einführung von Dienstbeziehungen in die Systemarchitektur würde aber bei atomarer Konfiguration nur eine weitere Verknüpfungsmöglichkeit für Systemkomponenten darstellen, die genauso wenig flexibel wäre wie das bisher vorgestellte Konzept zur Verwaltung von Plugins. Die atomare Konfiguration stößt zudem an ihre Grenzen, wenn eine Systemkomponente in Beziehung mit sich selber treten möchte, um z. B. der eigenen Funktionalität als Rückfallebene eine einfache Implementierung einer Erweiterung oder eines Dienstangebots beiseite zu stellen. Das Schalten einer einzigen Transition reicht nicht, um Beziehungen zu Netzmarken aufzubauen, die erst während des Schaltvorgangs in die Stellen gelegt werden, aus denen die Marken gebunden werden. Eine Fallunterscheidung, die auch den Zugriff auf Netzreferenzen des gerade integrierten Plugins ermöglicht, ließe sich zwar in die synchrone Kanalverschmelzung einbauen, wäre aber sehr aufwändig und schwer nachvollziehbar.

#### 4.3.8.2 Lebenszyklen

Das zuvor in Abbildung 4.16 dargestellte Netz zur Verwaltung von Beziehungen erzwingt keine atomare Konfiguration aller Dienst- und Erweiterungsbeziehungen eines Plugins, da die einzelnen Konfigurationskanäle zu beliebigen Zeitpunkten aktiviert werden können. Ein Verzicht auf atomare Konfiguration erlaubt es, die Integration oder Entfernung eines Plugins als beliebig komplexen Prozess zu betrachten. Allerdings müssen Anfang und Ende des Prozesses klar definiert werden, um einfach zwischen konsistenten Systemkonfigurationen und Zwischenzuständen unterscheiden zu können. Dienst- und Erweiterungsbeziehungen sollten nur genutzt werden, solange keine der beteiligten Systemkomponenten sich in einem Konfigurationsprozess befindet. Der Lebenszyklus jeder Systemkomponente muss klar definiert und für die Systemkomponenten wahrnehmbar modelliert werden.

In den bisherigen Modellierungsschritten ist ein einfacher Lebenszyklus sämtlicher Systemkomponenten enthalten, der durch die Transitionen `:add(p)` und `:remove(p)` des erweiterbaren Systems bzw. des PMS definiert wird. Allerdings wurde dieser Lebenszyklus bisher vom ihm unterliegenden Plugin nicht wahrgenommen, da es nur als passives Element an der Konfiguration der erweiterten Systemkomponenten beteiligt war. Je ein zusätzlicher synchroner Kanal an den Konfigurationstransitionen des PMS ermöglicht dem hinzugefügten bzw. entfernten Plugin eine Beteiligung am Konfigurationsprozess.

Das PMS in Abbildung 4.18 implementiert einen solchen einfachen Lebenszyklus mit den Kanälen `:PMSadd(pms)`[20] und `:PMSremove()`, die nun zur äußeren Schnittstelle jeder Systemkomponente, also jedes Plugins, gehören müssen. Die Kanäle lassen sich

[20] In Abbildung 4.18 ist der nur der Downlink `p:PMSadd(this)` zum Kanal mit der Signatur `:PMSadd(pms)` dargestellt, da es sich aus Sicht des PMS um eine innere Schnittstelle handelt. Im dargestellten Netz wird die Selbstreferenz `this` als Parameter übergeben, welche in passenden Plugin-Netzen an die dort verwendete Variable gebunden werden wird.

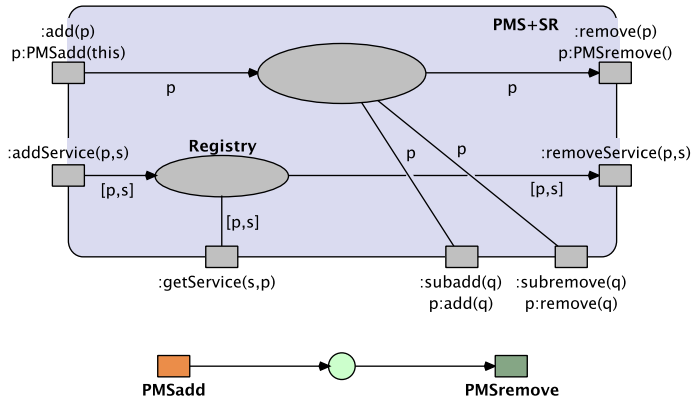



Abbildung 4.18: PMS mit einfachem Lebenszyklus

als Teil der Erweiterungsspezifikation des PMS auffassen und definieren den unten in Abbildung 4.18 dargestellten Lebenszyklus.

Der Kanal `:PMSadd(pms)` eines Plugins lässt sich für mehrere Aufgaben verwenden:

- Das hinzuzufügende Plugin wird über seine Integration ins System *informiert* und kann darauf z. B. mit einer internen Initialisierung reagieren.
- Das hinzuzufügende Plugin kann die *Aktivierung des Kanals beeinflussen* und somit das Hinzufügen unterbinden, falls aus seiner Sicht etwas gegen eine Integration spricht.
- Das hinzuzufügende Plugin bekommt über den Kanal `:PMSadd(pms)` eine *Referenz auf das PMS-Netz* übergeben.
- Das hinzuzufügende Plugin kann die übergebene Referenz nutzen, um rekursiv auf die Konfigurationsschnittstelle des PMS zuzugreifen und weitere *Beziehungen zu konfigurieren.*

Insbesondere der letzte Punkt erlaubt die Verlagerung der Konfigurationskoordination bei der Integration in die individuellen Plugins.

Entsprechend kann der Kanal `:PMSremove()` eines Plugins verwendet werden:

- Das zu entfernende Plugin wird über seine Entfernung aus dem System *informiert* und kann darauf z. B. mit einer internen Zustandsneutralisierung reagieren.
- Da das Plugin bei der Integration schon eine Referenz auf das PMS bekommen hat, kann es diese nutzen, um über die Konfigurationsschnittstelle des PMS weitere *Beziehungen aufzulösen.*
- Das zu entfernende Plugin kann die *Aktivierung des Kanals beeinflussen* und somit das Entfernen unterbinden, falls aus seiner Sicht etwas gegen eine Entfernung spricht. Hiermit ließe sich beispielsweise das Entfernen unterbinden, falls

dem Plugin weitere Beziehungen zu anderen Systemkomponenten bekannt sind, welche es nicht selber auflösen kann oder will.

Welche dieser Verwendungsmöglichkeiten im Plugin-System erwünscht sind, müssen die Entwickler der Systemarchitektur bei der Definition des Lebenszyklus beispielsweise in Form eines Nutzungsvertrages in der Plugin-Spezifikation festlegen. Das Petrinetzmodell des PMS bzw. das daraus extrahierte Lebenszyklusnetz gibt als Teil der Plugin-Spezifikation den protokollarischen Ablauf dieser Verwendungsmöglichkeiten vor.

Der in Abbildung 4.18 vorgegebene Lebenszyklus ist sehr einfach gehalten. Stattdessen können auch komplexere Lebenszyklen wie die in Kapitel 3 beobachteten definiert und – wie in Unterabschnitt 4.2.7 bereits umrissen – mit der Konfigurationsschnittstelle des PMS verzahnt werden. In Unterabschnitt 4.4.2 werden einige der Lebenszyklen aus Kapitel 3 auf ihre Verwendbarkeit im Modell untersucht.

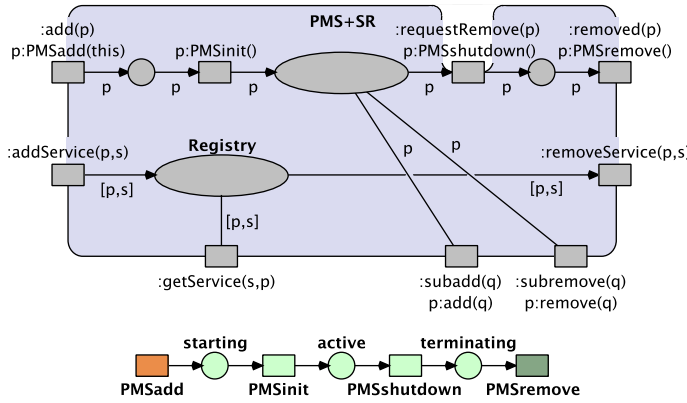



Abbildung 4.19: PMS mit komplexerem Lebenszyklus

Es ist allerdings festzustellen, dass ein komplexerer Lebenszyklus den Implementierungsaufwand und die Fehleranfälligkeit des Systems erhöht. Zur Illustration ist in Abbildung 4.19 ein PMS mit einem Lebenszyklus dargestellt, der dem Plugin die Möglichkeit gibt, seine Mitwirkung am Konfigurationsprozess nicht-atomar zu gestalten. Im Prinzip ist eine nicht-atomare Konfiguration flexibler, da die zu integrierende Systemkomponente komplexe Prozesse ablaufen lassen kann, statt sämtliche Initialisierungs- und Konfigurationsschritte in einem einzigen Schaltvorgang verschmelzen zu müssen.

Der Preis für diese Flexibilität wird in Abbildung 4.19 sichtbar: Die äußere Konfigurationsschnittstelle des PMS wird komplexer, da sie die Zeitbehaftetheit des Konfigurationsprozesses reflektieren muss. In der Abbildung sind zwei Möglichkeiten dargestellt, die sich hinsichtlich der Rückmeldung des Konfigurationsprozessabschlusses unterscheiden: Die Ausdehnung der Initialisierungsphase auf zwei Kanalkommunikationen (Beginn und Ende der Initialisierungsphase) ist nicht nach außen sichtbar, da nur der bereits bekannte Uplink `:add(p)` in der äußeren Schnittstelle angeboten wird.

Damit kann allerdings der externe Initiator des Konfigurationsprozesses nicht in Erfahrung bringen, wann der Konfigurationsprozess abgeschlossen ist. Die Ausdehnung der Konfigurationsphase zum Entfernen eines Plugins ist durch einen zusätzlichen nach außen sichtbaren Kanal `:requestRemove(p)` in der äußeren Schnittstelle des PMS modelliert. Ohne diesen Kanal wäre ein Externer nicht in der Lage, die Entfernung von Plugins anzustoßen, sondern könnte über den Kanal `:removed(p)` nur Plugins abholen, die „spontan“ (d.h. auf eigene Veranlassung oder auf Veranlassung des PMS) in den Entfernungsprozess eingetreten sind.[21] Beide Möglichkeiten lassen sich auch auf die jeweils andere Konfigurationsphase anwenden (sind aber nicht grafisch dargestellt): Ein zusätzlicher Kanal `:added(p)` an der Transition zum Abschluss der Initialisierungsphase könnte die Information der erfolgreichen Integration eines Plugins nach außen weitergeben. Ein Verzicht auf den Kanal `:removed(p)` – die damit beschriftete Transition würde zu einer internen Transition des PMS werden – würde eine externe Steuerung von der Aufgabe entbinden, auf das Ende des Entfernungsprozesses zu warten. Solch eine Modellierung wäre dann sinnvoll, wenn nur ein asynchroner Steuerungsimpuls modelliert werden soll.

Die Umstellung von einer atomaren Konfiguration auf eine durch zwei synchrone Kanäle begrenzte asynchrone Konfigurationsphase erfordert besondere Vorsicht bei der Festlegung der Semantik jedes Kanals. Die Möglichkeit der Nicht-Aktivierung eines Kanals durch das Plugin ist nur zu Beginn der Konfigurationsphase sinnvoll – ein Nicht-Aktivieren des zweiten Kanals würde zu einem Liegenbleiben der Netzmarke in den Pufferstellen führen, so dass die Konfiguration nicht abgeschlossen ist. Diese Vorgabe kann nur durch korrektes Verhalten aller beteiligten Systemkomponenten eingehalten werden. Ein fehlerhaft implementiertes Plugin kann somit zu einem inkonsistenten Zustand führen. Es ist daher wichtig, dass das PMS die sichtbare Integration bzw. Entfernung einer Systemkomponente an den beiden inneren Transitionen durchführt. Falls z. B. zusätzlich eine vom PMS koordinierte Konfiguration von Erweiterungs- oder Dienstbeziehungen stattfindet, sollte dies an der jeweils „richtigen“ Transition geschehen.

#### 4.3.8.3 Mehrere Ebenen

Mit der Definition eines Lebenszyklus durch das PMS wird festgelegt, dass jedes Plugin die passenden Kanäle als äußere Schnittstelle anbieten muss. Ob und wie das Plugin die über die Kanäle kommunizierte Information zur eigenen Initialisierung oder Konfiguration weiterer Beziehungen nutzt, ist freigestellt. In der Regel kann ein beliebiges Netz zum Plugin umfunktioniert werden, indem Transitionen mit passenden Uplinks ergänzt werden, diese Transitionen brauchen aber nicht unbedingt mit der restlichen Funktionalität des Netzes verbunden zu werden. Die Integration des Lebenszyklus in die Funktionalität einer Systemkomponente ist beliebig, solange die im Lebenszyklus vorgegebenen Prozesse nicht blockiert werden.

Gleiches gilt für Lebenszyklen, die sich aus anderen Beziehungen ergeben. Für Dienstbeziehungen wird in der Regel kein gesonderter Lebenszyklus definiert, die Kopplung und Entkopplung von Nutzer und Anbieter reicht zur Konfiguration der

[21] Aufgrund der veränderten Semantik des `:remove(p)`-Kanals (statt Aufforderung nur noch Bestätigung der Durchführung) wird der Kanal in `:removed(p)` umbenannt.

Beziehung aus.[22] Erweiterbare Systemkomponenten können hingegen mit ihren Konfigurationsschnittstellen eigene Lebenszyklen definieren, sie bedienen sich dabei derselben Technik wie das PMS.

In komplexen Systemen unter Referenzsemantik ist es daher möglich, dass ein Plugin zu mehreren Lebenszyklusdefinitionen konform implementiert werden muss. Auch hier gilt, dass die Transitionen mit den benötigten Kanalendpunkten zu den Lebenszyklen des PMS und aller erweiterten Systeme ohne Aufwand innerhalb der erweiternden Systemkomponente kombiniert werden können, indem sie einfach nebeneinander gestellt werden. Eine Kopplung der Lebenszyklen innerhalb eines mehrfach integrierten Plugins ist möglich und zulässig, solange keine Vorgaben eines Lebenszyklus verletzt werden. Eine sinnvolle Kopplung ergibt sich aus dem individuellen Kontext des Plugins und kann nicht allgemein beantwortet werden.

Abbildung 4.20 zeigt ein Beispiel, in dem sämtliche Vorschläge dieses Unterabschnitts – also Dienstverzeichnis, Lebenszyklen und eine Verwaltung auf mehreren Ebenen – zusammenkommen. Die Konfigurationskoordination obliegt im Wesentlichen den Plugins, allerdings bietet das PMS zusätzlich zur individuellen Registrierung jedem Plugin die atomare Registrierung von Diensten in einem Lebenszykluszustandsübergang an. Diese Kombination wurde gewählt, weil so die Konfigurationsschnittstelle jeder erweiterbaren Systemkomponente als Dienstangebot unter der Dienstspezifikation namens `extMgmt` (für „*extension management*“) aufgefasst werden kann. Das mittlere Netz stellt ein erweiterbares Plugin dar, welches zugleich als Dienstanbieter agiert, indem es seine Konfigurationsschnittstelle als Dienst meldet. Das unterste Plugin tritt als Dienstnutzer der Konfigurationsschnittstelle des mittleren Plugins auf, um sich in Eigeninitiative dort als Erweiterung zu registrieren.

Beide Plugins implementieren den vom PMS geforderten Lebenszyklus mit den vier Zustandsübergängen `PMSadd`, `PMSinitialize`, `PMSshutdown` und `PMSremove`. Das unterste Plugin implementiert zusätzlich den Lebenszyklus der Erweiterungsspezifikation des mittleren Plugins, welcher die zwei Zustandsübergänge `configure` und `deconfigure` umfasst. Die kleinen Stellen im untersten Netz stellen exemplarisch eine Kopplung der Integrationskoordination an den vom PMS vorgegebenen Lebenszyklus her. Ob und wie das unterste Plugin den von der mittleren Systemkomponente vorgegebenen Lebenszyklus mit zwei Kanälen `:configure()` und `:deconfigure()` nutzt, sei hier offen gelassen – eine Initialisierung interner Zustände zur Vorbereitung der nicht weiter konkretisierten Transitionen mit den Uplinks `:f()` und `:g()` ist denkbar und sinnvoll, aber nicht unbedingt notwendig.

#### 4.3.8.4 Diskussion

In den vorangegangenen Unterabschnitten sind mit Lebenszyklen und Dienstbeziehungen zwei Konzepte zur Flexibilisierung der Systemkonfiguration vorgestellt worden. Beide Konzepte erhöhen aber nicht nur die Flexibilität, sondern auch die Komplexität der Systemarchitektur und somit das Risiko für Implementierungsfehler, falls sich die Entwickler bei der Implementierung eines konkreten Systems der Anforderungen und Seiteneffekte des jeweiligen Konzepts nicht bewusst sind.

[22] Eine Ausnahme stellen die zentral koordinierten und flexibel auf- und abgebauten Dienst-Lebenszyklen des OSGi-Rahmenwerkes dar (siehe Unterabschnitt 3.2.1.6).

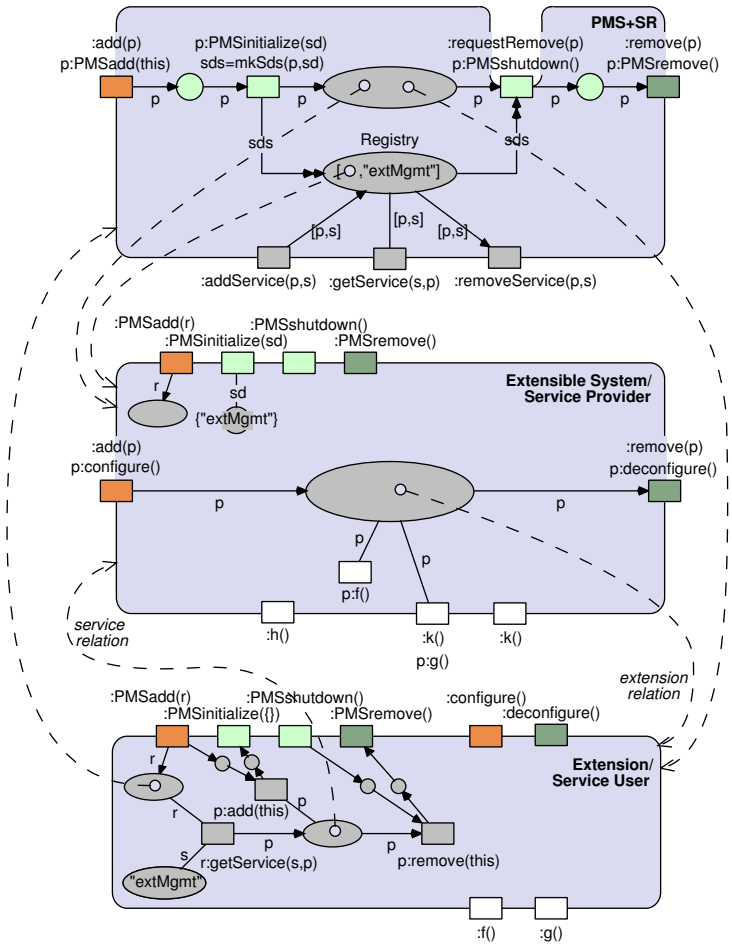



Abbildung 4.20: Erweiterbares System mit PMS und einer sich selbst konfigurierenden Erweiterung

Mit dem in Unterabschnitt 4.3.8.2 modellierten Konzept eines Lebenszyklus können Plugins an ihrem Integrationsprozess aktiv teilhaben und gegebenenfalls sogar die Rekonfiguration des Systems verhindern. Die Einführung mehrschrittiger Zustandsübergänge flexibilisiert den Konfigurationsprozess, da komplexe Abläufe modellierbar werden.

In Kombination mit der in Unterabschnitt 4.3.8.3 diskutierten Einführung individueller Lebenszyklen zu verschiedenen Erweiterungs- (oder Dienst-)spezifikationen können (und müssen) Systemkomponenten konform zu mehreren Lebenszyklen agieren. Das Aufeinandertreffen mehrerer Beziehungen und damit Lebenszyklen tritt nur in verwalteten Systemkomponenten, also Erweiterungen auf. Im PMS sowie in erweiterbaren Systemkomponenten, welche eigene Lebenszyklen definieren, sind die Lebenszyklustransitionen an die konkrete Verwaltungsstruktur gebunden. Ist eine Systemkomponente in mehreren Dimensionen unabhängig erweiterbar (siehe Unterabschnitt 4.3.3), so stehen die Verwaltungselemente in der Regel unabhängig nebeneinander. Sollten Abhängigkeiten zwischen verschiedenen Erweiterungsschnittstellen bestehen, lässt sich ein gemeinsamer Lebenszyklus definieren, so dass keine Komplikationen zu befürchten sind.

Die Einführung mehrerer komplexer Lebenszyklen führt im System zu einem Anstieg der Gefahr von Verklemmungen oder inkonsistenten Zuständen aufgrund von Nebenläufigkeit in den Konfigurationsprozessen. Die mit jedem Lebenszyklus definierte Anforderung an passende Erweiterungen, sich an die Vorgaben des Lebenszyklus zu halten und die definierten Kanäle zum richtigen Zeitpunkt zu aktivieren, birgt grundsätzlich die Gefahr, dass ein fehlerhaftes Plugin das erweiterbare System in einem Zwischenzustand des Konfigurationsprozesses festhält. Die Modellierung als Petrinetz öffnet den Weg zur Analyse auf Vorkommen solcher Probleme und zu Vermeidungsstrategien, die hier aber nicht weiter erörtert werden sollen. Die im Unterabschnitt 4.3.8.2 präsentierten Netze (wie z. B. jenes in Abbildung 4.19) haben allerdings die angenehme Eigenschaft, dass nur der zu diesem einen Plugin gehörende Kontrollfluss verklemmt, falls das Plugin den jeweils zweiten Konfigurationsschritt nicht aktiviert. Da keine anderen Marken während der Konfigurationsphasen reserviert werden, kann das restliche System weiter operieren, als ob das fehlerhafte Plugin nicht existierte.

Grundsätzlich gilt natürlich, dass die Verhinderung der Beteiligung von Plugins an ihrer Integration zu einem geringeren Fehlerrisiko führt, da die gesamte Konfiguration zentral und atomar gesteuert werden kann. Von den in Kapitel 3 untersuchten Plugin-Systemen verfolgen aber nur das MagicBeans-System (siehe Unterabschnitt 3.3.4) und die Mozilla-Chrome-Schicht (siehe Unterabschnitt 3.1.2.3) diesen Weg konsequent. Alle anderen Plugin-Systeme erlauben zusätzlich zur zentralen Konfiguration eine Partizipation der Plugins im Rahmen eines Lebenszyklus. Die geeignete Ausgestaltung von Lebenszyklen in Plugin-Systemen wird erst später in Unterabschnitt 4.4.2 diskutiert werden, wenn die Lebenszyklen der in Kapitel 3 untersuchten Plugin-Systeme besprochen werden.

Die in Unterabschnitt 4.3.8.1 modellierte Integration von Dienstbeziehungen in die Plugin-Architektur hilft, „gekünstelte“ Erweiterungsbeziehungen zu vermeiden. Als gekünstelte Erweiterungsbeziehung sei hier eine Erweiterungsbeziehung verstanden, die eigentlich nicht auf eine fachlich intendierte Erweiterbarkeit zurückzuführen ist, sondern lediglich der Bereitstellung und Nutzung von kontextunabhängiger Funktionalität

quer über Systemkomponenten hinweg dient. In einem System, welches ausschließlich die Erweiterungsbeziehung zur Kopplung von Systemkomponenten einsetzt, wären solcherart gekünstelte Beziehungen kaum zu umgehen.

Obwohl auch Dienstbeziehungen atomar und zentral konfiguriert werden können, suggeriert die Metapher der Dienstnutzung eher eine individuelle Lebensdauer einzelner Beziehungen. Letztere kann allerdings nicht ohne die Einführung von Lebenszyklen ermöglicht werden. Die ausführliche Modellierung von Dienstbeziehungen würde ähnlich viele Überlegungen und Verfeinerungsschritte wie die in diesem Abschnitt durchgeführte Modellierung der Erweiterungsbeziehung umfassen. Dienstbeziehungen wären auch unter einer *Netze-in-Netzen*-Semantik ohne Netzreferenzen modellierbar, wenn sämtliche Systemkomponenten als Netzmarken in einem zentralen Dienstinfrastruktur-Netz enthalten wären, welches neben der Aufgabe des Dienstverzeichnisses auch die fortwährende Kommunikation zwischen gekoppelten Dienstanbietern und -nutzern ermöglicht. Beim Verzicht auf Netzreferenzen würde die Dienstbeziehung daher keine andere Beziehung neben sich zulassen, da alle Komponenten flach in der Dienstverwaltung „enthalten“ sein müssen. Für die längerfristige Bindung von Dienstnutzern an individuelle Dienstanbieter müsste die Kommunikationsinfrastruktur besondere Vorkehrungen treffen.

Eine rekursive Modellierung der Dienstbeziehung ist zunächst nicht sinnvoll, da die Dienstverwaltungskomponente in der Regel keine fachliche Funktionalität bereitstellt und alle Dienstanbieter und -nutzer gleichermaßen flach organisiert sind. Allerdings kann auch jede Komponente für sich eine Dienstverwaltung mit eingeschränkter Sichtbarkeit anbieten, so dass analog zur unabhängigen Erweiterbarkeit in mehreren Dimensionen eine unabhängige Organisation von Dienstanbietern und -nutzern in mehreren Clustern möglich wäre. Falls diese Clusterung auf fachlichen Aspekten wie z. B. der inhaltlichen Verwandtschaft von Dienstspezifikation beruhte, käme diese Organisation allerdings einer Erweiterungsverwaltung sehr nahe und brächte keine wesentlichen Vorteile.

Zur Modellierung von Dienstbeziehungen drängt sich die Referenzsemantik geradezu auf, wenn man auf die explizite Modellierung einer zentralen Kommunikationsinfrastruktur verzichten möchte. Die Verwaltung von Netzreferenzen in einer allen Systemkomponenten bekannten (d.h. referenzierten) Registry führt zu zyklischen Referenzen in der Systemarchitektur. Die Einführung zyklischer Referenzen im Modell sorgt für einen drastischen Anstieg der Systemkomplexität und des Risikos fehlerhafter Verknüpfungen. Auch ein Umkehren (welches dank der bidirektionalen Kanäle problemlos möglich wäre) der Referenzierungsrichtung zwischen Diensteverzeichnis und den anderen Systemkomponenten würde zyklische Referenzen nicht erfolgreich vermeiden: Obwohl keine Systemkomponente mehr eine Referenz auf das PMS+SR-Netz benötigen würde, da jenes Netz ohnehin schon über Referenzen zu allen Systemkomponenten verfügt, könnten Referenzierungszyklen immer noch auftreten, sobald Systemkomponenten wechselseitig über Dienst- (oder Erweiterungs-)beziehungen miteinander verbunden wären. Insbesondere führt die Idee, die Konfiguration lokaler Erweiterungsbeziehungen als global verfügbaren Dienst zu betrachten, zwangsweise zu solchen Referenzierungszyklen.

Wie schon bei der Verwaltung von Erweiterungsbeziehungen gilt, dass ein System mit zentral gesteuerter, atomarer Konfiguration von Dienstbeziehungen den Ent-

wicklern individueller Systemkomponenten weniger Raum für Implementierungsfehler lässt.

Unter Referenzsemantik können Erweiterungs- und Dienstbeziehung wahlfrei kombiniert werden, ebenso können PMS und Dienstverwaltung zusammenfallen und Synergien bei der zentralen, atomaren Konfiguration oder der Definition von Lebenszyklen ausgenutzt werden. Die Beziehungen beider Modelle stehen unabhängig nebeneinander, können aber zum gegenseitigen Vorteil kombiniert werden. Die flache Organisation von Dienstanbietern und -nutzern ergänzt sich gut mit der tendenziell hierarchischen Organisation von erweiterbaren Systemen und Erweiterungen. Mit der Flexibilität steigt allerdings auch die Komplexität. Die den Entwicklern in einer hybriden Architektur ermöglichte Wahl zwischen Erweiterungs- und Dienstbeziehung lässt sich zudem nicht immer eindeutig treffen, da die Übergänge fließend sind (wie bereits in Unterabschnitt 4.1.3.1 diskutiert).

## 4.4 Anwendbarkeit des Modells auf existierende Plugin-Systeme

Das im vorigen Abschnitt entwickelte Plugin-Konzept soll nun im Vergleich zu der in Kapitel 3 vorgefundenen Vielfalt an praktischen Lösungen in existierenden Plugin-Systemen diskutiert werden. Als Ausgangspunkt des Abgleichs dient die zusammenfassende Diskussion existierender Plugin-Systeme in Unterabschnitt 3.4.1, wobei zur Vereinfachung nur die in Tabelle 3.2 namentlich genannten Systeme zur Diskussion herangezogen werden.

Entsprechend der Analyse im Unterabschnitt 3.4.1 soll ein besonderer Schwerpunkt der Diskussion auf den unterschiedlichen Beziehungsarten liegen, welche den jeweiligen Systemarchitekturen zugrunde liegen. Im Unterabschnitt 4.4.1 wird daher die Integrierbarkeit der in Unterabschnitt 3.4.2 erkannten Beziehungsarten in das konzeptionelle Modell des vorigen Abschnitts untersucht.

Das zweite Hauptaugenmerk gilt den Lebenszyklusnetzen, die während der Betrachtung existierender Plugin-Systeme in Kapitel 3 erarbeitet wurden. In Unterabschnitt 4.4.2 sollen die in den existierenden Systemen vorgefundenen Lebenszyklen miteinander verglichen und auf ihre Eignung im Rahmen der Erkenntnisse aus Unterabschnitt 4.3.8.2 überprüft werden.

Nachdem beide Schwerpunkte diskutiert wurden, können in Unterabschnitt 4.4.3 die untersuchten Plugin-Systeme noch einmal im Überblick auf ihre Modellierbarkeit im Rahmen der in dieser Arbeit entwickelten Konzepte untersucht werden. Abgerundet wird die Diskussion durch einige Anmerkungen zur Kompatibilität des entwickelten Modells mit den in Abschnitt 3.3 angeschnittenen Forschungsfeldern.

### 4.4.1 Beziehungsarten

Im Ergebnis des vorigen Abschnitts modelliert das entwickelte referenznetzbasierte Plugin-Konzept zwei Arten von Beziehungen, die hierarchische Erweiterungsbeziehung und die heterarchische Dienstbeziehung.

In der zusammenfassenden Diskussion existierender Plugin-Systeme tauchen deutlich mehr verschiedene Beziehungsbegriffe auf, namentlich Modul-, Objekt-, Erweiterungs- und Dienstbeziehung sowie Abhängigkeiten und Pfadreferenzen (siehe Unterabschnitt 3.4.2). Es stellt sich die Frage, ob das modellierte Plugin-Konzept diese Vielfalt an Beziehungsarten abdecken kann, ob die vielfältigen Begriffe sich vielleicht auf die zwei bereits modellierten Beziehungsarten zurückführen lassen, oder ob hinter dieser Begriffsvielfalt tatsächlich bisher noch nicht modellierte Konzepte verborgen sind.

Zunächst sollen die in Unterabschnitt 3.4.2 informell beschriebenen Beziehungsarten kurz darauf untersucht werden, inwieweit sie noch detailliert betrachtet werden müssen.

**Modulbeziehung** Die Modulbeziehung im technischen Sinne ist eine essentielle Grundlage von Plugin-Systemen. Sie wird aber in der Regel nicht explizit betrachtet, da sie als technisches Hilfsmittel auf einer anderen Abstraktionsebene als die anderen Beziehungsarten liegt. Lediglich in ganz einfachen Plugin-Systemen wie z. B. der NPAPI wird thematisiert, wie diese eigentlich statische Beziehung zwischen Code-Modulen dynamisiert werden kann. Die Modulbeziehung soll zum Zwecke der Ermittlung von Ähnlichkeiten zu den bereits modellierten Beziehungsarten ebenfalls modelliert werden.

**Objektbeziehung** Systeme auf Basis der allgemeinen Objektbeziehung wurden in Unterabschnitt 3.4.2 durch die Abwesenheit einer Festlegung beziehungsspezifischer Rollen zur einheitlichen Strukturierung des Systems charakterisiert. Dass objektbasierte Petrinetze zur Modellierung dieser Beziehungsart in der Lage sind, ist selbstverständlich. Die Objektbeziehung soll im Folgenden dennoch untersucht werden, da die in MagicBeans (siehe Unterabschnitt 3.3.4) vorgefundene Variante der Objektbeziehung mit Proxy-Objekten einer Modellierung und vergleichenden Diskussion bedarf.

**Erweiterungsbeziehung** Die Erweiterungsbeziehung ist bereits in Unterabschnitt 4.1.3 definiert worden und dient als Grundlage für das referenznetzbasierte Modell des Plugin-Konzepts (siehe Abschnitt 4.3). Eine weitere Diskussion der Modellierbarkeit dieser Beziehung in diesem Abschnitt erübrigt sich aufgrund der bereits erfolgten detaillierten Aufbereitung in vorigen Abschnitten.

**Dienstbeziehung** Die Dienstbeziehung ist bereits in Unterabschnitt 4.1.3 definiert worden. In Unterabschnitt 4.3.8.1 wurde die Dienstbeziehung in einer möglichen Variante in das Plugin-Konzept integriert. Wie in dem Unterabschnitt bereits erläutert, ginge die Modellierung aller Variationen der Dienstbeziehung über den Rahmen dieser Arbeit hinaus, wenn sie in derselben Detailtiefe wie bei der Betrachtung der intendierten Erweiterbarkeit durchgeführt würde. Daher soll auch in diesem Abschnitt nicht weiter auf die möglichen Modellierungsvarianten dieser Beziehung eingegangen werden.

**statische Abhängigkeit** Eine statische Abhängigkeit kann zwischen Plugins – wie in Unterabschnitt 3.4.2 erläutert – deswegen definiert werden, weil Plugin-Systeme notwendigerweise auf einer technischen Modulbeziehung aufbauen. Da auch in

der Modulbeziehung durch die Deklaration von genutzten Modul- und Prozedurbezeichnern eine Abhängigkeit geschaffen wird, unterscheidet sich die Beziehung zweier Systemkomponenten, zwischen denen eine statische Abhängigkeit besteht, nicht von der Modulbeziehung. In der folgenden Diskussion wird daher die statische Abhängigkeit gleichzeitig mit der Modulbeziehung betrachtet werden.

**funktionale Abhängigkeit** Eine funktionale Abhängigkeit ist eine Möglichkeit, Dienstbeziehungen deklarativ zu spezifizieren (siehe Unterabschnitt 3.4.2). Welche Systemkomponente die in der funktionalen Abhängigkeit deklarierten Funktionalität bereitstellt, ist beliebig, solange die bereitgestellte Funktionalität den geforderten Umfang hat. Diese Beziehung entspricht exakt einer Dienstbeziehung, in der ein Dienstnutzer ohne passenden Dienstanbieter nicht fortfahren kann. Die Identität des Dienstanbieters ist jedoch beliebig. In Unterabschnitt 4.3.8.1 wurde bereits die Möglichkeit diskutiert, Dienstbeziehungen in einem atomaren Konfigurationsschritt durch die zentrale Verwaltung auf- und abzubauen. Die dabei benötigte Beschreibung benötigter Dienste einer Systemkomponente entspricht genau der Deklaration einer funktionalen Abhängigkeit. Damit bedarf diese Beziehungsvariante keiner weiteren Modellierung mehr.

**Kopplung mittels Pfadreferenzen** Der Beziehungsaufbau mittels Pfadreferenzen wurde in Unterabschnitt 3.4.2 als ein Spezialfall der Nutzung von Namensräumen erkannt. Namensräume zur Definition von Integrationspunkten liegen prinzipiell allen Plugin-Systemen zugrunde und werden auch im konzeptionellen Modell schon zur Identifikation von Dienst- und Erweiterungsspezifikationen eingesetzt. Da aber zwei der betrachteten Plugin-Systeme, namentlich Mozilla-Chrome und NetBeans, auf die Einbettung der Pfadnamen in ein festes Erweiterungs- oder Dienstbeziehungsschema verzichten, ist zu untersuchen, ob und wie die ausschließliche Kopplung mittels Pfadreferenzen im modellierten Plugin-Konzept berücksichtigt werden kann oder sollte.

Nach dieser Vorbetrachtung gilt es, in den folgenden drei Unterabschnitten also die noch nicht diskutierten Beziehungsarten auf ihre Ähnlichkeit untereinander und ihre Modellierbarkeit im referenznetzbasierten Plugin-Konzept zu untersuchen: Modulbeziehung (statische Abhängigkeit), Objektbeziehung und Pfadreferenzen.

#### 4.4.1.1 Modulbeziehung (statische Abhängigkeit)

Die technische Modulbeziehung stellt eine seit vielen Jahren in der Softwareentwicklung bewährte Technik zur Zerlegung von Softwaresystemen in separat übersetzbare Teile bereit, welche auch die Wiederverwendung von Modulen erlaubt. Das Laden und Integrieren von Modulen zur Laufzeit erlaubt die dynamische Verknüpfung von Code entlang üblicherweise statisch vorgegebener Abhängigkeiten.

In der klassischen Modulbeziehung treffen zwei Rollen aufeinander: Das *bereitstellende* Modul (*Provider*) hält eine Menge von Prozeduren zur Nutzung durch andere Module vor. Das *aufrufende* Modul (*Caller*) enthält fest verdrahtet den Namen des bereitstellenden Moduls und der Prozeduren, die genutzt werden sollen. Da aus der

von Parnas in [Par72] vorangetriebenen Idee der hochgradig kohäsiven, abhängigkeitsarmen Modularisierung inzwischen das Konzept der Klassen im objektorientierten Softwareentwicklungsparadigma hervorgegangen ist, kann im Rahmen dieser Arbeit der Modulbegriff auf einer niedrigeren Ebene verwendet werden. Es können innerhalb eines Moduls auch Klassen oder andere komplexe Elemente gruppiert werden. Szyperski grenzt in [Szy02, Abschnitt 4.4, S. 39f.] Module durch die Abwesenheit von ressourcenbasierten Parametrisierungsmöglichkeiten und die Möglichkeit eines Modul-eigenen Zustandes gegen Komponenten ab.

Da das bereitstellende Modul im aufrufenden fest verdrahtet, also nicht ohne Weiteres austauschbar ist, spricht man auch von einer *statischen Abhängigkeit*. Beide Module können unabhängig voneinander übersetzt werden, sofern das abhängige Modul gegen eine Schnittstellenbeschreibung übersetzt wird.[23] Die Verbindung zwischen beiden Beteiligten wird zur Laufzeit durch einen externen Modul-Lademechanismus hergestellt, der Teil der Ausführungsumgebung ist (wie z. B. die Lademechanismen der Betriebssysteme für dynamische Bibliotheken, siehe Unterabschnitt 3.1.1.2, oder der Java-Klassen-Lademechanismus, siehe Unterabschnitt 3.2.1.2). Der Lademechanismus hat keine anwendungsspezifische Funktionalität, ist allerdings in der Regel derselbe, welcher die Anwendung selbst (also ihr „Hauptmodul“) lädt und startet.

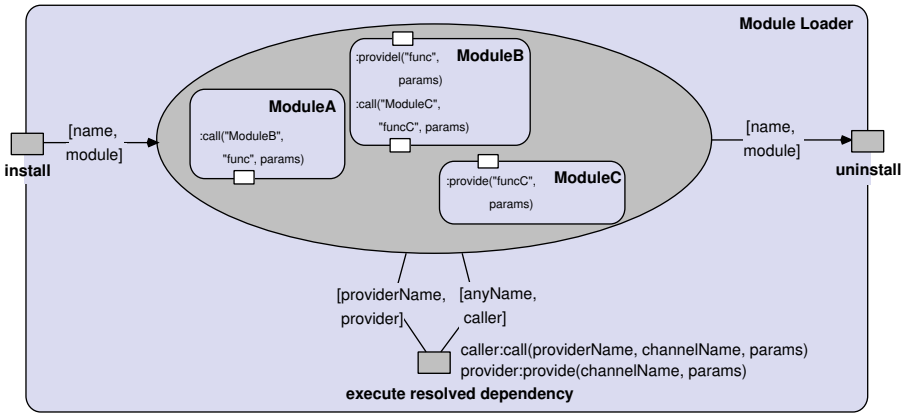



Abbildung 4.21: Einfaches Modell der Modulbeziehung

Im Paradigma der *Netze-in-Netzen* ist ein Modul-Lademechanismus recht einfach modellierbar, es muss lediglich die dynamische Kopplung separater Code-Module ermöglicht werden. In Abbildung 4.21 ist eine Modulverwaltung mit drei exemplarischen installierten Modulen dargestellt. Sämtliche Marken sind Paare aus einem eindeutigen Modulnamen (im Bild jeweils fett dargestellt) und dem eigentlichen Modul (Netz), um die Abhängigkeit anhand der Modulnamen auflösen zu können. Die dynamische Kopplung zweier Module wird im Netz durch die unterste Transition ermöglicht. Der Modul-Lader agiert hier als Vermittler, der je einen synchronen Kanal zu Aufrufer

[23] Die durch die Veränderlichkeit von Schnittstellenbeschreibungen entstehenden technischen Probleme sollen hier nicht weiter interessieren, sie wurden bereits in Unterabschnitt 3.3.8 angesprochen.

(*Caller*) und Bereitsteller (*Provider*) etabliert und dabei die Parameter bidirektional zwischen den Kanälen unifiziert.

Die Signatur eines Prozeduraufrufs wird durch den Namen des aufzurufenden Moduls und den Namen der aufzurufenden Prozedur eindeutig bestimmt, zudem können natürlich anwendungsspezifische Informationen (Parameter) ausgetauscht werden. Die weißen Transitionen der Module stellen die passenden Kanalendpunkte dar, welche jeweils eindeutig als Aufruf (`:call`) oder Bereitstellung (`:provide`) einzuordnen sind. Die Aufrufsignatur wird durch die Kanalparameter repräsentiert – im Falle des Aufrufers handelt es sich um drei Parameter (Zielmodul, Prozedurname, Parameter), im Falle des Bereitstellers sind nur zwei der Parameter nötig. Jedes Modul kann sowohl in der Rolle eines Aufrufers als auch eines Bereitstellers agieren (wie im Bild „ModuleB“), sofern dies im Modulcode vorgesehen ist.

Die Netze in Abbildung 4.21 stellen eine Verfeinerung des in Unterabschnitt 4.3.1 präsentierten Grundmodells erweiterbarer Systeme dar. Das äußere Netz enthält allerdings keinerlei anwendungsspezifische Funktionalität, sondern beschränkt sich ganz und gar auf die Verwaltung der Beziehungen und die Bereitstellung einer Kommunikationsinfrastruktur zwischen den Modulen – es liegt also eine flache Organisationsform ähnlich der in Unterabschnitt 4.3.8.1 diskutierten Dienstbeziehung vor. Die Ähnlichkeiten zwischen beiden Beziehungsarten gehen noch weiter, wenn man den Modulnamen als sehr primitive, nicht-parametrisierbare Dienstbeschreibung auffasst, zu der es nur einen einzigen Anbieter gibt. Dann agiert das abhängige Modul als Dienstnutzer und das bereitstellende Modul als Dienstanbieter, der gleichzeitig die Dienstspezifikation definiert. Da es nur einen einzigen Dienstanbieter je Dienstbeschreibung gibt, kann auf ein Diensteverzeichnis verzichtet werden. Stattdessen existiert aber mit der Modulverwaltung eine sehr ähnliche Funktionalität, die zu einem Modulnamen das bereitstellende Modul liefert. In der Tat kann man sich vorstellen, dass durch die Aufhebung der Eindeutigkeitsanforderung an Modulnamen eine Dienstbeziehung mit austauschbaren Anbietern entsteht, welche unter demselben Modulnamen firmieren.

Im Plugin-Konzept, das in Abschnitt 4.3 modelliert wurde, ist bereits die Dienstbeziehung eingearbeitet. Da die Dienstbeziehung nun als eine flexiblere Variante der Modulbeziehung betrachtet werden kann, muss die Modulbeziehung nicht separat im Konzept modelliert werden. Die detaillierte Betrachtung der Modulbeziehung bzw. der statischen Abhängigkeiten ist dennoch interessant, da sie eine unabdingbare technische Grundlage aller anderen diskutierten Beziehungsarten darstellt. Erst die technische Einrichtung des Modulladers ermöglicht die dynamische Komposition des Softwaresystems (entlang den architektonischen Vorgaben der jeweiligen Beziehungsarten) vor Ort beim Anwender. Da der Modul-Lademechanismus in der Regel nicht mehr Teil der zu entwickelnden Software ist, sondern eine Ebene tiefer bereitgestellt wird, findet er in den anderen Plugin-Systemen trotz seiner Verwendung kaum noch Beachtung. Auch in dem in dieser Arbeit entwickelten referenznetzbasierten Plugin-Konzept wurde vom Modul-Lademechanismus abstrahiert (siehe Unterabschnitt 4.2.6.1).

Zur Verdeutlichung des engen Zusammenhangs sei hier noch einmal diskutiert, inwiefern Dienst- und Erweiterungsbeziehung auf statische Abhängigkeiten angewiesen sind. In der Dienstbeziehung besteht eine statische Abhängigkeit sowohl des Dienstnutzers als auch des Dienstanbieters zur Dienstspezifikation, sofern diese als Code

in einem separaten Modul implementiert wird. Durch diese Konstruktion bestehen zwischen Dienstanbieter und -nutzer keine direkten statischen Abhängigkeiten mehr.

Die Erweiterungsbeziehung beinhaltet eine spezielle statische Abhängigkeit vom Code der Erweiterung zum Code des erweiterbaren Systems, nicht jedoch in Gegenrichtung. Dies liegt offensichtlich daran, dass die für die Beziehung elementare Schnittstellenspezifikation vom Kontext, also dem erweiterbaren System, vorgegeben wird. Zudem stellt der Kontext der Erweiterung den Rahmen zum Erbringen ihrer Funktionalität bereit. Trotzdem nimmt in der Erweiterungsbeziehung der Kontext die Rolle des Aufrufers und die Erweiterung die Rolle des Bereitstellers im Sinne der Modulbeziehung ein. Diese Nutzung in Gegenrichtung zur Abhängigkeit ist nur möglich, weil das erweiterbare System eine dynamische Verwaltung von Erweiterungsmodulen beinhaltet, statt den Namen eines bereitstellenden Moduls statisch festzuschreiben. Dieser Trick wird insbesondere bei nicht-rekursiven Plugin-Systemen wie z. B. der NPAPI deutlich, da dort die Anwendung selber für die Verwaltung der erweiternden Code-Module verantwortlich ist. Da die Anwendung so ihre „Abhängigkeit“ nach eigenem Bedarf bestimmen kann, ist der Begriff „statische Abhängigkeit“ im Kontext intendiert erweiterbarer Plugin-Systeme in dieser Richtung nicht mehr angebracht.

Die Anwendung erbringt natürlich ohne eine bestimmte Erweiterung eine andere Funktionalität als mit dieser. Betrachtete man also die Funktionalität des Gesamtsystems, so würde ohne die Erweiterung ein Teil der Funktionalität nicht erbringbar sein – hier besteht im gewissen Sinne statt der statischen eine flexible funktionale Abhängigkeit. Um auch ohne Erweiterung funktionstüchtig zu sein, muss der Kontext jene Transitionen umgehen können, die die funktionale Abhängigkeit beinhalten, wie bereits im Unterabschnitt 4.3.2 erläutert wurde. Alternativ kann das erweiterbare System entlang der Idee der *Fair Play Rule* [GB03, S. 66] eine Standard-Erweiterung vorsehen, so dass auf jeden Fall eine Erweiterung zum Aktivieren der Transitionen an der Erweiterungsschnittstelle zur Verfügung steht.

#### 4.4.1.2 Objektbeziehung

Wie bereits in der einleitenden Begriffsklärung dieses Abschnitts (und auch im Unterabschnitt 3.4.2) erläutert, steht der Begriff *Objektbeziehung* in erster Linie für die Abwesenheit architekturprägender Beziehungsvorgaben abseits vom Geheimnisprinzip und der engen Kopplung von Daten und Funktionen innerhalb eines Objekts. Alle im Rahmen der objektorientierten Softwareentwicklung modellierbaren Beziehungen können in einem auf der Objektbeziehung aufbauenden System genutzt werden, für jede Klasse können dabei eigene Regeln zur Verwaltung ihrer Beziehungspartner festgelegt werden. Die Rollen der an einer Beziehung beteiligten Objekte sind für jedes Paar individuell definierbar.

Der gemeinsame Nenner der Beziehungen in einem derartigen System umfasst folgende vier Punkte:

- Beliebig viele Objekte können eine Referenz auf ein- und dasselbe Objekt besitzen.
- Kommunikation zwischen Objekten ist möglich, sobald ein Objekt eine Referenz auf ein anderes besitzt.

- Die individuelle Schnittstelle des referenzierten Objektes ist maßgeblich für die zulässige Kommunikation.
- Objektreferenzen sind kommunizier- und duplizierbar.

Die ersten drei Eigenschaften gelten ebenfalls für die bereits in Unterabschnitt 4.4.1.1 besprochene Modulbeziehung, wobei der Modulname an die Stelle der Referenz tritt. Modulnamen werden jedoch üblicherweise nicht kommuniziert (auch wenn diese Flexibilisierung nach der bereits diskutierten Einführung der Modulverwaltung für Erweiterungs- oder Dienstbeziehungen keinen großen Schritt mehr darstellt).

Die Modellierung von Objektbeziehungen ist in einem objektbasierten Formalismus wie den Referenznetzen offensichtlich einfach. Das Paradigma der *Netze-in-Netzen* stellt unter Referenzsemantik mit synchronen Kanälen genau die Primitive zur Verfügung, die zu den eben genannten Eigenschaften passen. Da keine Einschränkungen zur Verwaltung oder zur Organisation der Beziehung vorgegeben werden, kann jedes Netz die ihm bekannten Referenzen selber auf beliebige Weise verwalten.

Die Objektbeziehung bietet keine über die Möglichkeiten der objektorientierten Softwareentwicklung hinausgehenden Mittel zur Organisation oder Strukturierung des Software-Systems. Aus diesem Grund ist sie für die Betrachtung im Rahmen dieser Arbeit eigentlich nicht von Interesse. Allerdings weisen einige der in Kapitel 3 untersuchten Softwaresysteme genau eine solche unstrukturierte Verwaltung der Systemkomponenten als zentrale Architektur auf. Die in XPCOM (siehe Unterabschnitt 3.1.2.1) und MagicBeans (siehe Unterabschnitt 3.3.4) definierten Schnittstellen firmieren zwar unter der Bezeichnung Komponentenverwaltung, definieren aber nur Verwaltungsschnittstellen, welche genau die Zuordnung von Referenzen und Schnittstellen der einzelnen Objekte koordinieren.

Die beiden genannten Systeme setzen allerdings bei der Verwaltung ihrer „Komponenten“ unterschiedliche Schwerpunkte und implementieren dabei Eigenschaften, die über die bisher in dieser Arbeit modellierten Konzepte hinausgehen. XPCOM ist stärker in Richtung einer Infrastruktur für objektorientierte Systeme ausgerichtet, es verwaltet daher eine Typhierarchie aus Klassen und Schnittstellen und erlaubt die Instantiierung mehrerer Objekte jeder Klasse. Die MagicBeans-Verwaltung strebt stattdessen eine Perfektionierung der zentralen Kontrolle über die im System existierenden Objektreferenzen an, indem es nur indirekte Referenzen zwischen Komponenten zulässt, deren Lebensdauer von der zentralen Verwaltung bestimmt wird. Für beide Ansätze sollen im Folgenden kurz diskutiert werden, ob und wie sie sich in das in Abschnitt 4.3 modellierte Plugin-Konzept einfügen lassen.

**Typhierarchie und Instantiierbarkeit** Objekte sind Instanzen von Klassen, und in der Regel können zu einer Klasse beliebig viele Objekte instantiiert werden. Im objektbasierten Formalismus der Referenznetze besteht die Möglichkeit der Instantiierung notwendigerweise ebenfalls, so dass eine Instantiierung von Netzen zu Netzinstanzen problemlos modellierbar ist.

Das in Abschnitt 4.3 modellierte System erzeugt keine Systemkomponenten neu, sondern verwaltet lediglich Referenzen zwischen Systemkomponenten, welche dem System von außen hinzugefügt werden. Die Instantiierung der Systemkomponenten findet

damit außerhalb des Modells statt. Wollte man die Instantiierung von Systemkomponenten innerhalb des Modells explizit modellieren, so hätte dies keine Auswirkungen auf die Verwaltung der einmal instantiierten Systemkomponenten, es müsste lediglich Funktionalität zur parametrisierbaren Instantiierung von Systemkomponenten ergänzt werden.

Wie bereits in Unterabschnitt 2.2.1 erwähnt, sieht Szyperski einen Unterschied zwischen Objekten und Komponenten in der fehlenden Instantiierbarkeit der Komponenten (als Ganzes). Die Übertragbarkeit dieser Sichtweise auf Plugins wurde ebenfalls mehrfach im Laufe dieser Arbeit angesprochen. Wie im Unterabschnitt 4.2.4 klargestellt wurde, wird ein Plugin innerhalb des Softwaresystems weder kopiert noch mehrfach instantiiert, so dass die Unterscheidung zwischen Netz und Netzinstanz im Rahmen der Modellierung nicht benötigt wird.

Ein wesentliches Element objektorientierter Modellierung und Implementierung ist die Definition einer Typ- und/oder Klassenhierarchie. Der Referenznetzformalismus unterstützt lediglich die objektbasierte Modellierung und Implementierung, daher fehlt genau dieses Element in Bezug auf Netze. Wie bereits in Unterabschnitt 4.2.5.4 angedeutet, lässt sich aber eine Weiterentwicklung des Referenznetzformalismus in dieser Richtung denken, so dass die fehlende technische Möglichkeit nicht gleichzeitig als konzeptionelle Beschränkung auffallen muss.

**Indirekte, zentral kontrollierte Referenzen** Ein Problem des Beziehungsaufbaus per Referenzübergabe ist die fehlende Möglichkeit, einmal übergebene Referenzen zurückzufordern bzw. deren Rückgabe zu erzwingen. MagicBeans löst dieses Problem, indem es keine direkten Referenzen zwischen Objekten verschiedener Plugins kommuniziert, sondern Proxy-Objekte zwischenschaltet. So hat eine zentrale Verwaltung die Möglichkeit, die Lebensdauer der Proxy-Objekte und damit der dadurch repräsentierten Referenzen zentral zu kontrollieren. Diese Technik ist in Abbildung 4.22 in Form von Referenznetzen illustriert. In der Mitte der unteren Reihe von Netzinstanzen ist ein Proxy-Objekt dargestellt, dass sich Objekt A gegenüber wie Objekt B verhält – es bietet den Kanal `:func(params)` an. Solange die Lebensdauer der Referenz noch nicht abgelaufen ist, wird der Kanal direkt verkettet mit einem zweiten, der dieselbe Anfrage an das echte Objekt B weiterreicht. Die grauen Transitionen dienen der Initialisierung und Terminierung des Proxy-Objekts und werden von der zentralen Verwaltung aus gesteuert. Die Rückwärtsreferenz auf Objekt A benötigt der Proxy für den im Bild nur abstrakt in Form der Anschrift „*unifyIndirectly(...)*“ dargestellten Mechanismus, der alle bei der Kanalverkettung ausgetauschten Objektreferenzen ebenfalls durch Proxy-Objekte ersetzt.

Im oberen Netz der Abbildung 4.22 sind diverse Details nicht dargestellt, so z. B. die zusätzlichen Netzelemente, welche bei der Konfiguration eines hinzugefügten oder entfernten Objekts die Menge der auf- oder abzubauenden Netzreferenzen ermitteln. Streng genommen enthält das dargestellte Netz bereits mehr Nebenläufigkeit als die eigentliche MagicBeans-Verwaltung, da letztere die *gesamte* Systemkonfiguration in einem atomaren Schritt verändert, statt nur die von den auf- oder abzubauenden Beziehungen betroffenen Objekte zu diesem Zweck zu reservieren. Die Petrinetzmodel-

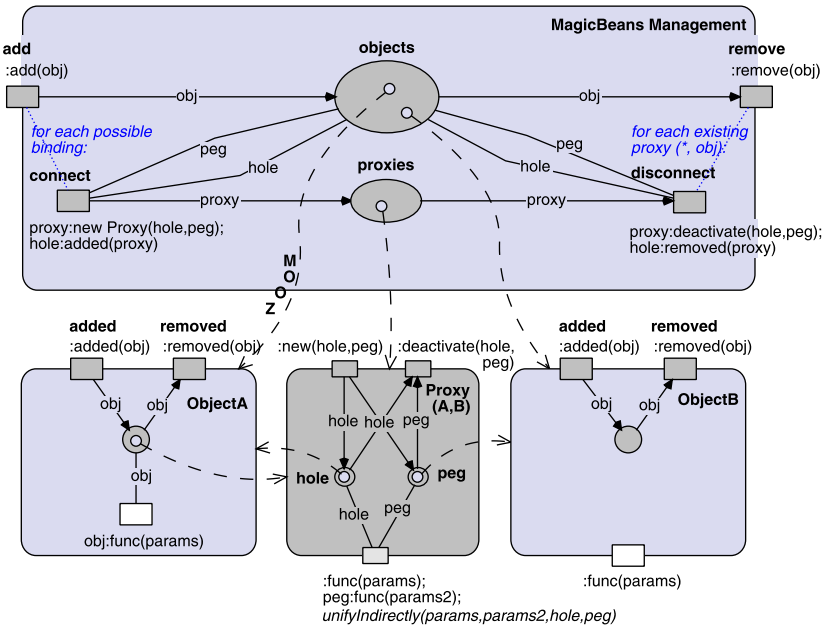



Abbildung 4.22: Extern kontrollierte Objektbeziehung (MagicBeans)

lierung der global atomaren Konfiguration wäre mittels flexibler oder Transfer-Kanten problemlos möglich.

Dieses hier im Rahmen der Objektbeziehung von MagicBeans implementierte Verfahren der Proxy-Objekte lässt sich natürlich auch auf die anderen Beziehungsarten übertragen, bei denen direkte Referenzen weitergegeben werden. Das betrifft insbesondere die Dienstbeziehung, welche in Unterabschnitt 4.3.8.1 mittels Referenzübergabe modelliert wurde. Die Erweiterungsbeziehung hat einen derartigen Mechanismus nicht nötig, da das erweiterbare System keine Referenzen an die Erweiterung übergeben muss und daher die Lebensdauer der Erweiterungsbeziehung selber steuern kann. Die Modulbeziehung, wie sie in Unterabschnitt 4.4.1.1 modelliert wurde, löst das Problem auf dem Wege der vermittelten Kommunikation, die sich von außen gesteuert unterbinden lässt. Eine alternative Modellierung der Modulbeziehung mittels Referenzübergabe hätte natürlich dieselben Probleme wie die Objekt- oder Dienstbeziehung. Anders herum lassen sich auch die beiden letzteren Beziehungen mit vermittelter Kommunikation unter einer *Netze-in-Netzen*-Semantik ohne Netzreferenzen modellieren und hätten dann keinen Bedarf für Proxy-Objekte.

#### 4.4.1.3 Kopplung mittels Pfadreferenzen

Unter einer Pfadreferenz ist die Benennung eines Elements in einem üblicherweise hierarchisch organisierten Namensraum durch zwei oder mehr ansonsten voneinander unabhängige Systemkomponenten zu verstehen. Wie bereits in Unterabschnitt 3.4.2

erläutert, benötigen sämtliche der in Kapitel 3 untersuchten Plugin-Systeme Namensräume, um die jeweils zueinander passenden Systemkomponenten beim Beziehungsaufbau ermitteln zu können. Auch im modellierten Plugin-Konzept wurde die Notwendigkeit von eindeutigen Bezeichnern für Erweiterungs- oder Dienstspezifikationen erkannt, sobald eine zentral gesteuerte Konfiguration oder ein zentrales Verzeichnis von verfügbaren Dienstanbietern vorgesehen wird (vgl. Unterabschnitte 4.3.6 und 4.3.8.1). Im Referenznetz-Modell sind unstrukturierte Zeichenketten als Bezeichner für Spezifikationen gewählt worden. Es besteht dabei natürlich die Möglichkeit, die Werte dieser Zeichenketten aus einem hierarchisch organisierten Namensraum zu wählen. Somit ist klar, dass Pfadreferenzen auch im modellierten Plugin-Konzept von Nutzen sein können.

Die beiden eingangs genannten Softwaresysteme Mozilla-Chrome und NetBeans, welche Pfadreferenzen zum Beziehungsaufbau ohne weitere Architektur-strukturierende Vorgaben nutzen, unterscheiden sich allerdings stark in den Eigenschaften der mittels Pfadreferenzen gekoppelten Systemkomponenten. Während im Mozilla-Chrome passive Datenstrukturen anhand der Pfadnamen zu einer zentralen Datenstruktur verwoben werden (siehe Unterabschnitt 3.1.2.3), welche dann zentral ausgewertet wird, eignen sich Pfadnamen in NetBeans aufgrund der angebotenen Beobachtungsmöglichkeiten zur Etablierung von Dienst- und Erweiterungsbeziehungen zwischen aktiven Systemkomponenten (siehe Unterabschnitt 3.3.2).

Wie bereits in Unterabschnitt 3.4.1 argumentiert, sind die per Pfadnamen identifizierten Overlays in der Mozilla-Chrome-Architektur aufgrund der Eigenschaft, dass die gekoppelten Systemkomponenten keinen Code enthalten, nicht als vollwertige Plugins zu werten. Die Chrome-Architektur kann eher im Rahmen der gesamten Mozilla-Architektur als eine Erweiterungsspezifikation verstanden werden, so dass jedes Overlay als Erweiterung des zentralen Chrome-Renderers, nicht aber als Erweiterung eines anderen Chrome-Dokuments betrachtet werden sollte. Unter diesem Blickwinkel ist die Integration von Overlays mittels Pfadreferenzen im Modell des Plugin-Konzepts problemlos implementierbar, da es sich lediglich um eine konkret zu definierende Erweiterungsspezifikation handelt.

Anders sieht die Situation jedoch bei den NetBeans-Modulen aus, welchen sich anhand einer Pfadreferenz im virtuellen Dateisystem nahezu beliebige Kommunikationsmöglichkeiten zu anderen Modulen eröffnen. Zunächst einmal erlaubt die Metapher des Dateisystems natürlich beliebig organisierte asynchrone Kommunikation ähnlich einem „Tuple Space“, in welcher ein Modul Informationen in einer Datei ablegt und andere Module diese Informationen lesen oder entnehmen. Die im virtuellen Dateisystem angebotenen Benachrichtigungsdienste erlauben aber auch eine synchrone Kommunikation, dank derer ein Modul unmittelbar auf Änderungen am Dateisystem reagieren kann.

Beide Kommunikationsmuster lassen sich in Referenznetzen sehr einfach nachbilden. Das bereits in Unterabschnitt 4.4.1.1 verwendete Prinzip der vermittelten Kommunikation durch ein übergeordnetes Infrastrukturnetz erlaubt synchrone Kommunikation. Der dort zur Identifikation der Kommunikationspartner verwendete Modulname kann hier durch den gemeinsam bekannten Pfadnamen ersetzt werden. Da jede Stelle eines Petrinetzes zudem im Verhalten einem „Tuple Space“ entspricht, lässt sich auch die asynchrone Kommunikation leicht nachbilden. Es braucht nur die synchron vermit-

telnde Kommunikationstransition im übergeordneten Infrastrukturnetz aufgespalten und durch eine Stelle ergänzt zu werden, in der die Nachrichten zwischengelagert statt sofort weitervermittelt werden.

## 4.4.2 Lebenszyklus und Konfiguration

In Unterabschnitt 4.3.8.2 wurde die Beteiligung der Plugins am Konfigurationsprozess prinzipiell ermöglicht, indem Lebenszyklen ins Modell des Plugin-Konzepts eingeführt wurden. Es sind aber nur zwei exemplarische, sehr einfach gestaltete Lebenszyklen betrachtet worden. In jedem in Kapitel 3 untersuchten Plugin-System lassen sich Lebenszyklus-definierende Schnittstellendeklarationen finden, die zum Teil recht unterschiedliche Zustände und Abläufe beschreiben. Es stellt sich die Frage, ob das modellierte Plugin-Konzept diese Vielfalt an Lebenszyklen darstellen kann oder ob die vielfältigen Lebenszyklen sich sogar auf die zwei in Unterabschnitt 4.3.8.2 modellierten einfachen Varianten zurückführen lassen.

Um diese Frage zu beantworten, soll zunächst die Modellierungstechnik für Lebenszyklen rekapituliert werden, welche während der Untersuchung angewendet wurde. Anschließend können die in Kapitel 3 erarbeiteten Lebenszyklusnetze überblicksweise gegenübergestellt, in ihrer Darstellung vereinheitlicht und – sofern möglich – zusammengefasst werden. Gemäß der in Unterabschnitt 4.2.7 präsentierten Technik können die aus dieser Analyse extrahierten Lebenszyklen als prozessorientierte Sicht in die funktional orientierten Strukturmodelle des Unterabschnitts 4.3 integriert werden. Dies ist bereits in Unterabschnitt 4.3.8.2 geschehen.

### 4.4.2.1 Modellierung von Lebenszyklen

Die klassische Form der Modellierung eines Lebenszyklus als Zustandsraum mit atomaren Übergängen ist ein endlicher Automat. Endliche Automaten lassen sich auf primitive Weise als S/T-Netze mit gelabelten Transitionen darstellen, in denen jeder Zustand einer Stelle und jeder Übergang einer Transition mit exakt einer Ein- und einer Ausgangskante entspricht. Die Label an den Transitionen ermöglichen es, mehrere alternative Zustandsübergänge mit derselben Bezeichung oder Aktion zu versehen, wie es in endlichen Automaten üblich ist. Die Anfangsmarkierung eines solchen Netzes besteht aus genau einer Marke in der Stelle, welche den Startzustand des endlichen Automaten verkörpert. Das resultierende Petrinetz ist 1-sicher. Abbildung 4.23 zeigt ein solches Netz schematisch (ohne Bezug auf ein konkretes Plugin-System).

Bei der Modellierung der Lebenszyklen der untersuchten Plugin-Systeme wird nach Zustandsübergängen unterschieden, welche von der verwalteten Systemkomponente wahrgenommen werden können (in Form eines von der Verwaltung an den Repräsentanten übermittelten Ereignisses) oder nicht. Zu den nicht-wahrnehmbaren Übergängen zählt beispielsweise das reine Laden von Code, ohne dass der Code dabei aufgerufen und über seine Zustandsänderung in Kenntnis gesetzt wird. Nicht-wahrnehmbare Übergänge werden wie z. B. die Transitionen „a“ und „b“ in Abbildung 4.23 blassgrau dargestellt.

In endlichen Automaten zur Modellierung von Lebenzyklen müssen gelegentlich verschiedene Übergänge dasselbe Symbol bzw. dieselbe Aktion tragen, falls ein Übergang

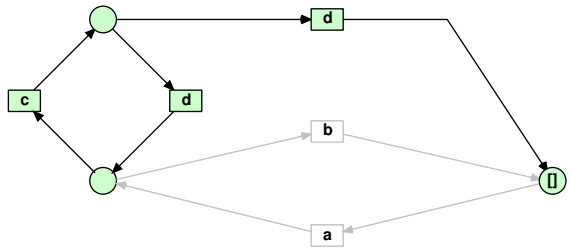



Abbildung 4.23: Modellierungselemente eines Lebenszyklusnetzes für *eine* Systemkomponente

aus mehreren Zuständen heraus möglich sein soll oder wenn verschiedene Übergänge der verwalteten Systemkomponente als ununterscheidbare Ereignisse mitgeteilt werden. Im S/T-Netz tragen in so einem Fall die Transitionen denselben Label. In Abbildung 4.23 tragen beispielsweise zwei Transitionen den Label „d“. Dieser Label modelliert beispielsweise den Umstand, dass die Verwaltung dem Plugin nur die Bezeichnung des Übergangs mitteilt, nicht jedoch den exakten Ausgangs- und Folgezustand des Übergangs. Dies ist im abstrakten Beispiel insofern konsequent, als dass das Plugin die durch die nicht-wahrnehmbaren Übergänge „a“ und „b“ unterschiedenen Folgezustände des Übergangs „d“ ohnehin nicht unterscheiden kann.

Diese Variante der Modellierung von Lebenszyklen benötigt eine Netzinstanz je verwalteter Systemkomponente, da nur eine einfache Marke zur Markierung des aktuellen Zustands verwendet wird. Die Technik stößt an ihre Grenzen, wenn Zustandsübergänge modelliert werden sollen, welche mehrere oder alle Systemkomponenten auf einmal betreffen oder auf komplexeren Beziehungen zwischen den Systemkomponenten basieren. Die Modellierung in Kapitel 3 erfolgt daher mit zentralisierten Lebenszyklusnetzen, in denen mehrere Marken – eine je verwalteter Systemkomponente – wandern. Die Marken müssen gefärbt sein, um die Zustände der einzelnen Systemkomponenten unterscheiden zu können. Solche Netze lassen sich zunächst als Faltung mehrerer Netzinstanzen auffassen, ergänzen aber Modellierungsmöglichkeiten, die in den Einzelnetzen nicht bestünden.

Abbildung 4.24 stellt ein solches schematisches Lebenszyklusnetz ohne Bezug zu einem konkreten Plugin-System (und auch ohne Bezug zur vorigen Abbildung 4.23) dar. Um weiterhin den Zustand jeder verwalteten Systemkomponente eindeutig bestimmen zu können, ist es wie zuvor notwendig, dass die Transitionen, welche reguläre Zustandsübergänge modellieren, keine Marken vernichten oder duplizieren. Da die Lebensdauer des Lebenszyklusnetzes nun die Lebensdauer der verwalteten Systemkomponenten übersteigt, muss es Transitionen geben, welche hinzukommende oder in das Netz als Marke einführen oder zu entfernende Systemkomponenten aus dem Netz herausnehmen. Diese Transitionen werden wie die beiden in Abbildung 4.24 als „install“ und „uninstall“ benannten farbig besonders hervorgehoben.

Eine zusätzliche Modellierungsmöglichkeit wird im Beispiel durch die Transitionen „a“ und „b“ illustriert: Zustandsübergänge, welche alle verwalteten Systemkomponen-

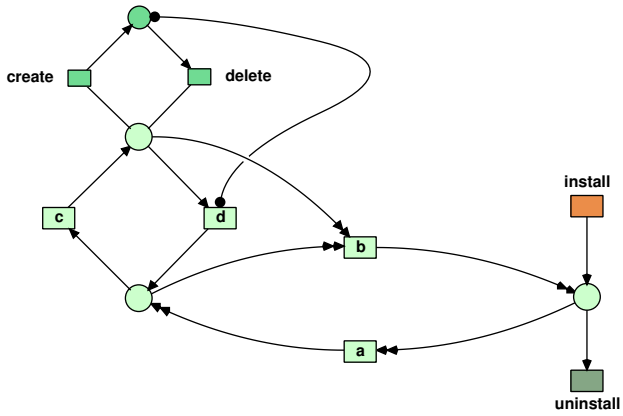



Abbildung 4.24: Modellierungselemente eines Lebenszyklusnetzes für *mehrere* Systemkomponenten

ten betreffen, können atomar ausgeführt werden, indem mittels flexibler Kanten mehrere Marken zeitgleich in einem Schaltvorgang bewegt werden. Dieses Verhalten entspricht einer Transfer-Kante. Die flexiblen Kanten können Transfer-Kanten emulieren, wenn zusätzliche Stellen zur Buchführung über die vorhandenen Marken neben die eigentlichen Stellen gelegt werden. Diese Stellen sind aber der Übersichtlichkeit halber nicht im Netz dargestellt, da sie etliche Elemente (Stellen und Kanten) duplizieren. Die Verwendung flexibler Kanten erlaubt es auch, statt sämtlicher Marken nur eine Teilmenge der verfügbaren Marken anhand konkret festzulegender Kriterien zu bestimmen und somit einen Zustandsübergang nur für diese Teilmenge wirksam werden zu lassen. Im Falle der Transition „b“ im Beispiel findet umgekehrt eine Zusammenführung der Marken zweier verschiedener Stellen statt.

Eine weitere im Beispiel dargestellte Modellierungsmöglichkeit betrifft die Darstellung von Nebenbedingungen für Zustandsübergänge, welche von der Zahl existierender Instanzen bestimmter Objekte einer verwalteten Systemkomponente abhängig sind. Zu betonen ist hierbei, dass die zur Modellierung eines solchen Zusammenhangs zusätzlich eingezeichneten Stellen und Transitionen *nicht* zum eigentlichen Lebenszyklus der verwalteten Systemkomponente gehören. Sie sind daher im Vergleich zu den Lebenszyklus-definierenden Elementen des Netzes kleiner und dunkler dargestellt.

In Abbildung 4.24 betrifft dies die Transitionen „create“ und „delete“, welche lediglich zur Modellierung einer Schaltbedingung der Transition „d“ in das Netz aufgenommen wurden. Zunächst einmal sind beide Transitionen mit Testkanten (ohne Pfeile) an den Lebenszykluszustand angebunden, in welchem sie aktiviert werden können. Testkanten vermeiden eine Einschränkung der Nebenläufigkeit der Erzeugungs- und Löschvorgänge, schließen aber dennoch ein gleichzeitiges Schalten etwa von „create“ und „d“ auf demselben Plugin aus. Die Transition „d“ kann aufgrund der Inhibitorkante nur schalten, solange keine Marken in der obersten Stelle liegen, welche mit der vom Zustandsübergang „d“ betroffenen Systemkomponente in einem Zusammenhang

stehen. Die Inhibitorkante ist also selektiv zu verstehen, so dass sie nicht auf Marken reagiert, welche zu anderen Systemkomponenten gehören.[24]

Die Möglichkeit, für die einzelnen Komponenten nicht-wahrnehmbare Übergänge zu modellieren, besteht weiterhin. Das Beispiel in Abbildung 4.24 macht allerdings keinen Gebrauch davon, da keine Transition blassgrau dargestellt ist.

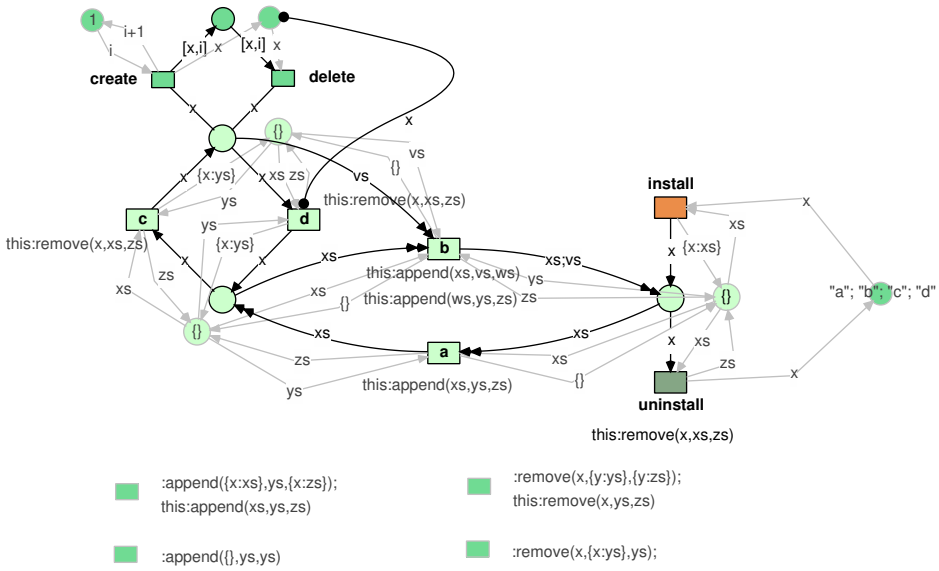



Abbildung 4.25: Operationale Vervollständigung des Netzes aus Abbildung 4.24

Abbildung 4.25 stellt die Implementierbarkeit der zuvor behaupteten Modellierungsabstraktionen unter Beweis, indem die nötigen Buchführungsstellen mit Listen als Marken und den zugehörigen Listenoperationen in funktionaler Programmierung mittels synchroner Kanalverschmelzung ausgeführt sind. Zu Testzwecken werden rechts im Bild vier Namen für konfigurierbare Systemkomponenten vergeben. Zusätzlich implementiert das Netz oben links einen Identitätszähler für erzeugte Instanzen, welcher allerdings keinen Einfluss auf die Semantik des Netzes hat.

#### 4.4.2.2 Gegenüberstellung der beobachteten Lebenszyklen

Die in Kapitel 3 betrachteten Plugin-Systeme bringen selten formalisierte Verhaltensbeschreibungen mit, stattdessen muss das dynamische Verhalten auf Basis statischer Schnittstellendeklarationen vermutet werden. Nur wenige Systeme bieten als einzige formalisierte Verhaltensspezifikation eine Beschreibung des Lebenzyklus' von Plugins.

[24] Das Renew-Handbuch weist darauf hin, dass Inhibitorkanten nicht in der Lage sind, selektiv auf Tupel zu reagieren, von denen nur einige der Komponenten relevant sind. Als Umgehungstechnik lässt sich auch hier eine Buchführungsstelle einrichten, welche nur die relevanten Komponenten als Marken enthält. [Vgl. KWD06b, Abschnitt 3.10.3]

Teil der Untersuchung existierender Plugin-Systeme war daher die Formalisierung der Plugin-Lebenszyklen in Form von Petrinetzen.

Generell ist anzumerken, dass es sich bei allen modellierten Lebenszyklen um im jeweiligen System global durchgesetzte Spezifikationen handelt. Lokale Lebenszyklen, die wie in Unterabschnitt 4.3.8.3 angesprochen nur im Kontext einer konkreten Erweiterungs- oder Dienstspezifikation gelten, waren in der Regel nicht Gegenstand der Untersuchung, da sie naturgemäß nicht in den betrachteten Rahmenwerken, sondern in individuellen erweiterbaren Systemkomponenten definiert werden.

In Abbildung 4.26 werden zunächst noch einmal die wesentlichen während der Untersuchung erarbeiteten Lebenszyklusnetze im Überblick dargestellt. Jedes der in Tabelle 3.2 angeführten Systeme mit Ausnahme von MagicBeans ist mit einem Netz vertreten. Zum MagicBeans-System (siehe Unterabschnitt 3.3.4) existiert kein Lebenszyklusnetz, da es lediglich atomare Übergänge der Konfiguration des Gesamtsystems kennt und den einzelnen Plugins keine Zustandsänderungen mitgeteilt werden. Ergänzt wird die Darstellung durch zwei weitere Lebenszyklusnetze für Dienstanbieter und -nutzer im OSGi-Rahmenwerk.

Zunächst einmal unterscheiden sich die Netze in den verwendeten Bezeichnern für Zustände und Übergänge, da diese aus den jeweiligen Begriffsgebäuden der einzelnen Plugin-Systeme stammen. Die Verschiedenheit der Lebenszyklusnetze resultiert aber darüber hinaus aus mehreren Faktoren, welche jeweils individuell bei der Modellierung der einzelnen Netze in den Unterabschnitten des Kapitels 3 besprochen wurden. Hierzu zählen die Eigenschaft der dynamischen Rekonfigurierbarkeit des Gesamtsystems, die Berücksichtigung von Zuständen des verwendeten Modullademechanismus, die erzwungene Kopplung mehrerer Lebenszyklen z. B. in Form der Existenz von Übergängen, welche mehrere Systemkomponenten zeitgleich betreffen oder der Berücksichtigung von Instanznutzungszählern, sowie die den Plugins eingeräumten Mitbestimmungsmöglichkeiten. Die genannten Faktoren sollen im Folgenden kurz diskutiert werden.

**Dynamische Rekonfigurierbarkeit** Von den betrachteten Plugin-Systemen unterstützt etwas über die Hälfte eine dynamische Rekonfiguration, so dass Plugins während der Laufzeit des Systems hinzugefügt oder entfernt werden können. Die Verwaltungen der Mozilla-Architekturen XPCOM und XPInstall, des OSGi-Rahmenwerkes, der NetBeans-Plattform und von MagicBeans bieten zur Laufzeit nutzbare Schnittstellen zum An- und Abmelden installierter Systemkomponenten. Im Gegensatz dazu ermitteln die Verwaltungen der Netscape-Plugins, des Mozilla-Chrome und der klassischen Eclipse-Plugins bei jedem Systemstart die installierte Gesamtkonfiguration und reagieren somit auf Änderungen, welche außerhalb der Laufzeit des Systems stattgefunden haben. Zu beachten ist, dass das NetBeans-System beide Varianten vereint, da seine Verzeichnisse sowohl beim Systemstart aus der vorgefundenen Installation heraus initialisiert als auch zu Laufzeit über Schnittstellen manipuliert werden. Letztlich werden auch im modernisierten Eclipse-System durch die Kombination der klassischen Plugin-Architektur mit dem OSGi-Rahmenwerk beide Verfahren verwendet. (Vgl. Diskussion in Unterabschnitt 3.4.1 sowie Tabelle 3.2)

In den in Abbildung 4.26 zusammengefassten Lebenszyklusnetzen spiegelt sich dieser Unterschied wieder. Die Systeme, welche bei Systemstart die Konfiguration ermitteln, verfügen über einen für alle Plugins zeitgleich stattfindenden Übergang. In Kontext der Netscape-Plugins (siehe Abbildung 3.2), der klassischen Eclipse-Plugins (siehe Abbildung 3.7) und der NetBeans-Module (siehe Abbildung 3.15) sind dies die Transitionen mit der Bezeichnung „platform startup“, im Kontext des Mozilla-Chrome (siehe Abbildung 3.4) weist die Transition „display“ dieses Verhalten auf. Interessanterweise ergibt sich nicht automatisch die Existenz einer Transition mit gegenteiliger Wirkung (also etwa der Bezeichnung „platform shutdown“). Im Falle des Mozilla-Chrome ist dies darauf zurückzuführen, dass die Pakete nichts über ihre Nutzung und damit auch deren Ende erfahren, da sie lediglich passive Datenstrukturen bereitstellen. Am Beispiel des OSGi-Rahmenwerkes (siehe Abbildung 3.10) lässt sich beobachten, dass auch bei explizit individueller und dynamischer Verwaltung jeder einzelnen Systemkomponente ein impliziter Übergang beim Herunterfahren der Plattform auftreten kann. Hier ist die Ursache darin zu suchen, dass im OSGi-Rahmenwerk das Ungültig-Werden von aufgelösten Abhängigkeiten nur bei Rekonfiguration eines weiterlaufenden Systems interessiert – nach einem Neustart der Plattform beginnen sämtliche Bundles standardmäßig im Zustand „installed“.

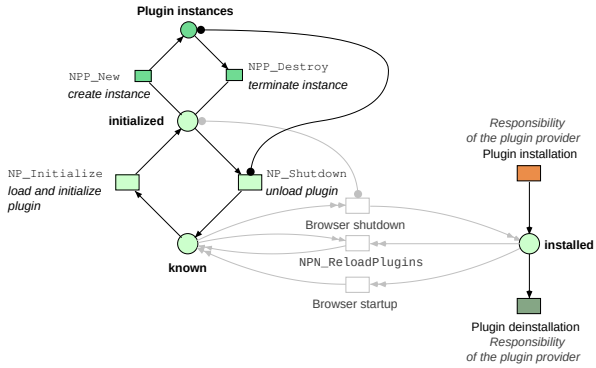
Falls wie in der NetBeans-Architektur sowohl die Installation zur Laufzeit des Systems als auch außerhalb derselben möglich sein soll, müssen im Petrinetz die Installations- und Deinstallationstransitionen mehrfach modelliert werden, wie in Abbildung 3.15 geschehen.

Wie bereits in Unterabschnitt 4.2.6.1 erläutert, wird im Rahmen der konzeptionellen Modellierung dieser Arbeit auf die explizite Betrachtung nicht-dynamischer Systeme verzichtet, da sie vom Verhalten her als Spezialfall der dynamischen Rekonfigurierbarkeit modellierbar sind. Dieser Verzicht wirkt sich auf die Modellierung der Lebenszyklusnetze vereinfachend aus, da die für die verwalteten Plugins ohnehin nicht wahrnehmbaren oben benannten Transitionen wegfallen können. Stattdessen lässt sich ein Systemstart wie eine Installation sämtlicher vorgefundener Plugins in kurzer Abfolge betrachten.

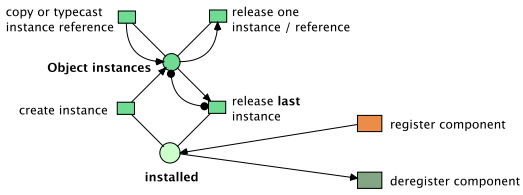
Zu beachten ist, dass das Mozilla-Chrome nach dieser Änderung auf einen trivialen Lebenszyklus reduziert ist. Dies rührt daher, dass das Mozilla-Chrome alleine – wie bereits in Unterabschnitt 3.4.1 diskutiert – kein vollwertiges Plugin-System darstellt, da dort lediglich passive Datenstrukturen kombiniert werden.

**Modulladememechanismen** Wie bereits in Unterabschnitt 4.4.1 einleitend angemerkt, sind Modulbeziehungen in weiter entwickelten Plugin-Systemen nicht mehr Gegenstand der Diskussion. Ihre technische Umsetzung geschieht für die Softwarearchitektur transparent durch das Betriebssystem oder eine programmiersprachenspezifische Ausführungsumgebung.

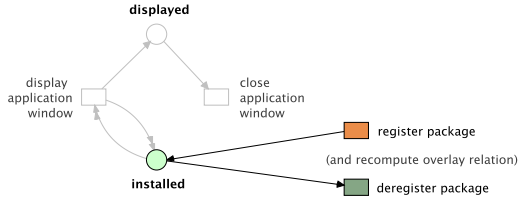
Da allerdings – wie am Beispiel der Netscape-Plugins in Unterabschnitt 3.1.1 erläutert – die eigentlich auf dauerhaft feststehende Beziehungen zwischen Modulen ausgelegten Modulladememechanismen im Kontext von Plugin-Systemen mit flexiblen Konfiguration konfrontiert werden, ist in der Verwaltung des Plugin-Systems eine flexible Steuerung des zugrunde liegenden Modulladers erforderlich. Alle untersuchten



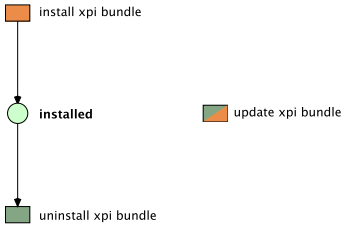

(a) Abb. 3.2 (S. 119): Netscape-Plugins



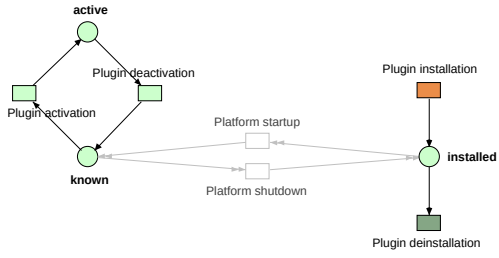

(b) Abb. 3.3 (S. 125): Mozilla-XPCOM




(c) Abb. 3.4 (S. 128): Mozilla-Chrome

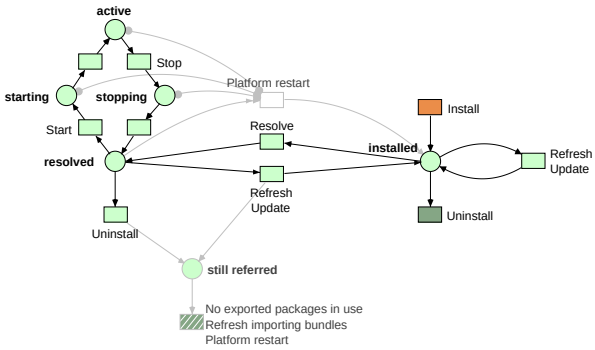



(d) Abb. 3.5 (S. 132): XPInstall

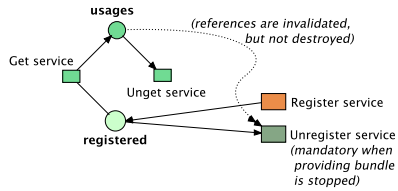



(e) Abb. 3.7 (S. 141): Eclipse-Plugins

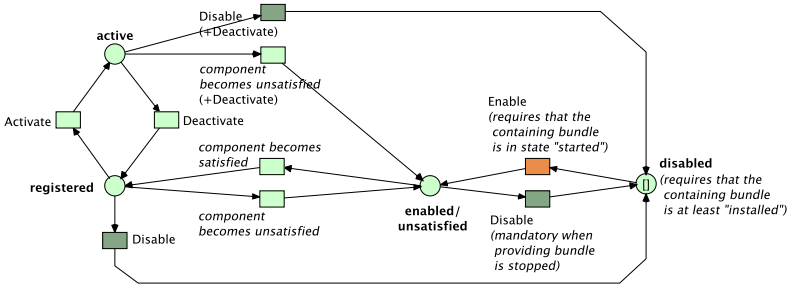
Abbildung 4.26: Plugin-Lebenszyklusnetze im Überblick



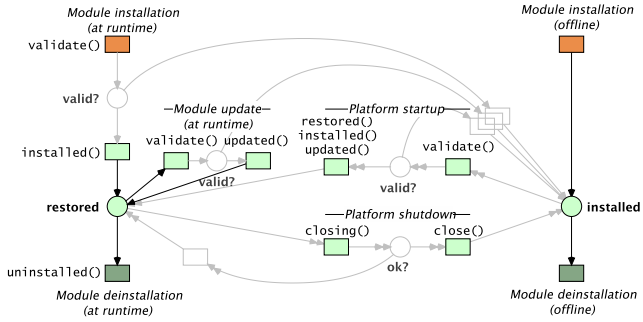

(f) Abb. 3.10 (S. 145): OSGi-Bundles




(g) Abb. 3.11 (S. 147): OSGi-Services




(h) Abb. 3.13 (S. 150): OSGi-Service-Components




(i) Abb. 3.15 (S. 160): NetBeans-Module

Plugin-Systeme arbeiten daher bei der Verwaltung ihrer Plugins mit dem Konzept statischer Abhängigkeiten, sei es nur zwischen Plugin und Kernanwendung oder auch zwischen individuellen Plugins.

Die Existenz von statischen Abhängigkeiten kann, muss aber nicht im Lebenszyklusnetz sichtbar werden. Einige Plugin-Systeme integrieren die Zustände des Modullademechanismus' in den Zustandsraum ihrer eigenen Verwaltung, um den Modullader effizient steuern zu können. Dies führt zur Unterscheidung von Zuständen wie z. B. „installed“ und „resolved“ im OSGi-Rahmenwerk (siehe Abbildung 3.10). Allerdings bekommen in der Regel die verwalteten Systemkomponenten ihre eigene Zulässigkeitsprüfung und Abhängigkeitsauflösung nicht mit, ihr Code wird z. B. im OSGi-Rahmenwerk erstmals beim Übergang „start“ aufgerufen.

Andere Systeme abstrahieren gänzlich von diesen Zuständen und betrachten lediglich solche Systemkomponenten, deren Abhängigkeiten bereits aufgelöst sind – ein Plugin mit unerfüllten Abhängigkeiten gilt dann als nicht-installiert, bzw. kann die Hinzufüge-Transition nicht passieren. Beispiele hierfür sind die Mozilla-Subsysteme XPCOM (siehe Abbildung 3.3), Chrome (siehe Abbildung 3.4) und XPInstall (siehe Abbildung 3.5) sowie das MagicBeans-System (siehe Unterabschnitt 3.3.4), welches auch aus diesem Grund ohne globalen Lebenszyklus auskommt.

Die Systeme, deren Lebenszyklen – wie im vorigen Unterabschnitt besprochen – aufgrund der zentralen Konfiguration beim Systemstart nicht-wahrnehmbare Übergänge enthalten, lassen sich nicht direkt anhand der Darstellung in Abbildung 4.26 einordnen. Allen gemeinsam ist notwendigerweise, dass Systemkomponenten ohne eine Abhängigkeitsprüfung installiert werden können, da dieser Vorgang außerhalb der Kontrolle der Pluginverwaltung stattfindet. Der zentrale Konfigurationsübergang beim Systemstart kann, muss aber nicht das Auflösen statischer Abhängigkeiten beinhalten. So werden Netscape-Plugins (siehe Abbildung 3.2) erst bei Bedarf, d.h. beim Übergang „initialize“, geladen, so dass nicht-ladbare Module diesen Zustandsübergang verhindern können. Klassische Eclipse-Plugins (siehe Abbildung 3.7) werden zwar ebenfalls erst bei erstmaliger Aktivierung geladen, aber die Eclipse-Pluginverwaltung prüft bereits beim Systemstart die separat deklarierten Abhängigkeiten auf Erfüllbarkeit. Die NetBeans-Verwaltung prüft ebenfalls schon im Rahmen der zweischrittigen Konfigurationsübergänge (siehe Abbildung 3.15) die Erfüllbarkeit von Abhängigkeiten.

Die beiden für Dienstbeziehungen definierten Lebenszyklusnetze aus dem OSGi-Rahmenwerk unterscheiden sich ebenfalls in der Berücksichtigung der Zustände des Modullademechanismus. Während die einfache Dienstverwaltung mit dem Netz aus Abbildung 3.11 nur Dienstanbieter betrachtet, welche bereits geladen sind, unterstützt die komplexere Service Component Runtime (SCR) mit dem in Abbildung 3.13 dargestellten Netz auch Dienstanbieter, deren Bundle-Abhängigkeiten noch nicht aufgelöst wurden. Zu beachten ist, dass der Zustand „unsatisfied“ im SCR-Lebenszyklus die Nichterfülltheit *funktionaler* Abhängigkeiten modelliert und daher *nicht* mit dem Modullademechanismus zusammenhängt.

Wie bereits in Unterabschnitt 4.2.6.1 argumentiert, wurde während der Modellierung in Unterabschnitt 4.3.8.2 die Unterscheidung von Zuständen des Modulladers vernachlässigt. Hintergrund dieser Entscheidung ist, dass in der Modellierung nur solche Übergänge von Interesse sind, welche die Plugins wahrnehmen können. Der Lebenszyklus dient im modellierten Plugin-Konzept im Wesentlichen als Schnittstel-

lenvereinbarung zwischen Verwaltung und Systemkomponente, die anderen Zustände und Übergänge sind lediglich verwaltungsintern von Interesse und wirken sich daher nicht architekturprägend aus.

Das Vernachlässigen von Zuständen, welche nur für den Modullader relevant sind, wirkt sich bei den zuvor genannten Lebenszyklusnetzen kaum aus. Die Lebenszyklen der Netscape- und der NetBeans-Plugins sowie der OSGi-Service-Components ändern sich durch diese Vereinfachung nicht, da die entsprechenden Übergänge neben dem Laden des Moduls noch andere Aufgaben im System benennen. Mit dem im vorigen Unterabschnitt besprochenen Wegfall der Zustände, welche nur bei nicht-dynamischer Konfiguration relevant sind, fällt bei klassischen Eclipse-Plugins sowie beim Mozilla-Chrome auch die Unterscheidung nach erfolgreicher Abhängigkeitsprüfung fort. Ein sichtbarer Unterschied tritt nur beim Lebenszyklus der OSGi-Bundles auf, hier fiele mit den Stellen „installed“ und „still referred“ die rechte Hälfte des Netzes quasi weg.

**Kopplung mehrerer Lebenszyklen** Einige der in Abbildung 4.26 dargestellten Lebenszyklusnetze nutzen die in Unterabschnitt 4.4.2.1 vorgeschlagene Technik zur Integration von Nebenbedingungen wie z. B. Instanzzählern in das Lebenszyklusnetz. De facto findet damit die Modellierung mehrerer Lebenszyklen in einem Netz statt, welche sich auf unterschiedliche Systemkomponenten beziehen. Diese Verbindung von Lebenszyklen ist dann sinnvoll, wenn zwischen den Lebenszyklen der verschiedenen Systemkomponenten eine enge Kopplung besteht.

Netscape-Plugins (siehe Abbildung 3.2) geben auf diese Weise einen Rahmen für den Lebenszyklus der Instanzen zur Präsentation eines konkreten Web-Elements vor, Komponenten in der Mozilla-XPCOM-Architektur (siehe Abbildung 3.3) koppeln so den Lebenszyklus von Objekten an den der jeweils zugrundeliegenden Klasse, und im Kontext der einfachen OSGi-Services (siehe Abbildung 3.11) wird die Dauer von Dienstnutzungsbeziehungen in Bezug zum Lebenszyklus des Dienstanbieters gesetzt.

Die Kopplung von Lebenszyklen kann auch anders modelliert werden, wie an den drei Lebenszyklusnetzen des OSGi-Rahmenwerkes zu sehen ist. Einfache OSGi-Dienste können nur von aktiven Bundles (wie im Bundle-Lebenszyklus in Abbildung 3.10 definiert) registriert werden. Die Lebenszyklen von OSGi-Service-Components (siehe Abbildung 3.13) sind hingegen auf komplexere Weise mit dem der OSGi-Bundles verzahnt, wie die Kommentare an den Transitionen „Enable“ und „Disable“ belegen.

Im Laufe der Modellierung des Plugin-Konzepts wurde in Unterabschnitt 4.3.8.3 bereits angedeutet, dass die Kopplung mehrerer Lebenszyklen in einer Systemkomponente, welche zu mehreren Erweiterungs- oder Dienstspezifikationen konform entworfen wird, grundsätzlich möglich ist. Die hier diskutierte Kopplung ist von leicht anderer Natur, da eine hierarchisch organisierte Beziehung zwischen „großer“ Systemkomponente und der untergeordneten Instanz besteht. Diese Art von Kopplung lässt sich ähnlich der in Unterabschnitt 4.3.4 und Unterabschnitt 4.3.8.3 modellierten mehrschichtigen Verwaltung herstellen, indem jede Systemkomponente zum globalen Lebenszyklus konform implementiert wird und gleichzeitig ihre Instanzen in einem lokalen Lebenszyklus steuert. Die Kopplung der Lebenszyklen wird dann in der Systemkomponente durch Verbindungen zwischen äußerer und innerer Schnittstelle realisiert.

Die Lebenszyklusnetze können dann für jede Verwaltungsebene vereinfacht definiert werden.

**Mitbestimmung** In Unterabschnitt 4.3.8.4 wurde bereits darauf hingewiesen, dass ein in einer Schnittstellenspezifikation festgeschriebener synchroner Kanal als Teil der Lebenszyklusdefinition auf verschiedene Weisen genutzt werden kann. Eine in den untersuchten Plugin-Systemen eher selten explizit deklarierte, aber fast immer vorhandene Nutzungsweise besteht darin, das hinzuzufügende oder zu entfernende Plugin auf seinen Konfigurationsprozess Einfluss nehmen zu lassen, so dass es ihn beispielsweise abbrechen kann.

Von den untersuchten Plugin-Systemen wurde diese Möglichkeit vor allem im Kontext der NetBeans-Plattform diskutiert, welche alle Zustandsübergänge in zwei Phasen unterteilt (siehe Abbildung 3.15). Der erste Teilübergang namens `validate()` sieht prinzipiell die Möglichkeit vor, dass ein Plugin seiner Integration in das System widersprechen kann. Der jeweils zweite Teilübergang hat nur noch informierenden Charakter, ein fehlerfrei arbeitendes Plugin darf diesen Übergang nicht mehr verhindern.

Alle anderen untersuchten Plugin-Systeme reagieren stattdessen auf die je nach System mehr oder weniger explizit eingeplante Möglichkeit, dass eine im Rahmen des Lebenszyklus aufgerufene Methode scheitert, also beispielsweise bei Java-Programmen mit einer `Exception` zurückkehrt anstelle normal zu terminieren. Die meisten der in Abbildung 4.26 zusammengefassten Lebenszyklen spiegeln diese Möglichkeit nicht wieder, in Petrinetzsemantik muss man sich eine gescheiterte Aktivierung also wie einen mangels Bindung nicht zustande gekommenen Schaltvorgang vorstellen.

Einzig der OSGi-Bundle-Lebenszyklus (siehe Abbildung 3.10) berücksichtigt ähnlich wie der NetBeans-Lebenszyklus eine zweiphasige Aktivierung und Deaktivierung. Diese bildet jedoch in erster Linie die Zeitbehaftetheit des Zustandsübergangs ab, ein Misserfolg des zweiten Übergangs ist nach Schalten des ersten Übergangs nicht vorgesehen.

**Ergebnis** Im Wesentlichen unterscheiden die in Abbildung 4.26 dargestellten Lebenszyklen maximal zwei Paare von Zuständen:

Zum einen wird zwischen dem unkonfigurierten und konfigurierten Zustand eines installierten Plugins unterschieden. Dies entspricht in der Darstellung der Lebenszyklusnetze der Trennung zwischen rechter und linker Seite. „Konfiguriert“ bedeutet je nach System etwas geringfügig anderes. In der Regel ist darin die Existenz eines funktionsfähigen Kontextes und/oder die erfolgreiche Auflösung statischer Abhängigkeiten des verwalteten Plugins enthalten. Im Falle einer konfigurierte OSGi-Service-Component sind ebenso wie beim NetBeans-Modul zusätzlich die funktionalen Abhängigkeiten erfolgreich aufgelöst. Diese Unterscheidung ist im Wesentlichen für die Verwaltung des Systems von Interesse, um die korrekte Funktion der Gesamtkomposition gewährleisten zu können.

Zum anderen wird zwischen dem ungenutzten und genutzten Zustand eines konfigurierten Plugins unterschieden. Dies entspricht der vertikal angeordneten Raute auf der linken Seite vieler Lebenszyklusnetze. „Genutzt“ bedeutet ebenfalls je nach System etwas geringfügig anderes. Viele Systeme nennen diesen Zustand „Aktiv“, um zu

verdeutlichen, dass die Funktionalität des Plugins aktiv genutzt wird. Im Falle eines „initialisierten“ Netscape-Plugins liegt jedoch noch nicht unbedingt eine Nutzung vor. Hier zeigt der Zustand lediglich an, dass der Code geladen und das Plugin auf seine Nutzung vorbereitet ist.

Die Lebenszyklusnetze der Mozilla-Subsysteme XPCOM, Chrome und XPInstall sowie der einfachen OSGi-Services sind derart einfach gehalten, dass beide Unterscheidungen nicht auftreten. Nach dem in den vorigen Unterabschnitten besprochenen Wegfall von Netzelementen, welche lediglich der Modellierung einer nicht-dynamischen Konfiguration oder von Zuständen des Modullademechanismus dienen, reduzieren sich diese Lebenszyklen auf das einfache Modell mit genau einer Transition je Konfigurationsschritt, welches sich bereits in Unterabschnitt 4.3.8.2 aus der Grundstruktur der Plugin-Verwaltung ergeben hat.

Für die anderen Lebenszyklusnetze, welche mindestens eine der Zustandsunterscheidungen enthalten, gilt ähnliches. Die Unterscheidung von Zuständen nach Konfiguriertheit entfällt mit derselben Begründung wie zuvor: Die verwalteten Systemkomponenten bekommen vom Übergang in den oder aus dem unkonfigurierten Zustand nichts mit. Somit bleiben zur Modellierung dieser Lebenszyklen lediglich die vertikal angeordneten Rauten zur Unterscheidung der Initialisiertheit bzw. Aktiviertheit. Diese Übergänge werden den verwalteten Systemkomponenten mitgeteilt und erfordern in der Regel sogar ihre aktive Beteiligung oder Zustimmung. Hier sind wiederum zwei Varianten zu beobachten – entweder werden die Übergänge ein- oder zweiphasig gestaltet. Genau diese beide Varianten von Lebenszyklen sind bereits in Unterabschnitt 4.3.8.2 modelliert worden. Damit ist klar, dass die im Rahmen der konzeptionellen Modellierung vorgeschlagenen beiden Varianten eines Lebenszyklus tatsächlich in der Lage sind, die Grundstrukturen aller in den untersuchten Plugin-Systemen vorgefundenen Lebensyzklusvarianten abzudecken.

### 4.4.3 Abgleich mit den untersuchten Systemen

In den vorangegangenen Unterabschnitten 4.4.1 und 4.4.2 wurde bereits diskutiert, inwieweit die in den untersuchten Plugin-Systemen vorgefundenen Beziehungsarten und Lebenszyklusdefinitionen durch das in Abschnitt 4.3 entwickelte konzeptionelle Modell abgedeckt sind oder sich darin integrieren lassen. Im Ergebnis dieser Diskussionen ist klar geworden, dass diese beiden wesentlichen Eigenschaften der untersuchten Plugin-Systeme im Modell abbildbar sind. Im Folgenden soll ergänzend auf Besonderheiten der einzelnen Systeme eingegangen werden, welche über die zuvor diskutierten zentralen Eigenschaften hinausgehen. Als Ausgangspunkt der Diskussion dient die Zusammenfassung der Untersuchungsergebnisse in Unterabschnitt 3.4.1.

Die Netscape-Plugin-API (**NPAPI**, siehe Unterabschnitt 3.1.1) repräsentiert im Rahmen dieser Arbeit wie bereits mehrfach erwähnt eine ganze Klasse von Plugin-Systemen. Diesen „klassischen“ Plugin-Systeme ist gemein, dass es sich in erster Linie um Anwendungen handelt, welche aus praktischen Erwägungen heraus mit einer fachspezifischen Erweiterungsschnittstelle versehen werden. Daher gibt es in diesen Systemen keine allein auf die Verwaltung von Plugins spezialisierte Systemkomponente, ebensowenig ist die rekursive Erweiterbarkeit einzelner Plugins vorgesehen. In der konzeptionellen Modellierung sind solche Plugin-Systeme bereits nach dem ersten Mo-

dellierungsschritt in Unterabschnitt 4.3.2 fast vollständig erfasst, lediglich die explizite Modellierung eines Lebenszyklus wie in Unterabschnitt 4.3.8.2 wird noch benötigt. Falls komplexere Anwendungen mit mehreren Erweiterungsspezifikationen versehen werden, lassen sich diese nach dem in Unterabschnitt 4.3.3 modellierten Konzept der unabhängigen Erweiterbarkeit integrieren. Die enge Verbindung der NPAPI-Verwaltung mit betriebssystemspezifischen Modullademechanismen ist, wie bereits in vorangegangenen Diskussionen festgestellt, für die konzeptionelle Modellierung im Rahmen dieser Arbeit nicht relevant.

Die den **Mozilla-Extensions** (siehe Unterabschnitt 3.1.2) neben der NPAPI zur Verfügung stehenden Subsysteme XPCOM, Chrome und XPInstall sind, wie bereits in Unterabschnitt 3.4.1 festgestellt, für sich genommen nicht als vollwertige Plugin-Systeme zu betrachten. Es handelt sich dabei jeweils um offene Systeme, welche durch den gemeinsamen Mantel des XPInstall-Toolkits zu einem erweiterbaren System zusammengefasst werden. Eine *intendierte* Erweiterbarkeit im strengen Sinne (siehe Definition 5) ist nicht auf der fachlichen Ebene der einzelnen XPCOM-Komponenten oder Chrome-Pakete, sondern nur auf der zentralen Ebene des Gesamtsystems festzustellen, wobei das XPInstall-Subsystem als gemeinsame Konfigurationsschnittstelle dient. Dennoch wurden sie in den vorangegangenen Diskussionen dieses Abschnitts individuell betrachtet, weil sie als Erweiterungsspezifikationen des Gesamtsystems verstanden werden können, die in ihrer jeweiligen Schnittstelle sehr verschieden sind. Insofern sind die Eigenheiten, welche mitunter nur schwierig in das modellierte Plugin-Konzept einzufügen sind, eigentlich als konkrete Varianten der im Modell nur abstrakt repräsentierten lokalen Funktionalität zu sehen. Dies betrifft sowohl die generische Objektverwaltung, welche von XPCOM-Komponenten zu implementieren ist, als auch die zentralisierte Montage passiver Datenstrukturen durch das Chrome-Subsystem. Die Funktionalität des XPInstall-Subsystems ist hingegen einfach im modellierten Plugin-Konzept wiederzufinden, da es sich hierbei im Wesentlichen um eine zentralisierte Plugin-Verwaltung handelt, wie sie in Unterabschnitt 4.3.6 eingeführt wurde.

Die Architektur der **Eclipse**-Plattform (siehe Abschnitt 3.2) wird nahezu vollständig vom modellierten Plugin-Konzept erfasst, da in der Eclipse-Architektur die Bedeutung der Erweiterungsbeziehung ebenso stark betont wird wie im referenznetzbasierten Modell des Plugin-Konzepts. Wie bereits in den Unterabschnitten Unterabschnitt 4.2.6.1 und 4.2.6.2 erläutert, findet allerdings mit der *Lazy Loading Rule* [GB03, S. 16] die zweite stark betonte Idee der Eclipse-Architektur keinen Eingang in die abstrakte Modellierung des Konzepts, da sie auf den Zuständen des hier ausgeblendeten Modullademechanismus aufbaut.

Unter den in den späteren Modellierungsschritten besprochenen Optionen nutzt das klassische Plugin-System die zentralisierte, nicht-dynamische Konfiguration durch eine zentrale Plugin-Verwaltung. Die modernere Variante der Eclipse-Plattform auf Basis des OSGi-Rahmenwerkes ergänzt neben der von den zentralen Eclipse-Plugins nicht genutzten Dienstverwaltung auch dynamische Konfigurierbarkeit in der klassischen Plugin-Verwaltung. Auch damit bewegt sich die Eclipse-Architektur weiterhin im Rahmen des hier modellierten Plugin-Konzepts.

Die ausgefeilte Dienstverwaltung des OSGi-Rahmenwerks passt in ihrer einfachen Variante auf die grob skizzierten Modellierungsvorschläge aus Unterabschnitt 4.3.8.1. Die komplexere Verwaltung der OSGi-Service-Components, welche sogar Einfluss auf

den Lebenszyklus der dienstanbietenden Plugins nehmen kann, erfordert eine über das hier Besprochene hinausgehende Modellierung. Wie bereits in Unterabschnitt 4.4.2.2 erläutert, reduziert sich auch der Lebenszyklus der Service-Components durch die im Rahmen der Modellierung eingeführten Abstraktionen auf den im Modell skizzierten Bereich.

Die Flexibilität der Module der **NetBeans**-Plattform (siehe Unterabschnitt 3.3.2) erschwert eine Einordnung im Rahmen des Plugin-Konzepts. NetBeans-Module können sich zwar wie Plugins entlang der intendierten Erweiterbarkeit aneinander koppeln, da die in NetBeans angebotene Konfigurations- und Kommunikationsinfrastruktur dieser Beziehung nicht im Wege steht. Aber die Module können auch jede andere Art von Beziehung eingehen. In Unterabschnitt 4.4.1 wurde angedeutet, wie solche Beziehungsarten in das modellierte Plugin-Konzept integriert werden können. Der Nachteil der Flexibilität der NetBeans-Architektur ist, dass den Entwicklern die Erschaffung einer wohlorganisierten, übersichtlichen Architektur erschwert wird. Interessant ist das in Abschnitt 4.4.2.2 besprochene Detail, dass einzig im NetBeans-System die Mitbestimmung der Module an ihrer eigenen Konfiguration explizit vorgesehen ist. Diese Möglichkeit ist im modellierten Plugin-Konzept ebenfalls enthalten.

Der Schwerpunkt des **MagicBeans**-Rahmenwerks (siehe Unterabschnitt 3.3.4) liegt in der zentralen, atomaren Rekonfiguration des dynamischen Systems, welche sämtliche Verbindungen steuert. Im modellierten Plugin-Konzept besteht die Möglichkeit der zentralen Konfiguration ebenfalls, allerdings ist das Referenznetzmodell grundsätzlich darauf ausgelegt, nur solche Plugins atomar zu konfigurieren, die von der Rekonfiguration auch tatsächlich betroffen sind. Somit erlaubt das Referenznetzmodell nebenläufige Konfiguration, die MagicBeans-Verwaltung hingegen nicht. Wie in Abschnitt 4.4.1.2 besprochen, lässt sich die Eigenschaft der MagicBeans-Verwaltung, die Lebensdauer von Beziehungen strikt zu kontrollieren auch im Referenznetzmodell nachbilden.

Das MagicBeans-System hat als einziges der untersuchten Systeme mit dem hier modellierten Plugin-Konzept eine formale Semantik gemein. Die Formalisierung des MagicBeans-System konzentriert sich auf die Zulässigkeit von Systemkonfigurationen und lässt dazwischen nur atomare Übergänge zu. Die operationale Semantik des Petrinetzmodells ermöglicht dagegen eine möglichst große Nebenläufigkeit bei der Rekonfiguration des Systems. Eine Typprüfung auf passende Schnittstellen von Erweiterungen oder Dienstanbietern ist zwar zunächst nicht im Petrinetzmodell enthalten, welches auf dynamischer Bindung basiert. Wie während der Modellierung erläutert, lässt sich aber eine typisierte Variante des Modells erstellen, indem entweder der Referenznetzformalismus entsprechend ergänzt oder die Typbezeichner der Systemkomponenten explizit über Kanäle und Stellen kommuniziert werden.

Die Gegenüberstellung konkreter Plugin-Architekturen wurde in den Unterabschnitten 3.3.6–3.3.9 durch einen Blick auf verwandte Themengebiete abgerundet, in denen erweiterbare Softwarearchitekturen entwickelt, unterstützt oder erforscht werden.

Die in Unterabschnitt 3.3.6 besprochene **Integration** unabhängig voneinander entstandener Plugin-Systeme lässt sich im modellierten Plugin-Konzept sehr leicht abbilden. Da ein Plugin-Management-System (PMS), wie es in den späteren Modellierungsschritten enthalten ist, sich nicht grundlegend von anderen erweiterbaren Systemkomponenten unterscheidet, kann es wie jede andere Systemkomponente auch einem

bestehenden System hinzugefügt werden. Sobald die Konfiguration des einen PMS als Systemkomponente unter der Regie eines anderen PMS erfolgreich abgeschlossen ist, stehen beide Verwaltungsebenen weitgehend unabhängig voneinander zur Verfügung.

Damit die Konfiguration erfolgreich verläuft, müssen das hinzugefügte und das aufnehmende PMS aber eine gemeinsame Schnittstelle besitzen. Dies erfordert in der Regel Anpassungen an einer der beiden Verwaltungskomponenten, um die getrennt gewachsenen Schnittstellen aufeinander abzubilden. Der in Unterabschnitt 3.4.1 zusammengefasst genannte Problembereich der Abbildung zwischen den verschiedenen Abstraktionskonzepten der beteiligten Softwarearchitekturen besteht somit auch im modellierten Plugin-Konzept. Die nötigen Anpassungen brauchen entsprechend der Idee der weitgehend unabhängigen Schachtelung der beiden Systeme nur an den Punkten vorgenommen zu werden, wo eine Kommunikation zwischen Systemkomponenten beider Systeme benötigt wird. Da bei der Modellierung des Plugin-Konzepts von konkreter Funktionalität abstrahiert wurde, ist der zweite in der Diskussion genannte Problembereich der Überschneidung oder Inkompatibilität grundlegender Klassenbibliotheken im Modell zwar nicht relevant, wird in der Praxis aber dennoch auftreten.

Von den im Kontext der Sprache **Java** (siehe Unterabschnitt 3.3.7) diskutierten Mechanismen zur Implementierung erweiterbarer Systeme befassen sich die meisten mit dem flexible Laden und Instantiieren von Klassen und einer Stärkung des Kapselungsprinzips. Damit liegen sie außerhalb des Betrachtungsbereichs der Modellierung des Plugin-Konzepts. Lediglich der `ServiceLoader` implementiert einen Mechanismus zum Beziehungsaufbau. Dieser orientiert sich an der Dienstmetapher und setzt diese in Form nicht-dynamischer funktionaler Abhängigkeiten um. Die Betrachtung funktionaler Abhängigkeiten im Modell wurde bereits in Unterabschnitt 4.4.1 besprochen.

Das im Kontext der der **verteilten Entwicklung** angesprochene Problem der Namenskonflikte und Evolution von Schnittstellen wird im modellierten Plugin-Konzept nicht behandelt. Da synchrone Kanäle in Referenznetzen mit dynamischer statt statischer Bindung arbeiten, können zwar keine falsch typisierten Kanäle zustande kommen, aber das Problem der semantischen Änderungen einer Schnittstelle besteht weiterhin. Immerhin erlaubt die Verwendung von Petrinetzen die grafische Darstellung des prozessorientierten Teils der Schnittstellensemantik, so dass inkompatible Änderungen prinzipiell mittels Petrinetzanalyse erkannt werden können.

Die in Unterabschnitt 3.3.9 angesprochene Ähnlichkeit zwischen generischen Plugin-Systemen und **Betriebssystemen** wird durch das modellierte Plugin-Konzept nicht beeinflusst. Die Betrachtung des Betriebssystemkerns als eine Systemkomponente mit zentraler Konfigurationsschnittstelle und diverser von verwalteten Systemkomponenten wie z. B. Gerätetreibern nutzbarer zentraler Infrastruktur ist im Modell problemlos möglich. Insbesondere die in Unterabschnitt 4.4.1.1 vorgeschlagene Infrastruktur zur vermittelten Kommunikation entspricht einer abstrakten Modellierung der durch ein Betriebssystem angebotenen Kommunikationskanäle. Die konkreten Binärschnittstellen liegen – wie zuvor schon angemerkt – außerhalb des bei der Modellierung betrachteten Bereichs. Paketverwaltungswerkzeuge wie RPM oder DPKG/APT lassen sich im Modell als Verwaltungskomponenten (wie ein PMS) integrieren, welche auf die Konfigurationsschnittstellen aller anderen Systemkomponenten zugreifen.

## 4.5 Diskussion

Um die konzeptionelle und praktische Qualität des im Abschnitt 4.3 entwickelten Referenznetzmodells des Plugin-Konzepts beurteilen zu können, sind mehrere Aspekte zu diskutieren. Ein sinnvoll nutzbares Modell muss sich vor allem an den in Abschnitt 4.1 ermittelten konzeptionellen Anforderungen messen lassen. Darüber hinaus sollte es in der Lage sein, die zentralen architektonischen Beziehungen und Eigenschaften der in Kapitel 3 untersuchten konkreten Plugin-Systeme abbilden zu können. Die von der Modellierung ausgehende Prägung der Entwicklung von Softwaresystemen stellt einen ergänzenden Diskussionspunkt dar, der den Vergleich mit den in Kapitel 2 betrachteten Techniken und Paradigmen zur Softwareentwicklung ermöglicht.

Dass das Modell sich zur Abbildung der in Kapitel 3 untersuchten konkreten Plugin-Systeme eignet, wurde bereits im vorigen Abschnitt diskutiert und bestätigt. Während der folgenden Diskussion bezüglich der in Abschnitt 4.1 ermittelten Anforderungen an das Modell bietet es sich allerdings an, die untersuchten Plugin-Systeme ebenfalls in das Schema der Anforderungen einzuordnen, soweit sie auf konkrete Systeme übertragbar sind.

### 4.5.1 Eigenschaften des konzeptionellen Modells

Die Frage, in wie weit das Modell den zuvor aufgestellten Anforderungen entspricht, wurde schon während der einzelnen Modellierungsschritte regelmäßig angesprochen. Die im Laufe des Modellierungsprozesses erreichten Eigenschaften werden im Folgenden noch einmal zu den erforderlichen Modellierungsschritten in Beziehung gesetzt.

Bereits ab dem ersten Modellierungsschritt (siehe Unterabschnitt 4.3.1) repräsentiert das Netz die grundlegende Eigenschaft eines **offenen, dynamisch rekonfigurierbaren Systems** wie imUnterabschnitt 4.1.1 definiert. Die äußere Konfigurationsschnittstelle des Systems lässt sich ebenfalls zu Beginn der Modellierung so **einfach** wie überhaupt nur denkbar nutzen. Allerdings erfordert die Konfiguration (Integration oder Entfernung) eines Plugins ab der Einführung von unabhängiger Erweiterbarkeit (siehe Unterabschnitt 4.3.3), nicht-hierarchischen Strukturen (siehe Unterabschnitt 4.3.5) und von Dienstbeziehungen (siehe Unterabschnitt 4.3.8.1) unter Umständen mehrere Schritte. Die in Unterabschnitt 4.1.2 geforderte Einfachheit der Konfiguration ist nach außen weiterhin gewährleistet, indem eine zentrale Verwaltung die Koordination der Konfigurationsschritte entweder vollständig oder in Zusammenarbeit mit den Plugins übernimmt.

Das modellierte Plugin-Konzept erhebt die in Unterabschnitt 4.1.3 geforderte **intendierte Erweiterbarkeit** zum zentralen Merkmal der Architektur (siehe Unterabschnitt 4.3.2), kann aber als hybrides System auch die Verwaltung von Dienstbeziehungen integrieren (siehe Unterabschnitt 4.3.8.1). Die Modellierbarkeit weiterer Beziehungsarten im Rahmen des Modells wurde in Unterabschnitt 4.4.1 diskutiert und bestätigt.

Es besteht aufgrund des verwendeten objektbasierten Formalismus der Referenznetze von Anfang an eine starke **Kapselung** der modellierten Systemkomponenten, welche ausschließlich über synchrone Kanäle miteinander kommunizieren können (vgl. Unterabschnitt 4.2.5). Somit ist auch die in Unterabschnitt 4.1.4 aufgestellte Anforde-

rung erfüllt. Zu betonen ist hierbei eine über normale objektorientierte Modellierung hinausgehende Autonomie der Schnittstellen-anbietenden Systemkomponenten, da sie technisch die Aktivierung ihrer Kanäle verhindern können. Im Modell wird die Ausnutzung dieser Autonomie stellenweise bewusst verboten, um eine zuverlässig reagierende Verwaltung des Systems zu ermöglichen. Das Schaltverhalten der Referenznetze stellt sicher, dass Plugins, welche diese Verbote missachten, die Funktionsfähigkeit des restlichen Systems nicht beeinträchtigen.

Die in Unterabschnitt 4.1.5 diskutierte Möglichkeit der expliziten Definition von **Lebenszyklen** für Plugins lässt sich im Modell jederzeit integrieren. Das bereits als Teil der Modellierungstechnik in Unterabschnitt 4.2.7 beschriebene Verfahren zur Verzahnung von Lebenszyklus- mit Strukturmodellen wurde in einem der letzten Modellierungsschritte in Unterabschnitt 4.3.8.2 exemplarisch auf das Modell des Plugin-Konzepts angewandt. In Unterabschnitt 4.4.2 wurden zudem die in den untersuchten Plugin-Systemen vorgefundenen Lebenszyklen gegeneinander und gegen die beiden exemplarischen Lebenszyklen im konzeptionellen Modell abgeglichen, so dass die konzeptionelle Modellierung alle wesentlichen Zustände und Übergänge der vorgefundenen Lebenszyklen abdeckt.

Die in Unterabschnitt 4.1.6 ermittelte Anforderung der Unterscheidung **mehrerer Ebenen** bei der Verwaltung von Plugins und der Definition von Lebenszyklen findet sich im Referenznetzmodell in natürlicher Weise wieder. Ab der Einführung rekursiver Erweiterbarkeit im Unterabschnitt 4.3.4 verwaltet jede erweiterbare Systemkomponente ihre Erweiterungen selber. Mit der Aufhebung der Beschränkung auf eine strikt hierarchische Organisation des Systems in Unterabschnitt 4.3.5 können Plugins zugleich zu mehreren Verwaltungen und Lebenszyklen konform implementiert werden. In den nachfolgenden Modellierungsschritten wurden verschiedene Varianten diskutiert, die einzelnen Verwaltungen unter einer zentralen Verwaltung zu koordinieren, ohne die Autonomie der einzelnen Verwaltungen zu stark zu beschneiden.

Aufgrund der Verwendung von Petrinetzen als Modellierungssprache ist eine intuitive Darstellung von **Nebenläufigkeit** von Grund auf vorhanden, die auf natürliche Weise zu einer hohen Nebenläufigkeit im System führt. Bei Petrinetzen erfordert die Einschränkung der Nebenläufigkeit zusätzlichen Aufwand – anders als in sequentiellen Programmiersprachen, in denen die Einführung von Nebenläufigkeit mit Aufwand verbunden ist. Damit stellt die Wahl der Modellierungssprache schon eine gute Voraussetzung zur Erfüllung der in Unterabschnitt 4.1.7 geforderten Berücksichtigung von Nebenläufigkeit in modernen Softwaresystemen dar. Während der Modellierung wurde zudem stets darauf geachtet, die im System mögliche Nebenläufigkeit nicht mehr als unbedingt notwendig durch eine zentrale Verwaltung oder Konfigurationsmechanismen einzuschränken (vgl. Unterabschnitt 4.2.3). Gleichermaßen ergibt sich aus der gewählten Modellierungssprache heraus automatisch die in Unterabschnitt 4.1.8 geforderte **präzise Semantik** der grafischen Modelle.

### 4.5.2 Eigenschaften der untersuchten Plugin-Systeme

Betrachtet man im Vergleich zum konzeptionellen Modell die existierenden Plugin-Systeme, so ist zunächst festzustellen, dass es sich bei sämtlichen in Kapitel 3 untersuchten Systemen um **offene Systeme** handelt. Nicht alle Systeme unterstützen

allerdings eine **dynamische Rekonfiguration**, wie bereits im Rahmen der Diskussion in Unterabschnitt 3.4.1 festgestellt. Es existieren auch Mischformen aus dynamischer und nicht-dynamischer Rekonfiguration. Beispielsweise besteht in einigen nicht-dynamischen Plugin-Systemen die Möglichkeit, während der Laufzeit des Systems die Gesamtkonfiguration neu zu bestimmen, so dass das System die Konfigurationsübergänge des Systemstarts erneut durchführt. Oder es wird eine vom Prinzip her dynamische Plugin-Verwaltung mit Plugins konfrontiert, welche nicht auf dynamische Konfigurationsänderungen reagieren können.

In allen Plugin-Systemen ist die dem Anwender zugemutete Steuerung der Konfiguration auf ein **einfaches** „Hinzufügen, Nutzen, Entfernen“ reduziert, wenn auch die benötigten technischen Schritte in ihrer Anwenderfreundlichkeit von simplen Operationen auf Dateisystemebene bis zu grafischen Dialogen mit integrierter Suchfunktion variieren. Lediglich in der Mozilla-Architektur bieten die Subsysteme nicht von sich aus eine derart einfache Konfiguration an, sondern setzen auf das gemeinsame Konfigurationswerkzeug XPInstall bzw. das nachfolgende Toolkit. Die dem Systementwickler gebotenen Konfigurationsschnittstellen der untersuchten Architekturen unterscheiden sich deutlicher. Während ein Plugin-Entwickler in der Netscape- und Mozilla-Anwendungsfamilie ebenso wie bei Nutzung der einfachen OSGi-Services die Integration seiner Funktionalität ins Gesamtsystem explizit programmieren muss und dabei auf die Einhaltung der Lebenszykluszustände zu achten hat, bieten Eclipse, NetBeans und MagicBeans einen deklarativen Mechanismus zum Beziehungsaufbau, der allerdings nur atomare Konfigurationsänderungen unter der Kontrolle der zentralen Verwaltung zulässt. Das in dieser Arbeit modellierte Plugin-Konzept kann in beide Richtungen ausgestaltet werden.

Grundsätzlich handelt es sich zwar bei jedem Plugin-System um ein **intendiert erweiterbares** System, aber nicht jedes der untersuchten Systeme beinhaltet ein Rahmenwerk, welches die Erweiterungsbeziehung explizit unterstützt. Streng genommen sind das Mozilla-Subsystem XPCOM ebenso wie das MagicBeans-System gar nicht auf die Erweiterungsbeziehung ausgelegt, da sie Klassen und Objekte verwalten, welche – wie in Unterabschnitt 4.4.1.2 diskutiert – nicht per se intendiert erweiterbar sind, sondern entlang beliebiger Beziehungsarten gekoppelt werden können. Das XPInstall-Rahmenwerk stellt weder eigene Konfigurationsfunktionalität zur Verfügung noch verwaltet es einen eigenen Zustand – da es lediglich an andere Mozilla-Subsysteme delegiert, ist es isoliert nicht als intendiert erweiterbares System zu betrachten. Das OSGi-Rahmenwerk konzentriert sich anstelle der Erweiterungs- auf die Dienstbeziehung. Die NetBeans-Plattform stellt hingegen Primitive zum Beziehungsaufbau zur Verfügung, welche sich für beliebige Beziehungsarten verwenden lassen. Lediglich Eclipse als „reines“ Plugin-System sowie die NPAPI als Stellvertreter für anwendungsspezifische Plugin-Systeme sehen explizit eine oder mehrere Erweiterungsspezifikationen vor.

Eigentlich ist anzunehmen, dass sämtliche Systeme, welche unter dem objektorientierten Softwareentwicklungsparadigma entwickelt werden, eine starke **Kapselung** von Natur aus unterstützen. In der Praxis besteht jedoch jedes Plugin aus mehreren Klassen und Objekten, welche individuelle Beziehungen mit Objekten anderer Plugins eingehen können. Eine explizite Kapselung auf Ebene der verwalteten Systemkomponenten bieten nur Eclipse, OSGi und NetBeans, da sie jedem Plugin einen festen Repräsentanten und auch einen eigenen Java-Classloader zuweisen. Da das Mozilla-

Subsystem XPCOM selber erst die objektorientierte Kapselung im Softwaresystem implementiert, passt hier natürlich die Kapselungsebene genau auf die verwalteten Klassen. Die Mozilla-Subsysteme Chrome und XPInstall betrachten jedoch die gesamte Anwendung als ein Konglomerat gleichberechtigter Einzelteile, welche in der Regel nur Elemente von Paketen darstellen. MagicBeans kontrolliert zwar mittels Proxy-Objekten die Beziehungen zwischen Objekten verschiedener Plugins, verzichtet jedoch auf eindeutige Repräsentanten und Classloader auf Ebene der Plugins, so dass die eindeutige Zuordnung von Klassen zu Plugins nicht sichergestellt ist. Das in dieser Arbeit präsentierte Modell nutzt die in der konzeptionellen Modellierung bestehende Möglichkeit, die Systemgrenzen von Klassen und Plugins aufeinander fallen zu lassen, so dass sie jeweils der Grenze eines Netzes entsprechen. Im Zuge einer konkreteren Implementierung mit mehreren Klassen je Plugin sind hier Ergänzungen vorzunehmen.

Wie bereits während der Untersuchung der existierenden Plugin-Systeme mehrfach angemerkt, werden **Lebenszyklen** nur selten explizit modelliert und deklariert. Die in der Untersuchung erarbeiteten Lebenszyklusnetze basieren in vielen Fällen auf der Analyse der Schnittstellendokumentation von Klassen, welche als Repräsentanten für Plugins verwendet werden. Einzige Ausnahme stellt hier das OSGi-Rahmenwerk dar, welches auf endliche Automaten zur Darstellung der möglichen Zustände und Übergänge zurückgreift. Das in weiten Teilen durch eine formale Spezifikation erfasste MagicBeans-System definiert hingegen keinen Lebenszyklus, da es nur den Zustand, in welchem ein Plugin ins System integriert ist, betrachtet. Hinzufügen und Entfernen finden als atomare Operationen statt.

Eine globale Pluginverwaltung findet sich in allen untersuchten Systemen. Aber nur Eclipse, OSGi und NetBeans bieten im Rahmenwerk bereits Möglichkeiten zur Verwaltung der lokalen Beziehungen zwischen Systemkomponenten auf **mehreren Ebenen**. Von diesen ist wiederum lediglich OSGi mit einer detaillierten Beschreibung der Kopplung globaler und lokaler Lebenszyklen aneinander ausgestattet. Im Falle der Netscape-Plugins fallen globale und lokale Verwaltung zusammen, da es wie in den meisten anwendungsspezifischen Plugin-Systemen nur eine einzige Erweiterungsschnittstelle gibt. In gewissem Sinne ließen sich zwar auch die Mozilla-Subsysteme NPAPI, XPCOM und Chrome als lokale Verwaltungen unter der globalen Verwaltung des XPInstall-Subsystems betrachten, aber die Verwendung von XPInstall ist nicht obligatorisch.

Eine Empfehlung zur Berücksichtigung von **Nebenläufigkeit** in Plugins findet sich interessanterweise nur in der Dokumentation zu Netscape-Plugins. In der Regel werden die verwalteten Systemkomponenten als reaktive Systeme verstanden, welche auf Anforderung ihre Funktionalität erbringen. Eine Verwaltung, welche mit nebenläufigen Konfigurationsanforderungen umgehen kann, bringt – laut Dokumentation – nur das OSGi-Rahmenwerk mit. Stattdessen dokumentieren viele Systeme, darunter die NPAPI, die Mozilla-Subsysteme und auch die MagicBeans-Verwaltung, dass eine nebenläufige Nutzung der Konfigurationsverwaltung nicht vorgesehen ist.

Da die untersuchten Plugin-Systeme reale Softwaresysteme und keine visualisierenden Modelle sind, erübrigt sich die Frage nach einer **präzisen Semantik** der **visuellen Modellierungssprache**. Lediglich die in Darwin modellierte Architektur der MagicBeans-Systemkonfiguration und die OSGi-Lebenszyklusmodelle gehen in diese

| System | offenes System | dynamisch rekonf. System | einfache Konfiguration | deklarative Konfiguration | Intendierte Erweiterbarkeit | Dienstnutzung | Kapselung auf Modulebene | formal definierter Lebenszyklus | mehrere Verwaltungsebenen | Verwaltungsebenenkopplung | unterstützt Nebenläufigkeit | Verwaltung threadsicher | visuelle Modellierung | präzise Semantik |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Netscape-Plugin-API | x | (x) | x | – | x | – | – | – | – | – | x | – | – | – |
| Mozilla Extensions | | | | | | | | | | | | | | |
| - XPCOM | x | (x) | – | – | – | – | x | – | – | – | – | – | – | – |
| - Chrome | x | – | – | – | x | – | – | – | – | – | – | – | – | – |
| - XPInstall | x | (x) | x | – | – | – | – | – | – | – | – | – | – | – |
| Eclipse-Plugins | | | | | | | | | | | | | | |
| - klassisch | x | – | x | x | x | – | x | – | x | – | ? | ? | – | – |
| - OSGi | x | x | x | (x) | – | x | x | x | x | x | ? | x | – | (x) |
| NetBeans | x | x | x | x | (x) | (x) | x | – | x | – | ? | ? | – | – |
| MagicBeans | x | x | x | x | – | – | – | – | – | – | – | – | (x) | x |
| Petrinetzmodell | x | x | x | (x) | x | x | (x) | x | x | x | x | x | x | x |

(x): Die Eigenschaft ist nur eingeschränkt vorhanden (siehe Text).
?: Die Eigenschaft war der Systemdokumentation nicht zu entnehmen.

Tabelle 4.1: Erfüllung der Anforderungen durch untersuchte Systeme und das konzeptionelle Modell

Richtung. Beide stellen allerdings keine visuelle Modellierung der eigentlichen Pluginverwaltung dar.

Die Ergebnisse der vorangegangenen Diskussion sind in Tabelle 4.1 noch einmal im Überblick zusammengefasst. Die letzte Zeile der Tabelle stellt die bereits im 4.5.1 diskutierten Eigenschaften des in dieser Arbeit entwickelten konzeptionellen Modells dagegen.

### 4.5.3 Einordnung als Softwareentwicklungsansatz

Das im Abschnitt 4.3 in mehreren Schritten entwickelte konzeptionelle Modell stellt kein unverändert zu verwendendes Modell für beliebige Anwendungsfälle dar. Stattdessen handelt es sich mehr um eine Sammlung von Modellierungsmöglichkeiten, welche in verschiedenen Kombinationen dazu geeignet sind, Plugin-Systeme zu modellieren.

Die in den Modellierungsschritten entwickelten Zwischenformen für erweiterbare Systemkomponenten oder das Plugin-Management-System (PMS) zeigen zum Teil unterschiedliche Richtungen der Verfeinerung auf. Bei der Entwicklung eines Plugin-Systems aus den modellierten Netzen heraus stehen daher verschiedene Alternativen zur Realisierung eines rekonfigurierbaren Systems mit unterschiedlichen Flexibilitäts- und Kontrollgraden zur Verfügung.

In diesem Sinne lässt sich das modellierte Plugin-Konzept mit den in Abschnitt 2.4 diskutierten Entwurfsmustern vergleichen, welche als auf die jeweilige Situation anzupassende, wiederverwendbare Problemlösungen auf der Ebene des Software-Entwurfs charakterisiert werden. Allerdings ist das modellierte Konzept sowohl zu konkret als auch zu komplex, um als ein einziges Entwurfsmuster behandelt werden zu können. In den über mehrere Modellierungsschritten erarbeiteten Netzen lassen sich aber wiederkehrende Elemente erkennen, so dass das gesamte Modell eher als eine Referenzarchitektur zu betrachten ist, welche eine Reihe von Entwurfsmustern für immer wiederkehrende Situationen vorgibt. Die enge Verzahnung der Grundbausteine des Modells erschwert allerdings eine präzise Abgrenzung unterschiedlicher Entwurfsmuster. Die zu diesem Zweck noch zu betreibende Analyse des Modells kann in weiterführenden Arbeiten erfolgen.

In Unterabschnitt 4.1.9.1 wurde bereits auf Ähnlichkeiten und Unterschiede des Plugin-Begriffs im Vergleich zu den verschiedenen architekturprägenden Konzepten der in Kapitel 2 betrachteten Softwareentwicklungstechniken und -paradigmen eingegangen. Die gegenseitige Abgrenzung der Begriffe Modul, Objekt, Komponente, Plugin und Agent lässt sich folgendermaßen zusammenfassen: Zwischen Modul, Komponente und Plugin kann bei rein technischer Betrachtung eine Spezialisierungshierarchie aufgestellt werden, die allerdings unter Berücksichtigung geschäftlicher, fachlicher oder integrativer Aspekte keinen Bestand hat. Das Objekt stellt eine eigene Variante der Spezialisierung des Modulbegriffs dar, welche Code und Zustand zusammenführt, während das auf Flexibilität ausgelegte Konzept des Agenten weitgehend orthogonal zu den anderen vier Konzepten definiert ist.

Das modellierte Plugin-Konzept fügt sich genau in dieses Diskussionsergebnis ein – auf technischer Ebene sind die modellierten Netze zugleich als Plugins, Komponenten und Module betrachtbar. Weder der geschäftliche noch der fachliche Aspekt der Konzepte werden explizit durch das Modell bestimmt. Allerdings legt die Modellierung deutlich die Integration von Systemkomponenten in ein Gesamtsystem entlang des Konzepts der intendierten Erweiterbarkeit nahe, welche nur bedingt zum komponentenbasierten Paradigma passt. Die Objekt-typischen Eigenschaften wie Instantiierbarkeit und Vererbung sind für Plugins ebensowenig interessant wie für Komponenten oder Module.

Um das modellierte Plugin-Konzept mit dem Konzept des Agenten zu vergleichen, sind die verschiedenen Abstraktionsebenen aus dem Paradigma der AOSE zu berücksichtigen. Der Agentenbegriff der Mikro-Ebene betont orthogonal zu den anderen Konzepten stehende Eigenschaften, welche im Wesentlichen die Autonomie und Adaptivität der Systemkomponenten (Agenten) betreffen. Die im Zuge der vorangegangenen Diskussion unter den Stichpunkten *Kapselung* und *Nebenläufigkeit* angesprochene Autonomie von Systemkomponenten bringt das Modell des Plugin-Konzepts näher an diese Ideen der agentenorientierten Softwareentwicklung als das komponentenba-

sierte Paradigma oder die Architekturbeschreibungssprachen. Sowohl die in Kapitel 2 vorgestellten Softwareentwicklungsansätze (mit Ausnahme der AOSE) als auch die in Kapitel 3 untersuchten Plugin-Systeme betrachten in der Regel *reaktive* Systemkomponenten, welche ihre Funktionalität nur auf Anforderung zum Gesamtsystem beisteuern. Das hier entwickelte Modell erlaubt hingegen Plugins, welche als weitgehend autonome Subsysteme arbeiten und ihre Ergebnisse gelegentlich mit dem gastgebenden System austauschen. Dass solche Plugins in der Realität benötigt werden, beweisen unter anderem NPAPI-Plugins wie z. B. der Adobe Reader, welcher sich als eigentlich eigenständige Anwendung in die Darstellung des Webbrowsers einfügt, oder im Hintergrund arbeitende Analyse-Plugins in Entwicklungsumgebungen wie Eclipse und NetBeans.

Auf der Makro-Ebene bieten Plugins einen Teil der Flexibilität, die durch die rollenbasierte Organisation eines Multiagentensystems beschreibbar ist. Zwar sind die Rollen eines Plugins durch die Erweiterungsspezifikationen, zu denen es passt, festgelegt, so dass ein Plugin nicht wie ein Agent Rollen nach Belieben einnehmen und abgeben kann. Das Plugin-System als Ganzes stellt aber durchaus eine adaptive Softwarearchitektur dar, da sich hinzugefügte Plugins ohne weitere von außen explizit durchzuführenden Konfigurationsschritte quasi „von alleine“ in das System integrieren. Mit diesem Ansatz zur Selbstorganisation geht das Plugin-Konzept auch über die in der Objektorientierung geforderten Eigenschaften von Objekten hinaus.

Eine interessante Analogie eröffnet sich, wenn das in der AOSE nur am Rande (als notwendige Ausführungsumgebung) behandelte Konzept der Agentenplattform mit dem Konzept des erweiterbaren Systems verglichen wird. Zwar ist die Schnittstelle zwischen Plattform und Agent auf simplen Nachrichtenempfang und -versand beschränkt. Dies lässt sich jedoch als eine maximal verallgemeinerte Zusammenfassung aller möglichen Erweiterungsspezifikationen auffassen. Der von einer Agentenplattform vorgeschriebene Lebenszyklus ähnelt dem von erweiterbaren Systemen oder Plugin-Verwaltungssystemen vorgegebenen Lebenszyklus, da er im Wesentlichen die Integration (Erzeugung) und Entfernung (Termination) der Systemkomponenten umfasst. Überträgt man die Konzepte der intendierten Erweiterbarkeit auf Agent und Plattform, so lässt sich tatsächlich eine Erweiterung der auf einer Agentenplattform verfügbaren Funktionalität durch jeden auf der Plattform befindlichen Agenten erkennen. Diese Erweiterbarkeit ist allerdings nur auf einer sehr allgemeinen Ebene als „vorgesehene Erweiterbarkeit“ zu verstehen, da außer der (technischen) Konvention, Funktionalität über Dienstbeziehungen anzubieten, keine fachlichen Vorgaben existieren. Die Analogie zwischen Plattform und erweiterbarem System lässt sich auch visuell gut erkennen, wenn die petrinetzbasierten Modelle der Mulan-Plattform (siehe Abbildung 2.14) und des PMS (siehe Abbildung 4.18) nebeneinander gestellt werden. Beide enthalten eine zentrale Stelle, auf welcher Systemkomponenten mit zusätzlicher Funktionalität hinzugefügt und entfernt werden können. Die konkreten fachlichen Kanäle zur Nutzung der hinzugefügten Funktionalität beim Plugin-System werden allerdings in der Agentenplattform durch generische Kanäle zum Nachrichtentransport ersetzt. Die fachspezifische Kopplung von Systemkomponenten wird auf der Agentenplattform eine Ebene tiefer an die Protokolle delegiert. Eine gründliche Untersuchung der Belastbarkeit dieser Analogie könnte sich lohnen, ist aber im Rahmen dieser Arbeit nicht durchführbar.

Die Integration von Lebenszyklen in das konzeptionelle Modell stellt zudem einen weiteren Aspekt dar, der die Nähe zu agentenorientierten Systemen verdeutlicht. Damit wird eine Verhaltensspezifikation ähnlich einem einfachen Agenten-Interaktionsprotokoll als vollwertiges Modellelement ins System eingebracht. Die Betrachtung des modellierten Plugin-Konzepts als echt agentenorientiertes System ginge im derzeitigen Entwicklungsstand allerdings zu weit, da die Plugins nur wenig Flexibilität in ihrem Verhalten an den Tag legen können.

### 4.5.4 Einordnung als Architekturbeschreibungssprache

In Abschnitt 2.3 werden Prinzipien der Beschreibung von Softwarearchitekturen sowie die Eigenschaften von Architekturbeschreibungssprachen (ADLs) vorgestellt. Da in diesem Kapitel die Modellierung der Softwarearchitektur mit Referenznetzen anstelle einer bereits existierenden ADL durchgeführt worden ist, soll hier die in Abschnitt 4.2 vorgestellte Modellierungstechnik als Architekturbeschreibungstechnik eingeordnet werden. Eine ausführliche Betrachtung oder gar formal vollständige Analyse der Modellierungstechnik ist im den Rahmen dieser Arbeit nicht möglich. Anhand der intuitiven Verwendung in der Modellierung ist es aber möglich, Fähigkeiten und Eigenschaften abzuschätzen und mit den Anforderungen an eine ADL zu vergleichen.

Von den in Abschnitt 2.3.2 genannten Anforderungen an eine Architekturbeschreibungssprache erfüllt die Modellierung mit *Netze-in-Netzen* offensichtlich die *Dekomposition* in *Komponenten*. Das Modell nutzt zwar mit dem synchronen Kanal einen einzigen *Verbinder*-Typ, eine Ausgestaltung der Umgebung von Kanalendpunkt-Transitionen lässt aber verschiedene Kommunikationsformen und Interaktionsprotokolle zu. Dennoch kann diese Ausgestaltung nicht als gleichrangige Modellierung von Verbindern betrachtet werden, da sie innerhalb der Beschreibung von Komponenten stattfindet.

Anders als viele Architekturbeschreibungssprachen, deren visuelle Fähigkeiten sich auf eine überblicksartige Darstellung der Systemstruktur konzentrieren, erlaubt die hier verwendete Modellierungstechnik auf Basis des *Netze-in-Netzen*-Paradigmas eine integrierte Darstellung von Struktur und Verhalten. Wie in Unterabschnitt 4.2.7 erläutert, lässt sich eine ablauforientierte Lebenszyklusdefinition mit einem strukturellen Modell verzahnen, so dass die strukturell in einer Systemkomponente „enthaltenen“ Erweiterungen als Zustandsmarken durch den Lebenszyklus wandern. Im komponentenbasierten Paradigma und vielen darauf aufsetzenden ADLs wäre die Verhaltensspezifikation maximal als Annotation an Schnittstellenspezifikationen definierbar.

Mit der möglichen Schachtelung von Netzen steht im Paradigma der *Netze-in-Netzen* offensichtlich ein Mittel zur Modellierung von *Abstraktion*, *Wiederverwendbarkeit* und *Konfiguration* zur Verfügung. Durch die lokale Sicht jeden Netzes besteht zwar keine Möglichkeit, eine hierarchische Systemkonfiguration in einem einzigen Diagramm im globalen Überblick zu beschreiben. Die Plugin-Verwaltungsstellen dokumentieren aber auf jeder Hierarchieebene die strukturelle Integration der innenliegenden Netzmarken. Wie z. B. in Abbildung 4.7 zu sehen, können auf dieser Ebene auch Kommunikationswege (Verbindungen) zwischen verschiedenen Subkomponenten dargestellt werden.

Durch die isolierte Darstellung jeder einzelnen Systemkomponente bietet jedes einzelne Referenznetzdiagramm zwar keinen Gesamtüberblick über die (hierarchische)

Systemstruktur, wie es viele ADLs versprechen. Allerdings endet dieses Versprechen der ADLs bei sehr großen, offenen oder nicht-hierarchisch organisierten Systemen aufgrund des Umstandes, dass ein vollständiges Diagramm nicht mehr übersichtlich auf begrenztem Platz unterzubringen wäre. Taylor, Medvidović und Dashofy empfehlen bei zu komplexen Systemen [TMD10, Abschnitt 6.6, S. ] verschiedene Techniken zur Reduktion der Komplexität, unter anderem das Modellieren einzelner Instanzen, die Ausnutzung von Symmetrien oder das Modellieren von Protokollen. Die in dieser Arbeit praktizierte Modellierung mit Referenznetzen setzt auf genau diese Techniken.

In dieser Arbeit ist mit dem Plugin-Konzept im Wesentlichen *ein* Architekturstil modelliert worden. Dennoch eignet sich die Modellierungstechnik im Sinne der *Heterogenität* auch für andere Architekturstile wie z. B. die MULAN-Referenzarchitektur (siehe Abschnitt 2.5.2) beweist. Aufgrund der formal definierten Semantik von Petrinetzen ist auch eine *Analyse* von Eigenschaften der Architektur denkbar. Es ist allerdings anzumerken, dass noch keine speziell auf den Formalismus der Referenznetze ausgelegten Analysewerkzeuge existieren.

Im Vergleich zu den in Unterabschnitt 2.3.5 vorgestellten ADLs, welche ebenfalls Petrinetze nutzen, ist zu dieser Modellierungstechnik zweierlei anzumerken: Zum einen zählt die hier vorgestellte Technik offensichtlich zu den Ansätzen, welche ausschließlich mit Petrinetzen arbeiten. Die Netze werden also nicht nur als ergänzende Notation zur Verhaltensspezifikation eingeführt. Zum anderen erlaubt die hier vorgestellte Technik eine dynamische Rekonfiguration der Architektur. Die Änderung der Systemkonfiguration ist zwar nur sehr eingeschränkt möglich, da in jeder Komponente die Integrationsmöglichkeit für andere Komponenten (Plugins bzw. Erweiterungen) in Form von Netzelementen statisch kodiert ist. Ebensowenig können die Schnittstellen von Komponenten dynamisch verändert werden. Das Instantiieren von Netzen und die Möglichkeit, Netzexemplare dynamisch als Marken auf den Konfigurationsstellen hinzuzufügen und zu entfernen, erlauben aber eine flexible dynamische Änderung der Architektur in diesen Grenzen.

Im Vergleich zu anderen dynamischen ADLs (siehe Unterabschnitt 2.3.4) ist daher anzumerken, dass die hier verwendete Technik zwar nur wenig mehr Operationen als z. B. Darwin bietet. Die Operationen erfordern jedoch keine Skriptsprache oder externe API, da sie in Form von Transitionen grafisch im selben Formalismus spezifiziert werden.

Aufgrund der operationalen Semantik von Referenznetzen eignet sich das Modell auch dazu, Plugin-Systeme zu implementieren. Die Implementierung erfordert keinen Wechsel der Programmiersprache, da die Referenznetze durch Java-Anschriften zu voll funktionsfähigen Softwaresystemen ausgebaut werden können. Eine prototypische Umsetzung des Modells auf verschiedenen Wegen findet im nächsten Kapitel statt.

# 5 Prototypische Umsetzung

Die Modellierung der im vorigen Kapitel präsentierten konzeptionellen Netze wurde fortlaufend von einer Entwicklung mehrerer Prototypen begleitet, um Erfahrungen sammeln, Erkenntnisse gewinnen, Konzepte ausprobieren und veranschaulichen zu können. Die Prototypen basieren auf verschiedenen Programmiersprachen bzw. Laufzeitumgebungen. In diesem Kapitel sollen einige der Prototypen in der chronologischen Reihenfolge ihrer Entstehung präsentiert werden.

Um die prototypisch implementierten Plugin-Architekturen ausgiebig testen zu können, wird ein Softwaresystem benötigt, welches in mehreren Dimensionen unabhängig und rekursiv erweiterbar ist. Für diesen Zweck eignet sich das bereits zur Modellierung und Ausführung der Petrinetz-Modelle eingesetzte Werkzeug Renew, welches zu Beginn der Arbeiten eine Komplexität erreicht hatte, welcher die monolithische Softwarearchitektur nicht mehr gerecht wurde. Die Ausgangssituation und der Umbau von Renew in ein erweiterbares System werden in Abschnitt 5.1 erläutert.

Zunächst entstand der in Abschnitt 5.2 vorgestellte, rein Java-basierte Prototyp einer Plugin-Verwaltung, welcher inzwischen mit mehreren Renew-Versionen ausgeliefert wurde. Ein Schwerpunkt bei der Entwicklung dieses Prototypen war neben der dynamischen Rekonfigurierbarkeit auch die Unterstützung von Konfigurationsänderungen außerhalb der Laufzeit des Systems.

Parallel zur konzeptionellen Modellierung entstanden mehrere Prototypen auf Basis der fortschreitenden Verfeinerung der Netze, also dem Ansatz „Implementierung durch Spezifikation“ folgend. Die im Renew-Simulator ausführbaren Referenznetze werden in Abschnitt 5.3 vorgestellt. Einer der Prototypen integriert sich in die Plugin-Verwaltung des zuerst erstellten rein Java-basierten Prototypen.

Gegen Ende der Arbeit entstand aus einigen technischen Unzulänglichkeiten des Java-Prototypen im Bereich des dynamischen Ladens von Klassen heraus der Wunsch nach einer ausgereiften dynamischen Modulverwaltung. Der in Abschnitt 5.4 vorgestellte Prototyp baut daher auf dem bereits im Kontext der Eclipse-Plattform beschriebenen OSGi-Rahmenwerk (siehe Unterabschnitt 3.2.1) auf. Da das OSGi-Rahmenwerk anstelle der Erweiterungsbeziehung auf die Dienstbeziehung zur Kopplung von Modulen setzt, ist die Modellierung von Erweiterungsbeziehungen auf Basis der Dienstbeziehungen ein wesentliches Thema dieses Prototypen.

Eine vergleichende Diskussion der Prototypen und der aus ihnen gewonnenen Erkenntnisse in Abschnitt 5.5 rundet das Kapitel ab.

## 5.1 Fallstudie Renew

Das Referenznetzwerkzeug Renew [KWD06a], welches auch im Rahmen dieser Arbeit zur Modellierung der Petrinetze und zum Zeichnen sämtlicher Abbildungen verwendet

wurde, befand sich zum Zeitpunkt des Beginns dieser Arbeit in einer Situation, welche die Einführung eines Plugin-Systems nahe legte. Im Folgenden soll diese Situation kurz geschildert werden, um anschließend auf die nötigen Umbauten zur „Pluginifizierung" der Anwendung einzugehen.

### 5.1.1 Motivation

Es lassen sich mehrere Gründe ausmachen, warum Renew in der damals aktuellen Version 1.6 von der Einführung eines Plugin-Systems profitieren musste:

**Viele Features** Die damals aktuelle Version 1.6 der Open-Source-Software wurde als monolithische Anwendung ausgeliefert. Diese umfasste einen allgemein verwendbaren Petrinetzsimulator, den Formalismus der Referenznetze sowie einige weitere experimentelle Petrinetzformalismen. Hinzu kam ein grafischer Editor für Petrinetze und andere Strichgrafiken. Mit dem Simulator und dem Editor waren somit bereits zwei Funktionalitätsbereiche in der Software enthalten, die auch unabhängig voneinander einen Nutzen hätten entfalten können.

Mit jeder vorangegangenen Version war die Menge der in die Software integrierten „Features" gewachsen. In den Editor wurden viele Fähigkeiten integriert, welche die Modellierung von Petrinetzen vereinfachten, die dynamische Visualisierung der Modelle verfeinerten oder zusätzliche grundlegende, Petrinetz-unabhängige grafische Zeichenwerkzeuge umfassten. Der Simulator hatte unter anderem eine Datenbankanbindung und eine Schnittstelle zur Beobachtung des Simulationszustandes über ein Netzwerk erhalten.

Nur für interne Entwickler zugänglich befand sich im Quellcode-Repository weitere Funktionalität, unter anderem eine petrinetzbasierte Workflow-Engine und die petrinetzbasierte Agentenplattform Capa. Beide bauten auf dem Renew-Code auf, ohne ihn unmittelbar zu verändern, nutzten aber in Renew vorhandene Schnittstellen zur Integration ihrer Funktionalität, so dass sie beim Start von Renew zur Verfügung stehen konnten.

**Unterschiedliche Anwenderprofile** Nicht jeder Renew-Anwender nutzte und nutzt dieselben Funktionen. Grundsätzlich sind zwei Interessengruppen in der Anwenderschaft von Renew zu unterscheiden (welche allerdings weich ineinander übergehen, wie die vorliegende Arbeit beweist, die zu beiden Gruppen zu zählen ist): Anwender mit eher *theoretischem Interesse* nutzen die Möglichkeit, Petrinetze zu modellieren, welche mit schwarzen Marken, Zeichenketten, Zahlen, Tupeln und Listen auskommen und allenfalls ein paar grundlegende algebraische Funktionen zur Ermittlung möglicher Schaltmodi benötigen. Anwender mit eher *praktischem Interesse* nutzen den Java-Referenznetzformalismus zur Softwareentwicklung, z. B. zur Kontrollflusssteuerung nebenläufiger Systeme oder zur Implementierung ganzer Anwendungen, und verwenden Objekte auf Basis beliebiger Java-Klassen. Während theoretisch interessierte Anwender das Markenspiel zur Visualisierung oder Analyse ihrer Modelle beobachten, benötigen praktisch interessierte Anwender eine *Integrierte Entwicklungsumgebung (IDE)* mit Fähigkeiten zum Ausführen und Debuggen der Software. Die Fähigkeiten

zum Betrieb des Simulators ohne grafische Oberfläche und zur ausfallsicheren Speicherung des Simulationszustandes in einer Datenbank sind in erster Linie für den Einsatz als Produktivsystem interessant. Sie werden weder in der theoretischen Modellierung noch in der prototypischen Softwareentwicklung benötigt.

Die bereits erwähnte unveröffentlichte Funktionalität wie die Workflow-Engine oder die Agentenplattform ist überwiegend für die praktisch interessierten Anwender von Interesse. Hier deutet sich eine weitere Unterteilung der Anwendergruppen an, da Workflows und Agenten häufig als voneinander unabhängige Disziplinen der informatischen Forschung betrachtet werden. Im monolithischen Renew wurde die unveröffentlichte Funktionalität jeweils beim Zusammenstellen der Auslieferungsversion aus dem Dateibaum entfernt, da sie die Anwendung vergrößerte und verlangsamte. Zudem gestaltete sich die Konfiguration der Anwendung komplizierter, wenn die unveröffentlichte Funktionalität integriert war. Vor allem aber war diese Funktionalität nur für praktisch interessierte Anwender in einem sehr speziellen Anwendungsbereich von Nutzen.

**Unterschiedliche Lizenzen** Bei der bei Auslieferung aus dem System entfernten Funktionalität handelte es sich zudem um Software, deren Veröffentlichungsstrategie noch nicht feststand. Die von Renew verwendete Open-Source-Lizenz „Gnu Lesser General Public License“ [LGPL, FSF99] erzwingt zwar eine Weiterverwendung derselben Lizenz für abgeleitete Arbeiten, aber nicht für Software, welche Renew unmodifiziert nutzt. Sowohl die Workflow-Engine als auch die Agentenplattform bauen auf einem unmodifizierten Renew-Kern auf, wären also separat lizenzierbar. Eine gebündelte Auslieferung dieser Softwarepakete hätte aber das separate Lizenzieren erschwert, da sie äußerlich als Teil von Renew erschienen wären.

### 5.1.2 Ausgangssituation

Renew verfügte in Version 1.6 über eine Softwarearchitektur, welche entlang mehrerer Java-Packages gegliedert war. Die Software umfasste ca. 900 Klassen, welche in rund 30 Packages unterschiedlicher Größe unterteilt waren. In Abbildung 5.1 ist die Architektur schematisch dargestellt. Die Namen der Java-Packages, welche den jeweiligen Baustein ausmachen, sind in `Schreibmaschinenschrift` notiert.

Die Farbbereiche im Bild visualisieren grob eine im System vorhandene horizontale Schichtenarchitektur. Die untersten Schichten werden vom Betriebssystem bzw. der plattformunabhängigen virtuellen Java-Maschine gestellt, direkt darüber sind einige verwendete Packages der Java-Klassenbibliothek aufgeführt. Auf dieser Java-Schicht setzt ein großer, in verschiedenen Grüntönen eingefärbter Block auf, welcher den Simulator inklusive Referenznetzformalismus beinhaltet. Obwohl innerhalb dieses Blockes sehr wohl einige Schichten ausgemacht werden können, sind die einzelnen Teile doch auf mehr Wegen untereinander verzahnt, als es die Grafik suggeriert – eine genauere Betrachtung folgt weiter unten. Zuoberst ist die orange eingefärbte grafische Oberfläche dargestellt, welche auf dem vom Simulator-unabhängigen Grafik-Framework JHotDraw (links dargestellt) aufsetzt. Die hellgrün eingefärbte Schicht mit den Elementen

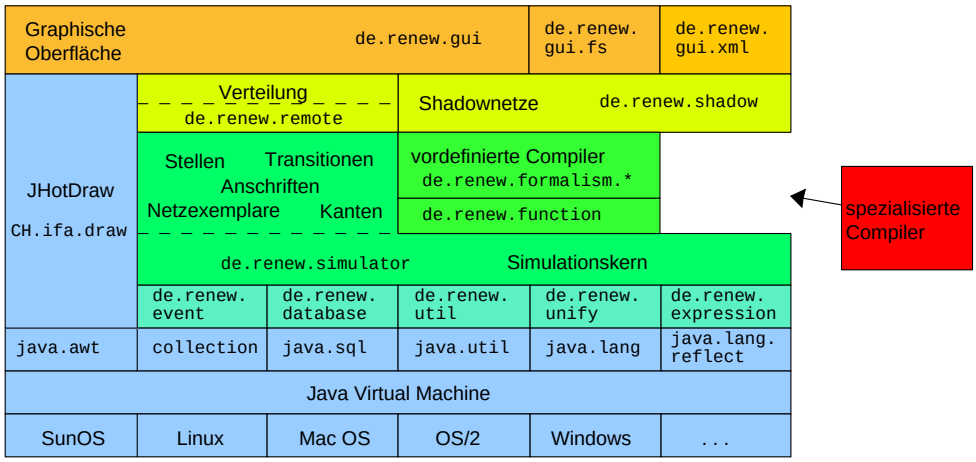



Abbildung 5.1: Schematischer Überblick der Renew-Architektur in Version 1.6 [aus Kum02b]

„Verteilung“ und „Shadownetze“ stellt die Schnittstelle des Simulationskerns zum grafischem Editor dar: Im Editor erstellte Petrinetze werden in Form so genannter *Schattennetze* an den Simulator übergeben, und die Beobachtung des Simulationszustandes ist über das Package `de.renew.remote` möglich. Renew kann mittels *spezialisierter Compiler* auf andere Petrinetzformalismen als den Java-Referenznetzformalismus eingestellt werden.

In Abbildung 5.2 sind sämtliche in der ausgelieferten Version von Renew enthaltenen Java-Packages mit ihren Abhängigkeiten dargestellt. Die beiden großen Rahmen repräsentieren die zwei verwendeten Präfixe im Package-Namensraum, verdeutlichen damit aber zugleich die Trennung der eigentlichen Renew-Klassen vom integrierten JHotDraw-Editor (rechts dargestellt). In dieser Abbildung werden die vielfältigen Zusammenhänge zwischen den Packages sichtbar, welche in Abbildung 5.1 sauber getrennt erscheinen. Zum Verständnis der folgenden Arbeiten zur Vorbereitung der Plugin-System-Einführung ist vor allem die Lage des Packages `shadow` relevant, welches die Datenstrukturen für Schattennetze und die von Compilern zu implementierende Schnittstelle definiert. Das Package `shadow` ist *zwischen* den `formalism`-Packages und dem `simulator`-Package angeordnet, da die zum jeweiligen Formalismus gehörenden Compiler die Schattennetze als Eingabeformat verarbeiten können müssen, und das Interface `ShadowCompiler` die Erstellung von Simulationsobjekten auf Basis der Schattennetze vorsieht. Das Package `application` enthält nur wenige Klassen zum Aufsetzen und Steuern des Simulators ohne grafische Oberfläche. Die Gesamtheit der Packages von der Reihe `event`, `simulator`, `database` bis einschließlich zum Package `unify` wird im Rahmen dieser Arbeit als eine Einheit („der Simulator“) betrachtet und nicht weiter aufgesplittet werden. Allerdings hat Olaf Kummer parallel zur Einführung des Plugin-Systems das zentrale Package `simulator` refaktorisiert und dabei die Auflösung einiger Abhängigkeiten erreicht.

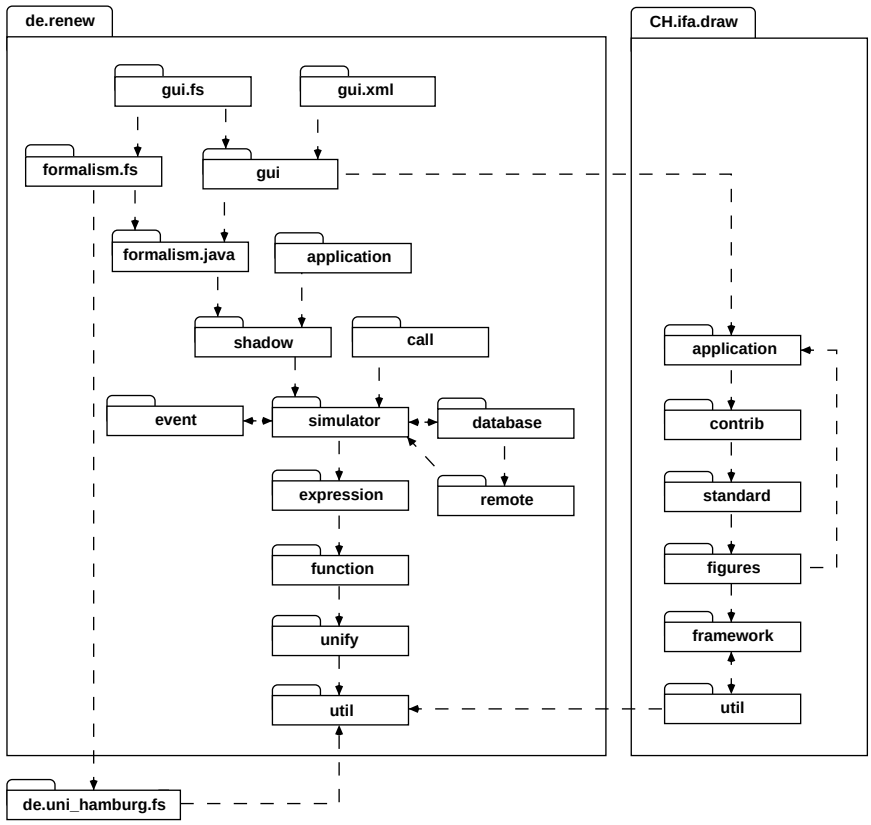



Abbildung 5.2: RENEW-Package-Abhängigkeiten in Version 1.6 [aus KWD01, Figure 2.1]

### 5.1.3 Vorhandene Erweiterungsmöglichkeiten

In Abbildung 5.1 ist die Möglichkeit besonders hervorgehoben, *spezialisierte Compiler* in RENEW zu integrieren. Die Schnittstellen in RENEW, welche diese Integration ermöglichen, stellen offensichtlich eine Erweiterungsspezifikation im Sinne des im Kapitel 4 entwickelten Plugin-Konzepts dar, da die Anwendung an dieser Stelle *intendiert erweiterbar* ist. Zum Zeitpunkt der Veröffentlichung von RENEW 1.6 existierten neben dem vordefinierten Java-Referenznetzcompiler im Wesentlichen mehrere Compiler für Varianten der Feature-Structure-Netze [siehe Wie01, DRW06] und ein Compiler für Workflow-Netze [Jac02, JKMUN02]. In den vergangenen Jahren wurden z. B. noch ein Compiler für Service-Description-Netze [siehe KO05], für Maria-Netze [siehe DF06] und für Scheme-Referenznetze [siehe DF07] ergänzt.

Aufgrund der regen Nutzung der Erweiterungsmöglichkeit durch spezialisierte Compiler ist offensichtlich, dass sie im Plugin-System erhalten bleiben muss. Für Erweiterungspezifikationen typisch lässt sich in RENEW keine einzelne streng gekapselte

Schnittstelle ausmachen, welche alle Aufgaben des Compilers vollständig abdeckt. Ein Compiler ist zwar gegen das Interface `ShadowCompiler` zu implementieren, dieses legt aber nur fest, wie der Renew-Simulator Netze zum Übersetzen an den Compiler übergeben kann. Die Anwendung stellt dem Compiler darüber hinaus viele Klassen in etlichen Packages des Renew-Simulationsblocks zur Nutzung zur Verfügung. Ein Formalismus kann die bestehende Klassenhierarchie durch weitere Klassen mit spezialisierter Funktionalität ergänzen, deren Integration in der Regel durch Verwendung dieser Klassen im Compiler erfolgt. Um Netze mit spezialisierter Syntax für den spezialisierten Compiler zeichnen zu können, muss unter Umständen auch die grafische Oberfläche um entsprechende Werkzeuge erweitert und das Schattennetz-Format um spezielle Datenstrukturen ergänzt werden.

Renew bot in Version 1.6 keine statische oder dynamische Verwaltung von verfügbaren Compilern als Erweiterungen, sondern nur die Möglichkeit, die Anwendung beim Start mittels eines Objekts vom Typ `de.renew.gui.RenewMode` zu parametrisieren. Die grafische Oberfläche entnahm diesem Objekt Informationen über den zu verwendenden Compiler, verfügbare grafische Werkzeuge und den einzustellenden Betriebsmodus des Simulators. Im reinen Simulationsbetrieb ohne grafische Oberfläche stand dieses `RenewMode`-Objekt nicht zur Verfügung. Stattdessen wurde der Simulator mit einem Schattennetz-System bestückt, welches die Angabe des zu verwendenden Compilers beinhaltete. Der Betriebsmodus des Simulators wurde explizit beim Aufruf angegeben.

Das dem Editor zugrunde liegende Rahmenwerk *JHotDraw* stellte und stellt im Wesentlichen eine Klassenbibliothek dar, welche grafische Zeichnungselemente und Manipulationswerkzeuge umfasst. Die Zusammenfassung dieser Elemente und Werkzeuge zu einem Editor geschieht auf monolithische Weise in Form einer zentralen, umfangreichen Klasse, welche die gesamte Anwendung repräsentiert. Eine Erweiterungsverwaltung oder Konfigurationsschnittstellen zum Hinzufügen und Entfernen von Elementen oder Werkzeugen sind nicht vorhanden. Renew ergänzt seine petrinetzspezifischen Elemente und Werkzeuge, indem es eine Subklasse der Anwendung anstelle der ursprünglichen Klasse verwendet. Die Subklasse überschreibt diverse Methoden, um die Renew-Elemente und -Werkzeuge in die Anwendung zu integrieren. Im monolithischen Renew bot diese Subklasse zudem über das bereits genannte `RenewMode`-Objekt einen primitiven Mechanismus zur Integration von formalismusspezifischen Werkzeugen als Erweiterungen.

Aufgrund der Verwendung von Renew als IDE war und ist es notwendig, dass Anwender beliebige Klassen, die spezifisch für die entwickelte Anwendung sind, innerhalb einer Simulation mittels Anschriften im Java-Referenznetzformalismus nutzen können. Anwender konnten eigene Klassen in Renew-Version 1.6 nur bekannt machen, indem sie vor dem Start von Renew den Java-Classpath entsprechend ergänzten. Renew bot darüber hinaus im „Class-Reinit-Mode“ einen eigenen Java-Classloader, welcher alle Klassen außerhalb der Renew-Package-Hierarchie bei jedem Simulationsstart vergaß und (bei Bedarf) neu lud.

Die bereits angesprochene Workflow-Engine nutzte und nutzt alle beschriebenen Möglichkeiten zur Erweiterung von Renew. Ein spezialisierter Compiler übersetzt so genannte „Task“-Transitionen in normale Transitionen und Stellen, welche im Renew-Kern bereits implementiert sind. Die grafische Oberfläche erhält ein zusätzliches Werk-

zeug zum Erstellen dieser Task-Transitionen. Zudem wurde im monolithischen Renew die Klasse `Server` des `remote`-Packages, welche als Repräsentant der laufenden Simulation diente (und immer noch dient), durch eine Subklasse überlagert, welche zusätzlich Zugriff auf die Rollen-, Rechte- und Task-Definitionen und die Aufgabenlisten der Workflow-Engine bot.[1] Die Verbindung der Elemente zu einer nutzbaren Anwendung geschah über ein mit entsprechenden Referenzen belegtes `RenewMode`-Objekt sowie durch die Auswertung einiger beim Start der Anwendung konfigurierbarer Attribute, welche mittels der in Java angebotenen `Properties`-Klasse übertragen wurden. Anwendungsspezifische Task-Definitionen konnten in Form von Java-Klassen über den Java-Classpath oder den Renew-Classloader bereitgestellt werden.

### 5.1.4 Dekomposition

Während der Entwicklung der ersten prototypischen Implementierung eines Plugin-Systems in Renew war es notwendig, die einzelnen Funktionsblöcke im Softwaresystem so voneinander zu separieren, dass sie als eigenständige Systemkomponenten ausgeliefert werden können. Ohne eine solche Dekomposition wäre die Umstellung der Software auf eine reine Plugin-Architektur nicht sinnvoll, da keine bereits integrierte Funktionalität ausgegliedert werden könnte. Die Dekomposition fand in Zusammenarbeit mit Jörn Schumacher im Rahmen seiner Diplomarbeit statt, siehe [Sch03, Kapitel 8] und wurde seitdem in kleinen Schritten bei Bedarf überarbeitet.

Die zentrale Metapher bei der Identifikation von möglichen Schnittstellen und Subsystemgrenzen im dekomponierten Renew ist die intendierte Erweiterbarkeit. Aus der vorangegangenen Beschreibung der Renew-Architektur ergeben sich mit dem Simulator und dem grafischen Editor zwei größere Blöcke, welche auch unabhängig voneinander verwendbar sein sollen. In beiden Blöcken werden Erweiterungsspezifikationen benötigt, um spezialisierte Compiler nebst passender grafischer Elemente und Werkzeuge in die Anwendung integrieren zu können.

In Abbildung 5.3 ist die während der Dekomposition grundsätzlich verfolgte Strategie dargestellt. Editor und Simulator stehen prinzipiell nebeneinander, allerdings kann der Editor den Simulator steuern. Jeder als Erweiterung zu integrierende Formalismus wird in einen ausführbaren Teil mit Compiler, Ausdrücken usw. und einen grafischen Teil mit Editor-Elementen und Werkzeugen unterteilt. Auch hier gilt, dass die grafischen Elemente und Werkzeuge über ihre Abbildung in die Domäne des Compilers Bescheid wissen müssen, so dass die Teile nicht vollständig unabhängig sind.

Eine Stärkung der Unabhängigkeit des Editors vom Simulator wäre z. B. nach dem *Bridge*-Pattern (siehe Unterabschnitt 2.4.4) möglich. Allerdings würde dies die Dekomposition des vorhandenen Codes in drei Plugins je Schicht erfordern. Die vorhandene Codebasis war nicht auf eine solche Trennung ausgerichtet, so dass die Beibehaltung der Abhängigkeit des Editors vom Simulator den einfacher realisierbaren Weg darstell-

[1] Im Zuge der Dekomposition wurde die softwarearchitektonisch unsaubere Lösung, das Server-Repräsentantenobjekt zu überlagern, durch eigene Schnittstellen zum entfernten Zugriff auf Daten des Workflow-Systems ersetzt. Die anderen von der Workflow-Engine genutzten Erweiterungsmöglichkeiten wurden während der Umstellung der Renew-Softwarearchitektur beibehalten oder auf die in den folgenden Unterabschnitten vorgestellten Erweiterungsspezifikationen mit ähnlichem Funktionsumfang umgestellt.

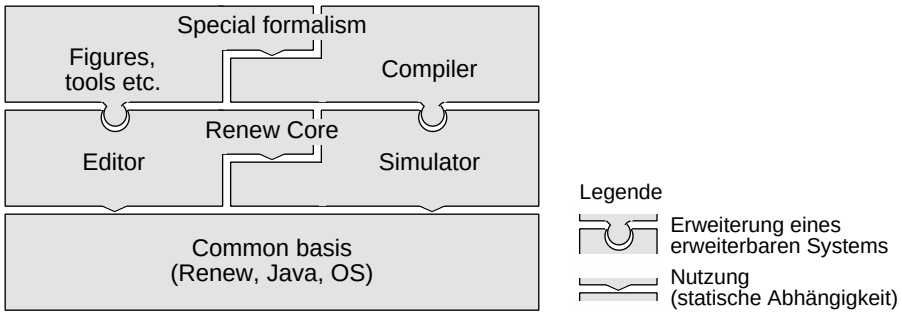



Abbildung 5.3: Grundsätzliche Dekompositionsrichtung in Renew 2.0

te. Eine Modellierung der Beziehung zwischen Editor und Simulator als Dienstbeziehung entsprach ebenfalls nicht dem vorhandenem Code des Editors, der sehr spezifisch auf die Steuerung des Renew-Simulators ausgelegt war und ist.

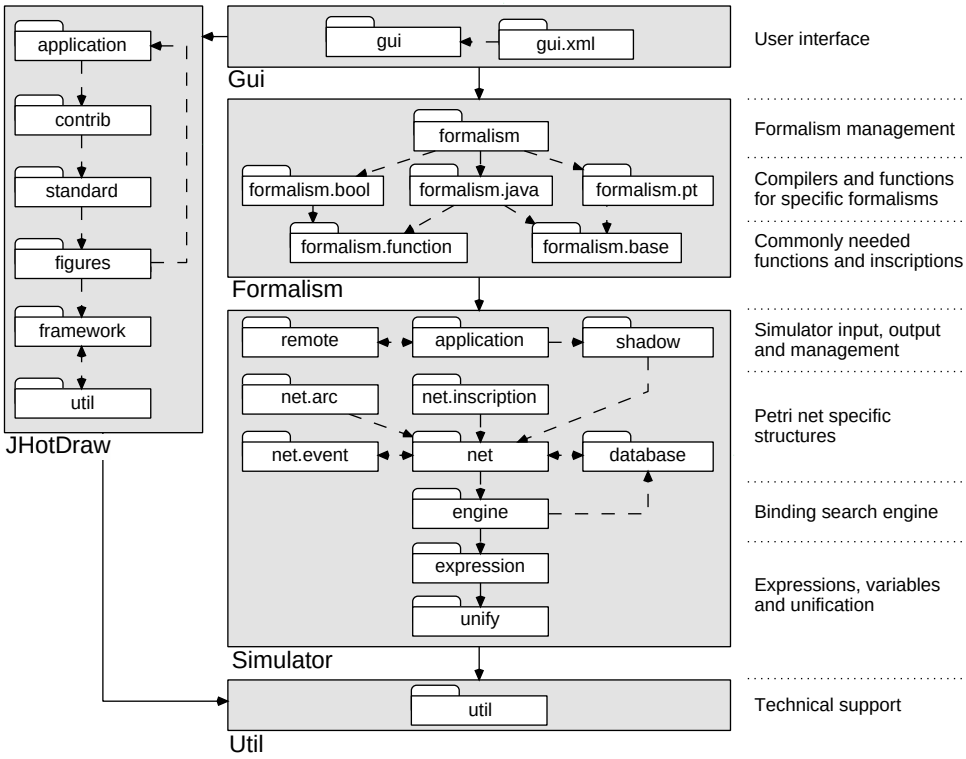



Abbildung 5.4: Renew-Plugin-Dekomposition in Version 2.0 [basierend auf Sch03, S. 91]

Aus der vorangegangenen Evaluation und den in Abbildung 5.2 dargestellten Package-Abhängigkeiten ergibt sich die in Abbildung 5.4 dargestellte Aufteilung der Packages in Plugins. Die Zeichnung stellt ausschließlich statische Abhängigkeiten zwischen Packages dar, wobei sämtliche Beziehungen ausgeblendet sind, welche sich transitiv über andere Beziehungen nachvollziehen lassen. Gestrichelte Linien visualisieren Abhängigkeiten zwischen Packages innerhalb eines Plugins. Die Abhängigkeiten zwischen Packages unterschiedlicher Plugins sind vergröbert in Form einer durchgezogenen Linie zwischen den betroffenen Plugins dargestellt. Die grauen Kästen repräsentieren je ein Plugin, dessen Name unter dem Kasten notiert ist.

Die Abbildung bezieht sich auf die RENEW-Package-Struktur *nach* Abschluss der bereits angesprochenen Refaktorisierung des Simulator-Kerns durch Olaf Kummer. Zudem sind einige kleinere zyklische Abhängigkeiten, die der Dekomposition in Plugins mit einer unidirektionalen Abhängigkeitsrichtung im Wege standen, im Rahmen der Diplomarbeit von Jörn Schumacher aufgelöst worden [siehe Sch03, Abschnitt 8.4].

Das umfangreiche Package `simulator` ist nun in mehrere Packages aufgeteilt, so dass die innere Schichtung in eine allgemeine Simulations-Engine mit Bindungssuche (Package `engine`) und eine Petrinetz-spezifische Abstraktionsschicht (Packages `net.*`) sichtbar wird. Die Eingabe- und Ausgabeschicht mit den Paketen `shadow` und `remote` sind nun in der Abhängigkeitshierarchie so angeordnet, wie es bereits in der Überblicksdarstellung der Abbildung 5.1 vorgesehen war. Das Package `application` beinhaltet nun anstelle der zuvor auf die Nutzung per Kommandozeile ausgelegten Simulationssteuerung eine Plugin-Fassade, welche anderen Systemkomponenten Möglichkeiten zur groben Steuerung des Simulators bietet und die Konfiguration von Erweiterungen übernimmt.

Die Zuordnung der Packages zu Plugins ergibt sich unmittelbar aus der zuvor besprochenen Dekompositionsrichtung. Das Package `util` bildet ein *Library Plugin* (siehe Unterabschnitt 2.4.4), da seine Klassen von sämtlichen RENEW-Packages inklusive des modifizierten JHotDraw-Rahmenwerkes benötigt werden. Der Simulator bildet das zentrale Plugin der Architektur und wird nicht weiter unterteilt, seine Schnittstelle zur Nutzung durch andere Plugins besteht im Wesentlichen aus den bereits erwähnten obersten drei Packages `remote`, `application` und `shadow`. Als erweiterbares System bietet der Simulator die Möglichkeit, auf globale Zustandsänderungen wie Simulationsstart und -ende mit beliebigen Manipulationen der Simulationsumgebung zu reagieren. Zudem lässt sich das Einspeisen von Netzen in die Simulation als Erweiterung der Simulator-Funktionalität auffassen.

Das Formalism-Plugin umfasst zunächst die neu eingeführte Formalismen-Verwaltung im Package `formalism`, welche die fachliche Erweiterungsverwaltung dieses Plugins im Sinne der intendierten Erweiterbarkeit darstellt. Daneben enthält das Package `function` eine Sammlung nützlicher Klassen, welche beispielsweise Funktionen zur Auswertung mathematischer Ausdrücke oder zum Interagieren mit Java-Objekten bereitstellen. Ebenfalls im Formalism-Plugin enthalten sind der eigentlichen Java-Referenznetzformalismus sowie zwei weitere exemplarische Formalismen. Es besteht eine Abhängigkeit von der Formalismen-Verwaltung zu den einzelnen Formalismen, da die Verwaltung diese drei Formalismen von sich aus integriert. Im Interesse einer sauberen Dekomposition muss dieses Plugin noch feiner unterteilt werden, wie es später auch im Rahmen des OSGi-basierten Prototypen geschieht (siehe Abschnitt 5.4).

Der Editor besteht aus zwei Plugins, dem als Basis dienenden JHotDraw und den Renew-spezifischen Erweiterungen. Da – wie in Unterabschnitt 5.1.3 bereits erwähnt – die zentrale Anwendungsklasse in JHotDraw nicht auf die Verwaltung von Erweiterungen eingerichtet ist und Renew Modifikationen durch Überschreiben von Methoden in einer Unterklasse ergänzt, sind die beiden Plugins sehr eng aneinander gekoppelt. Als erste Schritte in Richtung einer Entkopplung wurden in JHotDraw Erweiterungsschnittstellen zur Verwaltung von Menübefehlen, Werkzeugen und Dateitypen ergänzt. Das Renew-spezifische GUI-Plugin stellt lediglich zwei kleine Erweiterungsschnittstellen bereit, eine zur besonderen Darstellung von Marken und eine zur Erzeugung von Standard-Anschriften an Netzelementen. Alle anderen Erweiterungsmöglichkeiten des Editors sollen durch die für JHotDraw neu definierten Erweiterungsschnittstellen abgedeckt werden.

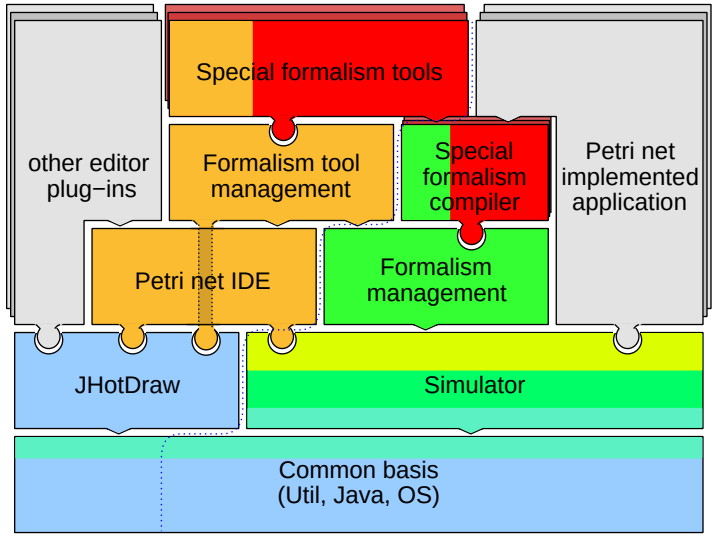



Abbildung 5.5: Beziehungen zwischen zentralen Renew-Plugins in Version 2.0

In Abbildung 5.5 ist das Ergebnis der Dekomposition noch einmal als schematisches Blockdiagramm dargestellt. Zugunsten einer übersichtlicheren zweidimensionalen Darstellung werden nur die wesentlichen Plugins und Beziehungen visualisiert. Die Abbildung basiert auf dem in [CDMR05, Fig. 11] präsentierten Abhängigkeits- und Erweiterungsdiagramm.

Die Blöcke „other editor plug-ins“, „special formalism tools“, „special formalism compiler“ und „Petri net implemented application“ stellen lediglich abstrakte Platzhalter für eine Vielzahl zwischenzeitlich entstandender Plugins dar. Wie schon in Abbildung 5.3 repräsentieren Puzzleteil-artige Nasen Erweiterungsbeziehungen, während die Spitzen für Nutzungsbeziehungen auf Basis einer statischen Abhängigkeit stehen.

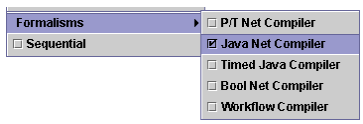
Durch das Diagramm zieht sich von links unten nach rechts oben die dünn gepunktet dargestellte Grenze zwischen der Funktionalität zur Simulation bzw. Ausführung und

der Funktionalität zur grafischen Modellierung bzw. Entwicklung von Petrinetzen bzw. petrinetzbasierten Anwendungen. Auf der Seite des grafischen Editors nutzen fast alle Plugins die neu geschaffenen Erweiterungsschnittstellen in JHotDraw. Auch die direkte Nutzung der JHotDraw-Schnittstellen durch Formalismen-spezifische Plugins ist denkbar, wird aber oftmals aufgrund der Erweiterungsschnittstelle des „Formalism tool management“-Plugins nicht benötigt. Die Erweiterungsschnittstellen des „Petri net IDE“-Plugins (auch „GUI“-Plugin genannt) sind im Bild nicht enthalten, da sie bisher nur sehr selten genutzt werden. Die vorhandenen Erweiterungen gehören in der Regel zu Plugins mit grafischen Werkzeugen für einen speziellen Formalismus.

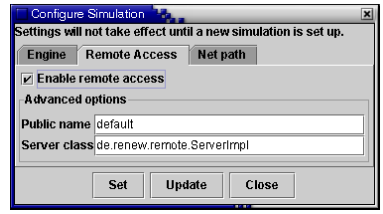
### 5.1.5 Neue Flexibilität

Die ursprüngliche Softwarearchitektur von Renew erlaubte die Konfiguration von Formalismus, Simulator-Betriebsmodus und Werkzeugen im grafischen Editor anhand eines Konfigurationsobjekts, das wie in Unterabschnitt 5.1.3 beschrieben bei Programmstart anzugeben war. Dieses Verfahren wäre prinzipiell auch in einem Plugin-System umsetzbar – statt aus den im monolithischen Renew verfügbaren Klassen könnte das Konfigurationsobjekt nun bei Programmstart eine Auswahl aus sämtlichen von beliebigen Plugins bereitgestellten Klassen treffen. Diese während der Laufzeit unveränderliche Konfiguration entspräche aber nicht der Vorstellung eines dynamisch rekonfigurierbaren Softwaresystems.

Um die für Eclipse propagierte *User Arbitration Rule* [GB03, S. 222] auch in Renew zu befolgen, sollte die Entscheidung zwischen der im System bekannten Plugin-Funktionalität auch zur Laufzeit dem Anwender überlassen werden. Sowohl die Formalismen-Compiler als auch die Betriebsmodi und die grafischen Werkzeuge sind nun über Erweiterungsschnittstellen im Renew-Kern bekannt, so dass es den jeweils erweiterten Systemkomponenten möglich ist, dem Anwender die verfügbaren Optionen in Form eines Dialogs oder Menüs anzubieten.




(a) Auswahl des Formalismus




(b) Konfiguration des Simulators

Abbildung 5.6: Konfigurationsmöglichkeiten in Renew 2.0

In Abbildung 5.6 sind die beiden neuen Konfigurationsmöglichkeiten so dargestellt, wie sie dem Anwender im laufenden Renew präsentiert werden. Die als Plugins installierten Formalismen werden in einem Menü aufgelistet. Der Betriebsmodus des Simulators lässt sich über einen Dialog konfigurieren, der wiederum erweiterbar ist. Plugins können diesem Dialog weitere Reiter mit zusätzlichen Konfigurationsoptionen

hinzufügen und sich damit in die zentral gesteuerte Konfiguration der Simulation einklinken. Die vorhandenen Reiter erlauben eine Kontrolle der Nebenläufigkeit während der Simulation, die Aktivierung und Konfiguration der Simulationsbeobachtung über ein Netzwerk und die Konfiguration von Verzeichnissen, aus denen während der Simulation bei Bedarf Netze nachgeladen werden können [vgl. KWD06b, Abschnitt 4.4.6]. Denkbar und sinnvoll, aber bisher noch nicht implementiert sind weitere Reiter zur Konfiguration des Klassenpfades, aus welchem während der Simulation Anwender-Klassen geladen werden, und zur Konfiguration der Datenbank-Anbindung des Simulators.

Ein Plugin, welches sich hier mit einem weiteren Reiter einklinkt, ist das später bereits im Kontext des Plugin-Systems von Sven Offermann entwickelte Plugin „Logging“, welches als interaktives Debugging-Werkzeug das zuvor auf der Konsole ausgegebene Simulations-Protokoll (*simulation log*) innerhalb der grafischen Oberfläche aufbereitet, so dass eine Navigation zu den beteiligten Netzinstanzen möglich ist. Im „Logging“-Reiter des Konfigurationsdialoges lassen sich z. B. die Größe des Ereignispuffers und der zu beobachtende Bereich der Simulation einstellen.

Die auf die Erweiterungsspezifikation ausgerichteten grafischen Konfigurationswerkzeuge erleichtern dem gelegentlichen Anwender die Auswahl der gewünschten Optionen und bieten auch dem erfahrenen Anwender die Möglichkeit zur dynamischen Rekonfiguration des Systems. Allerdings benötigt ein regelmäßiger Anwender üblicherweise über mehrere Laufzeiten der Anwendung hinweg eine gleichbleibende Konfiguration. Es ist dem Anwender nicht zumutbar, nach jedem Neustart der Anwendung eine immer gleiche Folge von Konfigurationsschritten abarbeiten zu müssen, bis Formalismus und Simulator im gewünschten Modus arbeiten. Außerdem stehen die grafischen Konfigurationswerkzeuge nicht zur Verfügung, wenn eine abgespeckte Systemkonfiguration mit lediglich jenen Plugins laufen soll, welche zur Ausführung eines Schattennetzsystems ohne grafische Oberfläche nötig sind.

Aus den eben genannten Gründen wurde der in Renew bereits für einige Simulationsparameter verwendete Mechanismus auf Basis der Klasse `java.util.Properties` zu einem zentralen und persistent einstellbaren Konfigurationsmechanismus ausgebaut. Die Properties können in der aktuellen Renew-Version auf vier Weisen gesetzt werden [siehe KWD06b, Abschnitt 2.6]. Dauerhafte Einstellungen können in einer Datei im Renew-Installationsverzeichnis oder im Home-Verzeichnis des Anwenders festgelegt werden, wobei die Parameter des Anwenders Vorrang vor den Parametern der Installation haben. Flüchtige Einstellungen können sowohl beim Start des Renew-Systems als auch während der Laufzeit vorgenommen werden. Die flüchtigen Einstellungen haben wiederum Vorrang vor den dauerhaft konfigurierten Parametern. Die zuvor benannten grafischen Werkzeuge sind an den Properties-Mechanismus angebunden, so dass sie wie flüchtige, zur Laufzeit gesetzte Parameter wirken. Zudem spiegelt der Simulationskonfigurationsdialog die aktuell gesetzten Parameter wieder, sobald der „Update“-Schalter betätigt wird.

Um auch ohne grafische Oberfläche flüchtige Änderungen an den Parametern und der Systemkonfiguration vornehmen zu können, ist Renew im Rahmen des ersten Prototyps um eine Kommando-Verarbeitung ergänzt worden, welche mittels verschiedener Plugins sowohl aus der grafischen Oberfläche heraus als auch von der Konsole aus angesprochen werden kann.

## 5.2 Erster Prototyp: Java

Die Software Renew wird seit Version 2.0 mit einer Plugin-basierten Architektur ausgeliefert, welche vollständig in Java implementiert ist. Das Plugin-Management-System (PMS), welches den Kern dieser Architektur bildet, basiert auf einem ersten Prototypen zum in Kapitel 4 entwickelten konzeptionellen Modell von Plugin-Systemen. Die Grundzüge dieses Prototypen sind im Rahmen der Diplomarbeit von Jörn Schumacher [Sch03] entstanden. Der Prototyp ist bis zur Einsatzreife weiterentwickelt worden, so dass er mit Renew 2.0 ausgeliefert werden konnte. Eine Anwender-orientierte Sichtweise auf die weiterentwickelte Version wurde in [KWD+04] präsentiert. Die Abbildung der im Rahmen der Referenznetzmodellierung erarbeiteten Konzepte auf den Prototypen wurde erstmals in [CDMR05] erläutert.

### 5.2.1 Zielsetzung

Die erste prototypische Implementierung eines Plugin-Systems für Renew ist parallel zu den Skizzen der ersten Modellierungsschritte entstanden. Bei der Entwicklung des Prototypen stand daher zunächst das Sammeln von Erfahrungen im Kontext der Realisierung von dynamischen Plugin-Systemen im Vordergrund. Zum Zeitpunkt der Entwicklung des Prototypen war die prominenteste in Java implementierte Software mit einer rein Plugin-basierten Softwarearchitektur, Eclipse, noch nicht dynamisch rekonfigurierbar.

Über das Sammeln von Erfahrungen hinaus sollte der Prototyp zu einem auslieferungsfähigen Plugin-System führen, welches die bereits in Unterabschnitt 5.1.1 angesprochenen Probleme lösen kann. Aus dem Ziel, ein praxistaugliches und anwenderfreundliches Plugin-System zu entwickeln, ergeben sich weitere Anforderungen an die Implementierung:

**Einfache dynamische Rekonfiguration** Der Prototyp soll das Hinzufügen und Entfernen von Plugins während der Laufzeit unterstützen. Diese Anforderungen sind bereits aus den Unterabschnitten 4.1.1 und Unterabschnitt 4.1.2 bekannt.

**Einfache Installation** Der Prototyp soll zusätzlich das Hinzufügen und Entfernen von Plugins in Form einfacher Dateioperationen außerhalb der Laufzeit des Systems unterstützen. Diese Anforderung ergibt sich daraus, dass es z. B. möglich sein soll, eine auf ein bestimmtes Anwenderprofil hin vorkonfigurierte Software-Installation im Dateisystem einzurichten. Die Verwaltung des Plugin-Systems muss daher in der Lage sein, aus einem vorgegebenen Installationsverzeichnis heraus beliebige dort bei Systemstart vorgefundene Plugins in das System zu integrieren.

**Performanz** Die Verwaltung von Plugins soll sich nach Möglichkeit nicht auf das Antwortverhalten der Anwendung im laufenden Betrieb auswirken. Ein gewisser zusätzlicher Aufwand während des Konfigurationsprozesses ist unvermeidbar und akzeptabel, in Phasen ohne Konfigurationsänderung soll das Plugin-System aber möglichst keinen Einfluss auf die Kommunikation zwischen den eigentlichen Objekten der Anwendung haben.

**Threadsicherheit** Renew ist aufgrund seiner fachlichen Bestimmung ein hochgradig nebenläufiges System. Falls aus Netzen heraus zentrale Funktionen – wie z. B. die Konfigurationsschnittstelle der Plugin-Verwaltung – angesprochen werden, so geschieht dies garantiert in anderen Threads als jenem, aus welchem das System initialisiert wird. Nebenläufigkeit muss aber nicht aus der Simulation von Netzen heraus entstehen, sie kann auch – wie bereits in Unterabschnitt 4.1.7 angesprochen – durch beliebige Plugins in das System getragen werden. Die Plugin-Verwaltung muss daher mit nebenläufig aufgerufenen Operationen rechnen und in geeigneter Weise damit verfahren.

**Laufzeit der Anwendung(en)** Der Lebenszyklus des Gesamtsystems soll – wie die Lebenszyklen der individuellen Plugins – entsprechend dem Bedarf des Anwenders gesteuert werden, ohne von diesem eine besondere Aufmerksamkeit zu erfordern. Während die Lebenszyklen einzelner Plugins genau durch die vom Anwender durchgeführten Vorgänge der Installation bzw. Deinstallation bestimmt sind, muss der Lebenszyklus des Gesamtsystems der Laufzeit der „Anwendung“ entsprechen.

Die „Anwendung“ ist in einem generischen Plugin-System nicht eindeutig bestimmt, da jede Systemkomponente dem Anwender eine eigene Benutzungsschnittstelle präsentieren kann. Die Laufzeit des Gesamtsystems sollte die Laufzeit aller einzelnen Benutzungsschnittstellen überdecken. Die Verwaltung des Plugin-Systems muss daher den Plugins eine Möglichkeit bieten, auf die Laufzeit des Systems Einfluss nehmen zu können, ohne die Laufzeit anderer Plugins zu beeinträchtigen.

Die genannten Ziele verlangen zum Teil eine gegensätzliche Ausrichtung der Implementierung. Insbesondere die Anforderung der Erhaltung der Performanz der gesamten Anwendung erzwingt bei der Realisierung der anderen Anforderungen mitunter eine eher pragmatische Herangehensweise. Trotz oder gerade aufgrund des Anspruchs, eine praxistaugliche Implementierung erreichen zu wollen, wurde bei der Entwicklung des Prototypen anstelle einer sauberen, aber umständlichen objektorientierten Umsetzung der architektonischen Vision eine schnellere Variante implementiert, welche nur die nötigste Funktionalität bereitstellt.

### 5.2.2 Erweiterbarkeit durch Entwurfsmuster

Um den ersten Prototypen des Plugin-Systems erstellen zu können, muss die weitgehend monolithisch organisierte Softwarearchitektur von Renew (wie in Unterabschnitt 5.1.4 erläutert) aufgelockert werden. Eine reine Dekomposition des Systems in mehrere Komponenten würde aber nicht zu einem Plugin-System, sondern lediglich zu einer komponentenbasierten Softwarearchitektur führen. Essentiell ist daher auch die Schaffung von Erweiterungsschnittstellen, um die aus der Dekomposition heraus entstandenen Systemkomponenten in ihre neuen Rollen als erweiterbare Systeme und passende Erweiterungen umzugestalten.

Aufgrund der Prämisse, während der Laufzeit möglichst wenig Zeit für die Kommunikation über Plugin-Grenzen hinweg verbrauchen zu wollen, wird die Kopplung

zwischen Erweiterung und erweiterbarem System unmittelbar über Java-Objektreferenzen hergestellt. Dieses Technik benötigt lediglich während des Kopplungs- und Entkopplungsprozesses mehr Rechenzeit als in der monolithischen Softwarearchitektur. Die zuvor direkt kommunizierenden Objekte können dies nach erfolgter Kopplung weiterhin tun.

Der bereits in Abschnitt 4.4.1.2 angesprochene Nachteil dieser Technik liegt darin, dass die zentrale Verwaltung keine direkte Kontrolle mehr über die Lebensdauer der einmal hergestellten Beziehungen hat. Die Verwaltung kann lediglich alle betroffenen Plugins bei einer Rekonfiguration auffordern, Referenzen auf Objekte aus offiziell nicht mehr verfügbaren Plugins freizugeben. Zur erfolgreichen Entkopplung ist das System daher auf die Kooperation aller Plugins angewiesen.

Konsequenterweise wird auch bei der Herstellung der Kopplung auf die Kooperation der Plugins gesetzt. Der Prototyp bietet kein zentral bereitgestelltes Rahmenwerk für Erweiterungsbeziehungen, sondern überlässt die Spezifikation und Implementierung von Erweiterungsspezifikationen vollständig den einzelnen Plugins. Damit ist auch die Aufgabe der fachlichen Integration vollständig auf die Plugins übergegangen – die erweiterbare Systemkomponente stellt die Konfigurationsschnittstelle, das erweiternde Plugin führt im Rahmen seiner Initialisierungsphase die Konfigurationsoperationen aus. Somit kann in jedem Plugin auf die Symmetrie zwischen koppelnden und entkoppelnden Konfigurationsphasen geachtet werden.

Der Verzicht auf ein zentrales Rahmenwerk für Erweiterungsbeziehungen folgt aus der Erkenntnis, dass die schon vor der Dekomposition in Renew enthaltenen Erweiterungsmöglichkeiten mittels sehr unterschiedlicher Techniken realisiert sind. Um möglichst wenig Änderungen an prinzipiell funktionierendem Code vornehmen zu müssen, wird die technische Ausgestaltung der fachlichen Schnittstellen jeder Erweiterungsbeziehung der jeweiligen erweiterbaren Systemkomponente überlassen. Damit ist jede erweiterbare Systemkomponente – wie im konzeptionellen Modell vorgesehen – vollkommen frei in der Gestaltung ihrer (fachlichen) Konfigurationsschnittstelle für Erweiterungen und kann die Schnittstelle auf die charakteristischen Merkmale der jeweiligen fachlichen Erweiterbarkeit hin ausrichten.

Dennoch treten bei der Implementierung von Konfigurationsschnittstellen immer wieder ähnliche Aufgabenstellungen auf, so dass sich die Verwendung von Entwurfsmustern wie den in den Unterabschnitten 2.4.1 und 2.4.2 besprochenen anbietet. Sämtliche Erweiterungsspezifikationen der zentralen Renew-Plugins greifen auf ein oder mehrere Entwurfsmuster zurück, um die jeweiligen Erweiterungsbeziehungen zu implementieren. Schumacher erläutert in [Sch03, Abschnitt 9.1] die Verwendung der bekannten Entwurfsmuster Observer, Chain of Responsibility und Command zur Ausgestaltung von Erweiterungsbeziehungen.

Das *Command*-Pattern wird beispielsweise seit jeher in JHotDraw zur Implementierung von Manipulationen an Zeichnungsobjekten eingesetzt. Die neu erstellte Erweiterungsschnittstelle zur Integration von Funktionalität in das Menü der Anwendung akzeptiert daher Command-Objekte. Eine Variante des *Chain-of-Responsibility*-Pat-

terns[2] wird im Petrinetz-spezifischen Teil des Editors verwendet, um die Darstellung konkreter Marken durch Erweiterungen durchführen zu lassen.

Das *Observer*-Pattern ist das unspezifischste der von Schumacher verwendeten Entwurfsmuster, da es auch nicht-intendierte Erweiterbarkeit zulässt. Sowohl der Simulatorkern als auch die grafische Ebene der Formalismenverwaltung bieten in ihrer Fassade eine Beobachtungsmöglichkeit für Zustandsänderungen. Diese Beobachtungsmöglichkeit darf von Erweiterungen genutzt werden, um die Konfiguration des beobachteten Plugins zu verfeinern, indem beispielsweise der Betriebsmodus des Simulators, die Menge der zu kompilierenden Netze oder die Menüstruktur des Editors angepasst wird. Da auf diese Weise der Konfigurationsprozess des Simulators bzw. der Formalismenverwaltung um zusätzliche Operationen erweitert werden kann, stellt das *Observer*-Pattern tatsächlich den Verwaltungsteil einer Erweiterungsschnittstelle dar.

Ergänzt wird dieses von Jörn Schumacher in [Sch03, Abschnitt 9.1] bereits erläuterte Spektrum durch weitere Entwurfsmuster, die teilweise schon im monolithischen Renew oder auch erst in der Weiterentwicklung des dekomponierten Renews zum Einsatz kamen. Das in Unterabschnitt 5.1.3 bereits angesprochene Interface `ShadowNetCompiler`, dessen Implementierungen beliebige Petrinetz-Formalismen in Renew einbringen können, ist de facto eine komplexe Anwendung des *Abstract-Factory*-Patterns. Jeder Compiler nimmt eine Menge an Schattennetzen inklusive der Stellen, Transitionen, Kanten und Anschriften entgegen und instantiiert auf Basis dieser Information eine Vielzahl an Objekten, welche im Rahmen des Simulators das den Netzen entsprechende Verhalten implementieren.

Wie bereits in Tabelle 2.2 überblicksweise dargestellt, kommt kaum ein Plugin-System ohne das *Registry*-Pattern aus. Auch im Renew-Plugin-System wird das Pattern häufig verwendet. Unter anderem pflegt die zentrale Plugin-Verwaltung eine Registry der installierten Plugins, um zu integrierenden Plugins auf Anfrage Referenzen auf passende erweiterbare Systemkomponenten zu liefern. JHotDraw wurde um eine Registry für Typen von Zeichnungen ergänzt, so dass Plugins für die Softwareentwicklung im Rahmen der Paose statt Petrinetze auch Use-Case- oder Interaktionsdiagramme integrieren können. Die inzwischen in mehreren Plugins wie z. B. dem Simulator, der Workflow-Engine und auch im Kontext der Agentenplattform Capa vorhandene Fähigkeit der entfernten Beobachtung und/oder Steuerung eines laufenden Systems ist durch eine zentrale Registry für angebotene Zugriffsmöglichkeiten vereinheitlicht worden. So muss ein Anwender unabhängig von der Anzahl der in der aktuellen Systemkonfiguration angebotenen Netzwerkprotokolle nur einmal die Verbindung zwischen Simulation und grafischer Oberfläche herstellen, um alle Zugriffsmöglichkeiten nutzen zu können.

Da im Renew-Plugin-System die Aufgabe der integrierenden Konfiguration bei den erweiternden Plugins liegt, benötigen diese eine Zugriffsmöglichkeit auf die Konfigu-

[2] Das Entwurfsmuster *Chain of Responsibility* sieht laut [GHJV98, S. 223ff.] vor, die möglicherweise zuständigen Objekte in einer verketteten Liste zu organisieren, so dass jedes Objekt in der Kette den Aufruf weiterleiten kann und muss. Die hier verwendete Variante organisiert die Objekte stattdessen als Menge, deren Elemente der eigentliche Aufrufer der Reihe nach explizit anspricht, bis eines sich für zuständig erklärt. Im konkreten Fall entlastet diese Version die Objekte von der Weiterleitungsfunktionalität. Da es nur einen einzigen Ausgangspunkt für die Operationen im System gibt, steht keine Duplikation des Verwaltungscodes zu befürchten.

rationsschnittstellen der passenden erweiterbaren Systeme. Dieses Problem wird im Renew-Plugin-System auf zwei verschiedene Weisen gelöst. Zum einen wird auf das *Singleton*-Pattern zurückgegriffen, so dass aus dem Java-Code sämtlicher statisch abhängiger Plugins heraus direkt mit der *einen* Instanz einer Erweiterungsverwaltung kommuniziert werden kann. Neben der zentralen Plugin-Verwaltung selber nutzen beispielsweise die neu eingeführten Menü- und Dateitypen-Verwaltungen unter JHotDraw dieses Muster. Ein Nachteil dieser Technik ist, dass – ohne eine strenge Kontrolle über die Sichtbarkeit von Klassen und Packages – Konfigurationsschnittstellen eines Plugins an beliebiger Stelle im Plugin angeboten werden können und somit keine überblicksgebende zentrale Plugin-Fassade existiert. Im JHotDraw-Plugin wurde diese Technik vor allem aus dem Grund verwendet, dass der zugrunde liegende JHotDraw-Code genau diese Fassade ohnehin nicht bietet.

Die andere in Renew verwendete Möglichkeit zum Auffinden von Konfigurationsschnittstellen führt daher über die Plugin-Repräsentanten. In jedem Plugin existiert ein Objekt, welches als Repräsentant des Plugins dient, von der Plugin-Verwaltung instantiiert, über Lebenszykluszustandsänderungen informiert und mit der Konfiguration zur Integration und Entkopplung des Plugins beauftragt wird. In den meisten erweiterbaren Plugins basiert dieses Objekt auf einer spezialisierten Klasse, welche als *Fassade* [siehe GHJV98, S. 185ff.] für alle Konfigurationsschnittstellen des Plugins dient. Eine sich integrierende Erweiterung kann daher über den Plugin-Manager die Referenz auf den Repräsentanten erhalten und über dessen Konfigurationsschnittstellen die dynamische Integration erreichen.

Im Vergleich zum Singleton-Pattern, welches in einem unveränderlichen System die naheliegende Lösung zum Ansprechen der Repräsentanten wäre, erfordert der Weg über die Plugin-Verwaltung einen höheren Aufwand im Programmcode der Erweiterung. So muss die Indirektion aufgelöst und der Rückgabewert der generischen Abfragemethoden des Plugin-Managers in den richtigen Typ umgewandelt werden, zudem ist je nach Konfiguration mit unterschiedlichen Suchergebnissen zu rechnen. Um „faulen“ Programmierern diesen Aufwand zu ersparen, verbergen einige Plugins diesen Aufwand in einer Methode, welche eine dem Singleton-Pattern ähnliche Semantik hat. Die Plugin-Repräsentantenklasse bietet dann eine statisch referenzierbare Methode `getCurrent()` an, welche die dynamische Anfrage nach der Referenz bei der Plugin-Verwaltung kapselt. Obwohl diese Kombination dem Plugin-Entwickler zunächst wie eine Erleichterung vorkommt, gefährdet sie doch die Entkopplung der Plugins. Von der Nutzung dieser Singleton-artigen Methoden zum Erhalten einer Referenz ist daher abzuraten. Sinnvoller ist die Verlagerung dieser Methoden vom referenzierten zu allen anfragenden Plugins, obwohl dies eine Code-Duplikation darstellt. Diese Duplikation ist der dynamischen Konfigurierbarkeit geschuldet und daher nicht vermeidbar.

Ein neu erstelltes und in Form einer Hilfsklasse unterstütztes Entwurfsmuster ist die so genannte weiche Abhängigkeit (`SoftDependency`). Eine weiche Abhängigkeit besteht, falls ein Plugin ein anderes erweitern können soll, *ohne* von diesem statisch abhängig zu sein. Ein Beispiel ist die Agentenplattform Capa, deren Funktionalität im Prinzip vollständig im nicht-grafischen Bereich der Petrinetz-Simulation verortet ist. Um jedoch in der grafischen Darstellung des Renew-Editors Agenten als Marken besonders darzustellen, erweitert Capa auch das Petrinetz-spezifische Editor-Plugin. Diese Funktionalität hat im Vergleich zum eigentlichen Zweck des Capa-Plugins nur

einen marginalen Stellenwert und rechtfertigt keine statische Abhängigkeit von der grafischen Oberfläche. Die weiche Abhängigkeit erlaubt es, diese zusätzliche Funktionalität nur dann ins System zu integrieren, wenn die grafische Oberfläche vorhanden ist. Es handelt sich dabei um eine Alternative zum in Unterabschnitt 2.4.4 besprochenen Entwurfsmuster des *Bridge-Plugins*, da die Agentenmarkendarstellung auch in einem eigenen Plugin, welches auf Capa und dem Editor aufbaut, hätte realisiert werden können.

Die hier präsentierten Beispiele für die Verwendung von Entwurfsmustern im Renew-Plugin-System sind keineswegs vollständig. Sie repräsentieren lediglich einige Muster, welche häufiger im System angetroffen werden können.

### 5.2.3 Laden von Klassen

Üblicherweise wird *vor* dem Start einer monolithischen Java-Anwendung die Variable `CLASSPATH` gesetzt, so dass die Anwendung während ihrer Laufzeit auf sämtlichen darüber auffindbaren Code zugreifen kann. Ein Java-Classloader sucht und lädt jede Klasse bei Bedarf und hält sie bis zum Ende der Laufzeit bereit. Wie bereits in Unterabschnitt 3.2.1.2 angesprochen, kann und muss dieses einfache Prinzip für eine dynamisch konfigurierbare Anwendung – wie ein Plugin-System – aus zwei Gründen verändert werden. Erstens können während der Laufzeit der Anwendung Plugins hinzugefügt werden, deren Code nicht im anfänglich konfigurierten `CLASSPATH` enthalten ist. Zweitens können während der Laufzeit Plugins wieder aus dem System entfernt werden, was zur Folge haben sollte, dass auch die dazugehörigen Klassen wieder "vergessen" werden.

Für den ersten Prototypen wurde ein einfacher Classloader entwickelt, welcher der ersten Anforderung gerecht wird, da er den ihm bekannten `CLASSPATH` während der Laufzeit des Plugin-Systems um weitere Orte mit Plugin-Code ergänzen kann. Die Java-Architektur verbietet es allerdings, dass ein Classloader einmal geladene Klassen wieder vergisst. Der Einfachheit halber wurde im ersten Prototyp auf die Umsetzung der zweiten Anforderung verzichtet.

Wie bereits in Unterabschnitt 5.1.3 angesprochen, bietet das monolithische Renew bereits einen „vergesslichen" Classloader zum Zwecke des Testens von Java-Referenznetzen mit vom Anwender bereitgestellten Anwendungsklassen. Die Lebensdauer von Klassen ist in diesem Fall an die Laufzeit einer Simulation innerhalb von Renew gekoppelt, so dass während einer Simulation weiterhin gilt, dass keine Klassen vergessen werden können. Diese Fähigkeit sollte nach Möglichkeit auch im Plugin-basierten Renew erhalten bleiben.

In Abbildung 5.7 ist der vorläufige Endstand der in mehreren Entwicklungsstufen entstandenen Classloader-Architektur dargestellt. Sie entstand unter der Prämisse, dem Anwender sämtliche CLASSPATH-Konfiguration abzunehmen, so dass er tatsächlich eine einfachst mögliche Konfigurationsänderung durch simples Installieren von Plugins durchführen kann.

Die Classloader-Hierarchie folgt dem in der Java-Laufzeitumgebung verankerten Prinzip der „Parent-first"-Delegation. Beim Suchen und Laden einer Klasse haben die Classloader Vorrang, welche weiter oben in der Abbildung angeordnet sind.

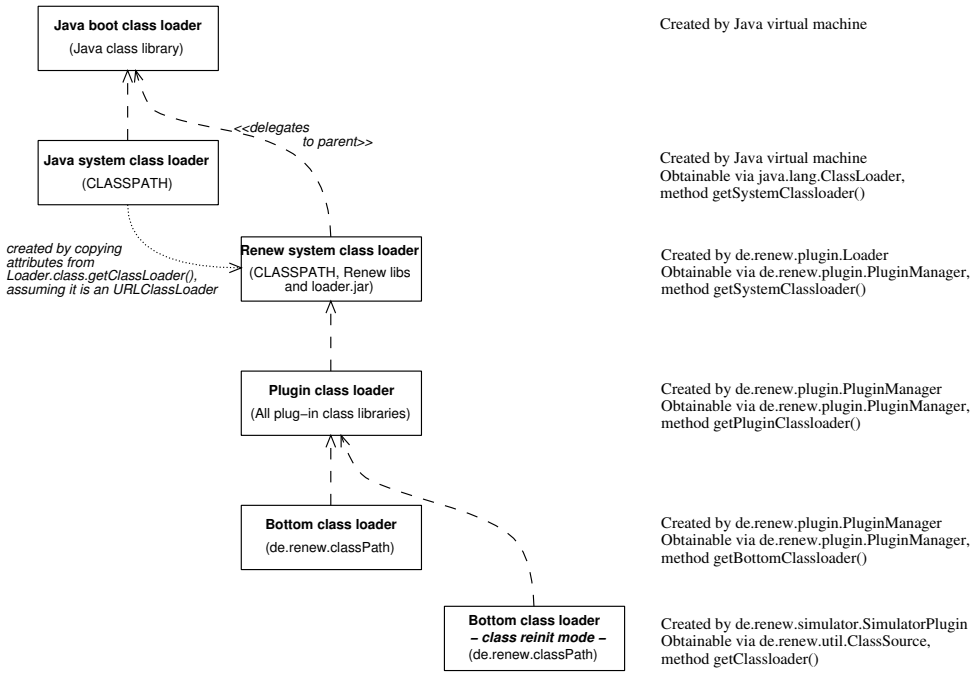



Abbildung 5.7: Die Delegationshierarchie der Classloader im Renew-Plugin-System

Der Java-System-Classloader wird durch einen alternativen RENEW-System-Classloader ersetzt, welcher einige zusätzliche Bibliotheken in den `CLASSPATH` mit aufnimmt, auf welche die Plugin-Verwaltung angewiesen ist. Dieser Classloader lädt auch die Klassen der Plugin-Verwaltung selber. Sämtlicher Plugin-Code wird von einem speziellen Classloader geladen, welcher während der Laufzeit von der Plugin-Verwaltung alle Orte mit Plugin-Code entgegennimmt.

Anwendungsklassen zur Verwendung in der Petrinetz-Simulation werden vom so genannten `BottomClassloader` geladen, der Code auf Basis einer beim Start von RENEW anzugebenden `CLASSPATH`-artigen Property namens `de.renew.classPath` findet. Die Möglichkeit, diese Klassen zu „vergessen“, wird durch ein Ersetzen des gesamten Classloaders beim Start einer Simulation implementiert, sofern der *Class-Reinit-Mode* aktiviert ist.

Für den praktischen Einsatz als IDE hat sich diese Classloader-Architektur als ausreichend herausgestellt, obwohl sie das Entladen von Plugin-Klassen nicht unterstützt. Das Entfernen von Plugins findet im Alltag der Software-Entwickler selten statt. Die damit verbundene Entkopplung der Plugin-Funktionalität ist auch ohne das Entladen der Klassen durchführbar und stellt den Anwender in vielen Fällen bereits zufrieden. Allerdings führt die auf das Entfernen folgende Integration einer modifizierten Va-

riante des Plugins zu unerwünschten Effekten, da noch die alten Klassen verwendet werden.

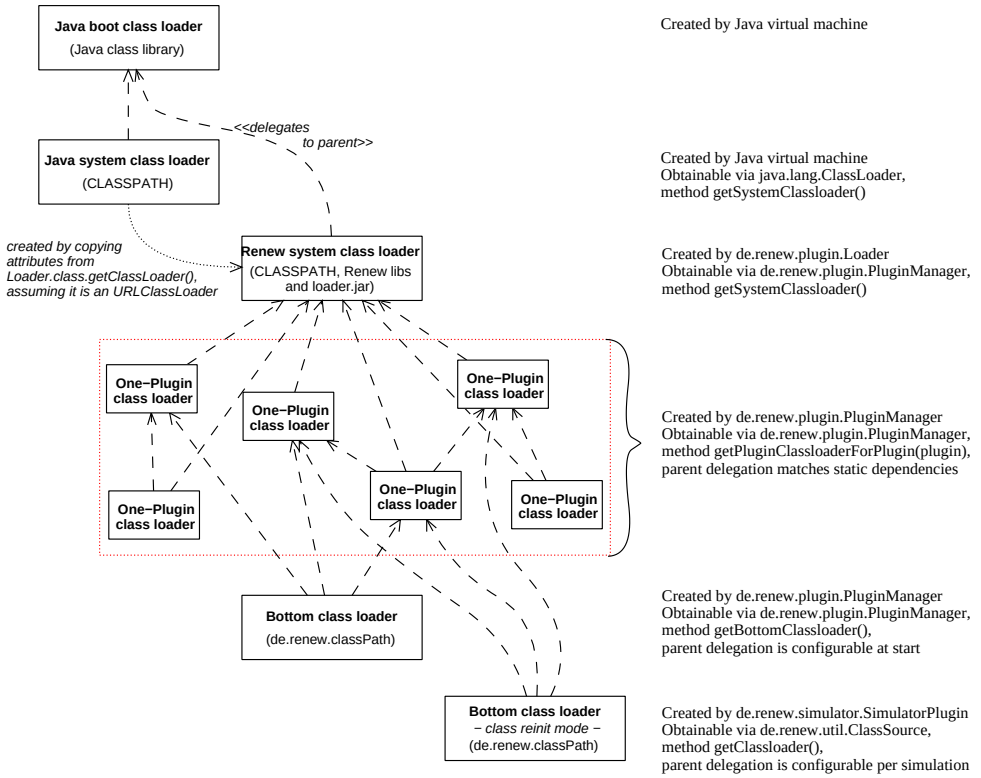



Abbildung 5.8: Mögliche Weiterentwicklung der Classloader im Renew-Plugin-System

Eine Weiterentwicklung der Classloader-Architektur könnte wie in Abbildung 5.8 dargestellt funktionieren. An die Stelle eines einzigen Classloaders für sämtliche Plugins (in der Abbildung durch den gepunkteten Kasten angedeutet) treten individuelle Classloader für jedes einzelne Plugin. Das „Parent-first“-Delegationsschema könnte so ausgebaut werden, dass jeder Plugin-Classloader über mehrere „parents“ verfügt, so dass die Delegationsketten entlang der in den Plugins definierten statischen Abhängigkeiten aufgefächert würden.

Wenn ein Plugin aus dem System entfernt wird, würde auch der zugehörige Classloader aus der Delegationshierarchie herausgenommen werden, so dass alle über ihn geladenen Klassen „vergessen“ sind. Diese Operation ist nur möglich, wenn keine weiteren Classloader unterhalb des zu entfernenden Classloaders eingeordnet sind. Diese Bedingung ergibt sich aus den definierten statischen Abhängigkeiten und stellt kein Problem dar, da vor dem Entfernen eines Plugins, von dem andere Plugins abhängig sind, ohnehin sämtliche Abhängigkeiten aufgelöst werden müssten.

Für die Bottom-Classloader ergäbe sich eine Besonderheit, da es möglich sein soll, aus Netzen in der Renew-Simulation heraus auch auf Plugin-Funktionalität zuzugreifen. Um nicht vom Bottom-Classloader aus eine nicht wieder auflösbare Abhängigkeit zu sämtlichen installierten Plugins aufbauen zu müssen, böte es sich an, den Anwender die „statischen“ Abhängigkeiten der Anwendungsklassen und -netze spezifizieren zu lassen. Die könnte auf eine ähnliche Weise geschehen, wie der `de.renew.classPath` die Nutzung von Nicht-Plugin-Klassen spezifiziert. Ein Entfernen von Plugins wäre dann möglich, solange der Bottom-Classloader nicht an den zugehörigen Plugin-Classloader delegiert. Alternativ könnte bei nicht laufender Simulation ein neuer Bottom-Classloader ohne die störende Abhängigkeit den alten ersetzen.

Die in Abbildung 5.8 dargestellte Classloader-Architektur ist im aktuellen Renew nicht implementiert, da sie durch den in Abschnitt 5.4 beschriebenen OSGi-basierten Prototypen überflüssig wird.

### 5.2.4 Laden von Netzen

Aus der Sicht eines Software-entwickelnden Renew-Anwenders handelt es sich bei Java-Referenznetzen gleichermaßen wie bei konventionell in Textform verfassten Java-Klassen um Code, der zur entwickelten Anwendung gehört. Es ist daher konsequent, für Renew-Netze während der Simulation (d.h. der Laufzeit der Anwendung) einen gleichberechtigten Lademechanismus wie für Java-Klassen vorzusehen. Die ersten Renew-Versionen boten jedoch nur die Möglichkeit, eine Simulation mit einem festen Satz an Netzen, dem „Schattennetzsystem“ (*shadow net system*), aufzusetzen. Bereits in Version 1.6 wurde ein Mechanismus zum Nachladen von Netzen bei Bedarf während der Simulation in Renew integriert, der aber weder Namensräume definiert noch eine Delegationshierarchie wie die Java-Classloader beherrscht.

Eine gleichberechtigte Integration von Java-Netzen als Klassen in die Java-Classloader-Architektur steht immer noch aus. Der globale Netzlademechanismus aus Version 1.6 wurde aber im Zuge der Einführung des Plugin-Systems mit einer Erweiterungsschnittstelle versehen, mittels derer Plugins zusätzliche Techniken zum Finden und Laden von Netzen in den Netzlademechanismus injizieren können. Der Petrinetz-spezifische Teil des grafischen Editors nutzt diese Schnittstelle, um auf Anforderung Netzzeichnungen, die erst nach Simulationsstart im Editor geöffnet wurden, nachträglich zu kompilieren und der Simulation zur Verfügung zu stellen. Diese Netzzeichnungen können sogar bei Anforderung automatisch im Dateisystem gesucht und im Editor geöffnet werden.

Plugins wie z. B. der Agentenplattform Capa, welche selber einen gewissen Petrinetz-Codeanteil beinhalten und diesen in jeder Renew-Simulation zur Verfügung stellen wollen, steht eine weitere Möglichkeit zur Integration ihrer Netze zur Verfügung. Als Erweiterung des Simulator-Plugins können sie auf das Ereignis des Simulationsstarts hin ihre Netze direkt in die Simulation injizieren.

### 5.2.5 Plugin-Manager

Die Verwaltung des Renew-Plugin-Systems ist im Java-Package `de.renew.plugin` implementiert, die zentrale Klasse des Packages ist der `PluginManager`. Er ist als

*Singleton* im System global bekannt und stellt die Konfigurationsschnittstelle zum Hinzufügen und Entfernen von Plugins während der Laufzeit bereit. Darüber hinaus implementiert oder steuert der `PluginManager` viele der in den vorangegangenen Unterabschnitten besprochenen Funktionen und greift zu diesem Zweck auf weitere Klassen des Packages zurück.

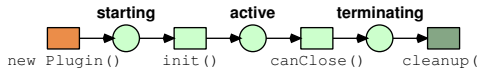



Abbildung 5.9: Lebensyklus von Plugins im Java-Prototypen

Der von der RENEW-Plugin-Verwaltung vorgegebene globale Lebenszyklus für Plugins umfasst je zwei Konfigurationsübergänge beim Integrieren und Entfernen von Plugins und entspricht damit ungefähr dem in Unterabschnitt 4.3.8.2 präsentierten zweischrittigen Modell. In Abbildung 5.9 ist der Lebenszyklus noch einmal als Petrinetz dargestellt. Im Lebenszyklus ist die Möglichkeit vorgesehen, dass ein Plugin Konfigurationsänderungen verweigert. So darf das Plugin im Konstruktor des Repräsentanten-Objekts den Lade- und Initialisierungsvorgang mit einer `PluginException` abbrechen. War die Instantiierung des Repräsentanten erfolgreich, muss der zweite Initialisierungschritt ebenfalls gelingen. Beim Entfernen von Plugins wird zunächst mit `canClose` die Zustimmung eingeholt, bevor das Plugin mit `cleanup` über das Ende seines Lebenszyklus informiert und damit zum Entkoppeln all seiner Erweiterungsbeziehungen aufgefordert wird.

In Jörn Schumachers Diplomarbeit [siehe Sch03, Abschnitt 7.2] lag ein besonderer Schwerpunkt auf dem Erkennen und Umsetzen einer außerhalb der Laufzeit der Anwendung möglicherweise veränderten Systemkonfiguration. Das Integrieren eines Plugins umfasst verwaltungsintern vier Schritte: das (1) Finden des Plugins, das (2) Überprüfen statischer Abhängigkeiten, das (3) Laden des Codes (gleichbedeutend mit dem Instantiieren des Repräsentanten-Objekts) und das (4) Initialisieren des Plugins. Die letzten beiden Schritte erfordern die Mitwirkung des Plugins, da sie Teil des Plugin-Lebenszyklus' sind.

Beim Start des Systems kommt den ersten beiden Schritten eine besondere Bedeutung zu, da alle installierten Plugins auf einmal integriert werden sollen, ohne ein Eingreifen seitens des Anwenders zu erfordern. Der `PluginManager` durchsucht eine konfigurierbare Menge an Verzeichnissen nach installierten Plugins. Das standardmäßig vorgesehene Plugin-Verzeichnis ermittelt der `PluginManager` automatisch relativ zur Lage seines eigenen Codes.

Jedes Plugin muss eine standardisierte Dateistruktur aufweisen, insbesondere muss eine Datei `plugin.cfg` vorhanden sein, welche unter anderem die statischen Abhängigkeiten zu anderen Plugins spezifiziert. Die Plugins werden entsprechend der durch die statischen Abhängigkeiten definierten partiellen Ordnung sortiert und der Reihe nach geladen und initialisiert. So findet jedes Plugin bei seiner Initialisierung sämtliche statischen Abhängigkeiten erfüllt vor. Plugins mit unerfüllbaren Abhängigkeiten werden weder geladen noch initialisiert.

Die Anforderung der Threadsicherheit wird im Prototypen auf primitive Weise erfüllt, indem sämtliche Operationen zur Konfiguration des Systems im wechselseitigen Ausschluss stehen. Die Kennzeichnung aller Konfigurationsoperationen durch das Java-Sprachkonstrukt `synchronized` stellt dies sicher, führt aber auch zu einer im Vergleich zum konzeptionellen Modell deutlich reduzierten Nebenläufigkeit in der Verwaltung. Das Implementieren einer nebenläufigen Plugin-Verwaltung wäre mit Java-Sprachkonstrukten sehr aufwendig geworden. Im Kontext von Renew steht mit der Petrinetzprogrammierung eine deutlich elegantere Lösung zur Verfügung, welche im nächsten Prototypen ausgenutzt wird.

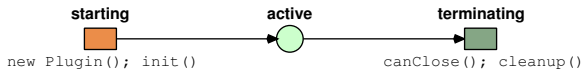



Abbildung 5.10: Lebenszyklus unter atomarer Sichtweise

Da die Plugin-Verwaltung die beiden jeweils aufeinanderfolgenden Konfigurationsschritte zum Hinzufügen bzw. zum Entfernen von Plugins unmittelbar hintereinander ausführt und keine nebenläufigen Konfigurationsoperationen möglich sind, lässt sich der Lebenszyklus von Plugins auch vereinfacht wie in Abbildung 5.10 dargestellt interpretieren. Diese Interpretation verdeutlicht, dass aus Sicht des Systems kein stabiler Zwischenzustand existiert. Würden anstelle von Methodenaufrufen bidirektionale Kanäle zur Implementierung der Zustandsübergänge eingesetzt werden, ließe sich auch dem Plugin gegenüber jeweils ein atomarer Konfigurationsschritt spezifizieren, welcher weiterhin auf das Einverständnis des Plugins Rücksicht nimmt.

Neben den essentiellen Aufgaben, die sich aus der angebotenen Konfigurationsschnittstelle heraus ergeben, implementiert der `PluginManager` weitere Funktionen, welche die Bedienung und Konfiguration des Systems für den Anwender erleichtern. Wie bereits in Unterabschnitt 5.1.5 angesprochen, benötigt das Plugin-System eine zentrale, von einer grafischen Oberfläche oder Textkonsole unabhängige Steuerungsmöglichkeit, um Konfigurationsänderungen durchzuführen. Zu diesem Zweck bietet der `PluginManager` eine erweiterbare Menge an Kommandos zum Diagnostizieren und Konfigurieren der Systemkomposition an. Plugins wie der grafische Editor oder der Simulator integrieren sich hier mit speziellen Kommandos zum Öffnen von Zeichnungen oder zur Steuerung der Simulation. Der `PluginManager` selber definiert weitere Kommandos zum Auslesen und Verändern der in Unterabschnitt 5.1.5 bereits angesprochenen `Properties` zur Konfiguration von Plugin-Parametern.

Beim Start des Systems führt der `PluginManager` *ein* übergebenes Kommando aus, so dass dieses Kommando die Benutzungsschnittstelle der Anwendung auswählen und initialisieren kann. Während der Laufzeit des Systems können weitere Kommandos zur Ausführung an den `PluginManager` übergeben werden, zwei alternative „Prompt“-Plugins präsentieren dem Anwender eine Interaktionsmöglichkeit zum Absetzen solcher Kommandos.

Das Renew-Plugin-System kann als Unterbau für verschiedene Anwendungen dienen, welche auch gleichzeitig installiert sein können. Eine bereits im monolithischen Renew vorhandene „Mehrköpfigkeit“ bestand mit dem Betrieb als grafische IDE und

dem reinen Simulationsbetrieb ohne grafische Oberfläche. Die Workflow-Engine bot wiederum eigene grafische Oberflächen zur Verwaltung der Agenda einer Person oder der Parameter des Workflow-Systems an. Die zuvor angesprochene Lösung, beim Start des Plugin-Systems ein Kommando zur Auswahl der zu startenden Benutzungsschnittstelle mitzugeben, geht bereits auf diesen Umstand zurück.

Das Beenden der Anwendung stellt im Plugin-System jedoch eine neue Herausforderung dar. In der monolithischen Version von Renew hatte grundsätzlich nur eine der Benutzungsschnittstellen die alleinige Kontrolle über die Lebensdauer der Anwendung. In häufigsten Fall war dies der grafische Editor, aber im grafiklosen Betrieb wurde die Laufzeit der Anwendung beispielsweise von der Simulator-Konsole bestimmt. Im generischen Plugin-System können verschiedene Plugins gleichzeitig Laufzeit-bestimmend erscheinen. Ein Anwendungsszenario ist beispielsweise die vorübergehende Nutzung der grafischen Oberfläche innerhalb einer eigentlich kommandozeilengesteuerten Simulation – in diesem Fall will der Anwender, wenn er den grafischen Editor schließt, nicht auch die laufende Simulation beenden. Ein anderer Anwendungsfall ist die Nutzung der grafischen Agenda des Workflow-Systems in Ergänzung zum grafischen Netzeditor.

Das Plugin-System sollte erst dann beendet werden, wenn auch die letzte grafische oder nicht-grafische Anwendung eines Plugins beendet ist. Eine Möglichkeit wäre, die Lebenszyklen der Plugins zu überlagern, so dass das Gesamtsystem beendet wird, sobald das letzte Plugin aus dem System entfernt wird. Diese Lösung würde aber vom Anwender erfordern, jedes Plugin explizit zu beenden. Alternativ könnte sich jedes Plugin am Ende seiner aktiven Nutzung selber deinstallieren – dies wäre aber eine semantisch nicht sinnvolle Kopplung von Installations- und Nutzungsdauer.

Der Renew-`PluginManager` implementiert stattdessen einen separaten Zustandsraum zur Steuerung der Laufzeit der Anwendung. Plugins können sich beim `PluginManager` als aktiv melden (Methode `blockExit()`) und auch wieder abmelden (Methode `exitOk()`). Diese Zustandsübergänge können nur für aktuell ins System integrierte Plugins angefordert werden, sind aber nicht direkt an die Übergänge des eigentlichen Plugin-Lebenszyklus' gekoppelt. Der Plugin-Manager beendet die Laufzeit des Systems automatisch, sobald kein Plugin als aktiv registriert ist. Beim Beenden wird für alle Plugins der Lebenszyklusübergang `canClose(); cleanup()` durchgeführt, so dass alle Plugins eine weitere Chance bekommen, ihre Entfernung aus dem laufenden System zu unterbinden.

Der Editor wechselt beispielsweise in den aktiven Zustand, solange das Hauptfenster sichtbar ist – so kann während der Laufzeit der Editor mehrfach geöffnet und geschlossen werden. Der Simulator registriert sich als aktiv, solange eine Simulation läuft, und stellt damit den nicht-grafischen Simulationsbetrieb sicher. Weitere Plugins, die sich als aktiv registrieren, sind die Workflow-Engine mit ihren grafischen Dialogen zur Konfiguration und zur Arbeits-Agenda sowie das Kommandozeilen-Prompt-Plugin, welches den Zustand auf Kommando des Anwenders hin wechselt.

Für den eher theoretisch interessierten Petrinetz-Modellierer reicht in der Regel der reine Editor-Betrieb mit grafischer Benutzungsoberfläche aus. Der grafische Editor kann daher so konfiguriert werden, dass er sein Fenster auch ohne explizit bei Systemstart anzugebendes Kommando öffnet. Dementsprechend fordert der Editor beim Schließen seines Fensters den `PluginManager` zum Stoppen des Gesamtsystems auf, ohne Rücksicht auf andere noch als aktiv registrierte Plugins (wie z. B. eine laufende

Simulation) zu nehmen. Somit lässt sich auch das Verhalten des Renew-Editors in Version 1.6 im Plugin-System nachbilden.

Der Renew-Plugin-Manager vereint damit mehrere Funktionen in sich, welche über die reine Verwaltung der Plugins und Konfiguration des Systems hinausgehen. Er erreicht damit das Ziel, für den normalen Anwender möglichst unsichtbar zu agieren, und dem fortgeschrittenen Anwender eine weitgehende Kontrolle der dynamischen Systemkonfiguration zu ermöglichen.

## 5.3 Zweiter Prototyp: Netze

Das prototypische netzbasierte Plugin-System geht direkt aus den in Abschnitt 4.3 präsentierten Referenznetzen hervor. Es stellt eine exemplarische Konkretisierung der Netze des konzeptionellen Modells im Kontext einer frei gewählten Anwendungsfunktionalität dar. Eine Variante des Prototypen ist an den im vorigen Abschnitt vorgestellten Java-Prototypen gekoppelt. Sie erlaubt die Integration von Plugin-Funktionalität in das dekomponierte Renew und wurde im Rahmen des Konferenzvortrages zum Papier [CDMR05] vorgeführt.

### 5.3.1 Zielsetzung

Die netzbasierte prototypische Implementierung ist im Anschluss an die Entwicklung der in Abschnitt 4.3 präsentierten konzeptionellen Modelle entstanden. Die Netze des konzeptionellen Modells verfügen als Java-Referenznetze über eine operationale Semantik, welche die Ausführung des modellierten Systems auf Basis des Renew-Simulators gestattet. Der Übersichtlichkeit halber wurden allerdings in den Abbildungen des Abschnitts 4.3 einige komplexe Anschriften an Transitionen und Kanten ausgelassen, die zur Funktionsfähigkeit des Systems notwendig sind. Um einen praktikablen Prototypen zu erhalten, ist darüber hinaus die als Platzhalter modellierte konkrete Funktionalität einer Anwendung zu ergänzen.

Der primäre Zweck des netzbasierten Prototypen ist also die Unterfütterung der während der Modellierung aufgestellten Behauptung, mit den Netzen ein Plugin-System praktisch implementieren zu können. Die Einfachheit der Konfiguration durch den Anwender kann dabei zunächst auf der in Unterabschnitt 4.2.6.1 eingeführten Abstraktionsebene bleiben. Eine Realisierung von Komfortfunktionen zum automatisierten Finden und Laden von Plugins, wie sie im Java-Prototypen vorgesehen sind, ist damit nicht notwendig. Stattdessen soll die Einfachheit der dynamischen Rekonfiguration dadurch erreicht sein, dass ein einzelner explizit vom Anwender angeforderter Schaltvorgang ausreicht, um die Konfigurationsänderung vollständig umzusetzen.

Der im vorigen Abschnitt präsentierte Java-Prototyp enthält bereits die Komfortfunktionen, welche im Netz-Prototyp ausgeblendet werden. Allerdings bietet die Verwaltung des Java-Prototypen eine geringere Nebenläufigkeit als die Netze des konzeptionellen Modells und damit auch des Netz-Prototypen. Es bietet sich daher an, eine Variante des Netz-Prototypen zu erstellen, welche an den Java-Prototypen gekoppelt ist und die Ablaufsteuerung unter Rückgriff auf den bereits in Java implementierten Plugin-Lader übernimmt. Plugins zu dieser Variante des Netz-Prototypen sind in der

Lage, die in Java implementierte Anwendung RENEW um weitere Funktionalität zu ergänzen. Ein exemplarisch im Kontext der Anwendung RENEW konkretisiertes Plugin soll diese Möglichkeit illustrieren, indem es auch die in Form von Java-Interfaces definierten Erweiterungsmöglichkeiten der dekomponierten Anwendung nutzt.

### 5.3.2 Netze

Als Ausgangsbasis für den netzbasierten Prototypen eines konkreten Plugin-Systems dient die in Abbildung 4.20 präsentierte Alternative unter den letzten Modellierungsschritten. Dabei handelt es sich um eine Plugin-Verwaltung, welche einen zweischrittigen Lebenszyklus definiert, eine atomare Konfiguration von Dienstbeziehungen unterstützt und die Steuerung der notwendigen Konfigurationsschritte zum Aufbau von Erweiterungsbeziehungen explizit den individuellen Plugins überlässt. Dieses Modell wurde gewählt, weil es relativ nah an den im vorigen Abschnitt vorgestellten Konfigurationsschnittstellen orientiert ist. Es stellt allerdings nicht die aus Sicht eines Plugin-Entwicklers aufwandsärmste Variante dar.

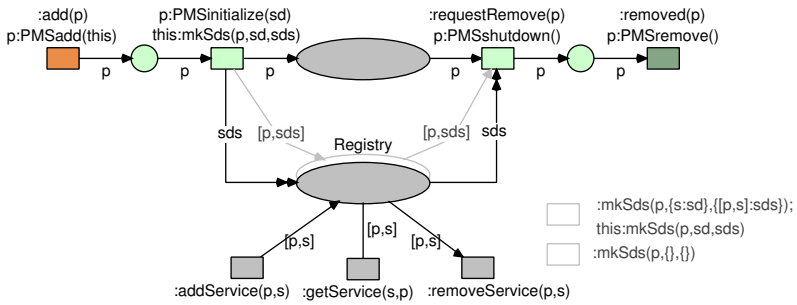



Abbildung 5.11: Das Plugin-Verwaltungssystem des Netz-Prototypen (`PMS.rnw`)

Abbildung 5.11 stellt das im Vergleich zur Ausgangsbasis nur geringfügig verfeinerte Referenznetz `PMS` dar, welches die Plugin-Verwaltung implementiert. Die Ergänzungen betreffen ausschließlich die Verwaltung der zentral konfigurierten Dienstbeziehungen. Zwei hohl gezeichnete Transitionen im unteren rechten Bereich des Netzes implementieren eine rekursive Kanalverschmelzung, welche aus einer Liste von Dienstbezeichnern eine Liste von Paaren macht, deren jeweils erste Komponente eine Referenz auf das dienstanbietende Plugin ist. Die Paare dieser Liste werden über die flexible Kante in die Dienst-Registry eingespeist. Hinter der Registry ist eine zweite Stelle zur Buchhaltung der über die flexiblen Kanten bewegten Markenmengen eingezeichnet. Diese ermöglicht beim Konfigurationsschritt `:requestRemove(p)` die Wiederverwendung der bei der Integration erstellten Dienstliste, ohne das Plugin erneut befragen zu müssen.

Als Beispiel für ein erweiterbares Plugin soll das bereits aus dem RENEW-Handbuch bekannte Nikolaus-Beispiel herangezogen werden [KWD06b, Fig. 3.15]. Es ist etwas einfacher gestaltet als die in Abschnitt 2.1 vorgestellten Versionen, reicht aber

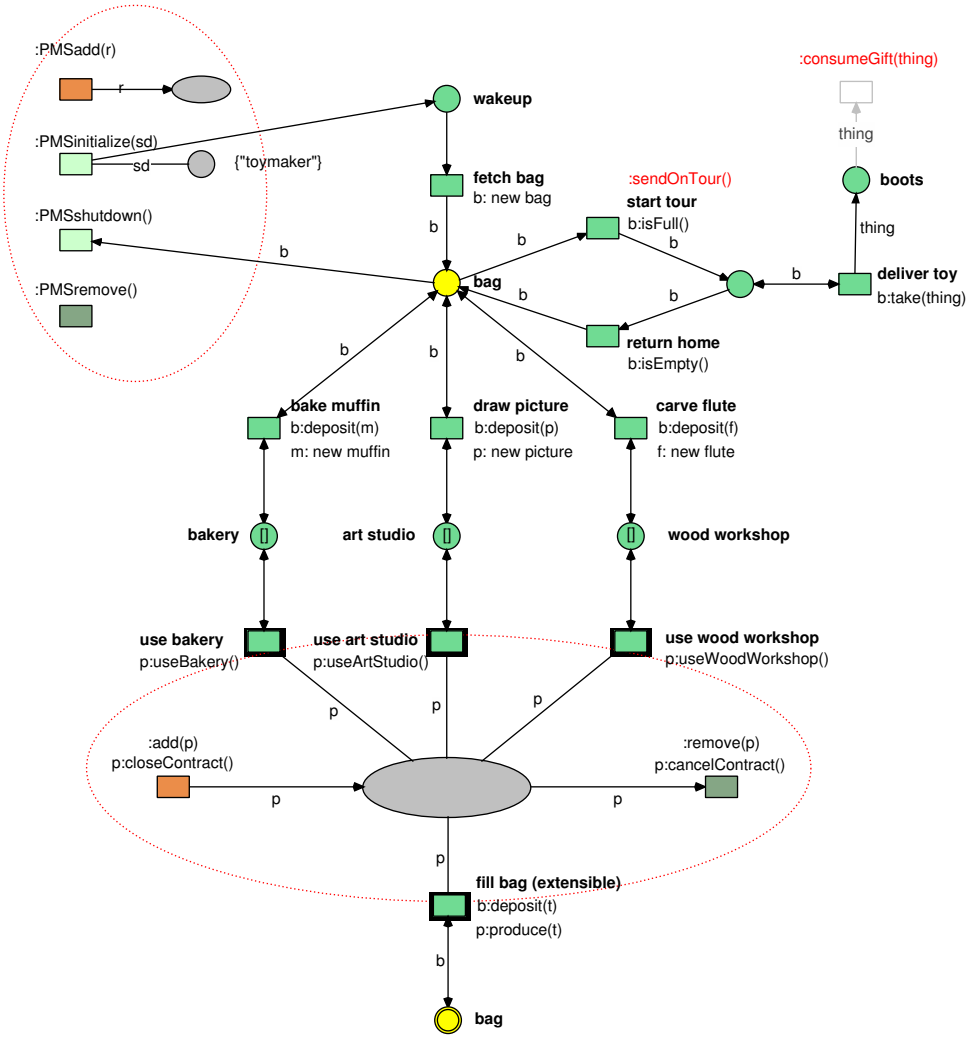



Abbildung 5.12: Ein erweiterbarer Nikolausbetrieb (`santa.rnw`)

zur Illustration der Möglichkeiten der Plugin-Konzepte aus. Um Erweiterbarkeit beschreiben zu können, wird der Produktionsprozess der Weihnachtgeschenke detaillierter ausgearbeitet. Das in Abbildung 5.12 dargestellte Netz unterscheidet im oberen rechten Bereich zwei Phasen der Nikolaus-Tätigkeit: die Herstellung von Spielzeugen und die Rundreise zum Verteilen derselben. Zwischen den Phasen wird mittels der Transitionen „start tour“ und „return home“ gewechselt, die jeweils nur aktiviert sind, wenn der Sack vollständig gefüllt bzw. geleert ist. Ein als abstrakter Datentyp mit den Operationen `:deposit(thing)`, `:take(thing)`, `:isFull()` und `:isEmpty()` implementierter Geschenksack dient der Lagerung und dem Transport der Geschenke. Er stellt eine einfache Erweiterung des Netzes „bag“ aus dem RENEW-Handbuch dar [KWD06b, Fig. 3.16].

Zur Herstellung von Geschenken verfügt der Nikolaus im mittleren Bereich des Netzes über eine Bäckerei, ein Atelier und eine Holzwerkstatt (Stellen „bakery“, „art studio“ und „wood workshop“), in welchen er drei Arten von Geschenken herstellen kann: Muffins, Bilder und Flöten. Die Werkstätten sind in diesem Beispiel lediglich als abstrakte Ressourcen modelliert, welche jeweils die Herstellung eines Geschenkes zur Zeit ermöglichen. Eine komplexere Modellierung der Werkstätten ist für die Zwecke dieses Beispiels nicht notwendig.

Die Herstellung von Geschenken ist in dem Sinne *intendiert erweiterbar*, als dass der Nikolaus Experten anheuern kann, die in seinen Werkstätten weitere Arten von Geschenken herstellen können. Die Erweiterungsspezifikation umfasst den im unteren Bereich des Netzes durch die gepunktete Ellipse gekennzeichneten Bereich. Zentrales Element dieses Bereiches ist die bereits aus dem konzeptionellen Modell bekannte Verwaltung von Erweiterungen. Der Lebenszyklus, der möglichen Erweiterungen auferlegt wird, umfasst beim Hinzufügen und Entfernen jeweils einen Konfigurationsschritt, welcher durch die synchronen Kanäle `p:closeContract()` und `p:cancelContract()` kommuniziert wird.

Drei fett berandete Transitionen erlauben den die Geschenkherstellung erweiternden Helfern mittels synchroner Kanäle die Nutzung der verschiedenen Werkstätten. Ganz unten stellt eine weitere fett berandet dargestellte Transition den Kanal `produce(t)` bereit, über welchen ein von einem Helfer fertiggestelltes Spielzeug `t` in den Sack des Nikolaus' gesteckt werden kann. Die Stelle „bag“, welche den Geschenksack aufnimmt, ist zur Vermeidung unübersichtlicher Kanten quer durch das Netz unten als virtuelle Stelle dupliziert dargestellt. Eine Beschränkung zulässiger Abläufe zwischen den verschiedenen Kanälen zur Herstellung und Ablieferung von Spielzeugen findet nicht statt, so dass beliebige – auch mehrschrittige – Produktionsprozesse möglich sind.

Da auch der Nikolausbetrieb selber als ein Plugin im PMS registriert werden soll, ist im oberen linken Bereich des Netzes der notwendige Lebenszyklus implementiert. Die vom PMS übergebene Referenz auf die Konfigurationsverwaltung wird im Nikolausbetrieb nicht benötigt, da er in diesem Beispiel keine Dienste anderer Systemkomponenten benötigt und sich nirgends als Erweiterung registriert. Über den Kanal `:PMSinitialize(sd)` wird bekanntgegeben, dass der Nikolausbetrieb den Dienst `"toymaker"` – im Sinne der Registrierung von Spielzeugexperten als Erweiterungen der Geschenkherstellung – anbietet. Die Entfernung des Nikolausbetriebs aus dem Plugin-System benötigt ebenfalls keine komplexen Konfigurationsschritte. Die Kante von der Stelle „bag“ zur Lebenszyklustransition mit dem Kanal `:PMSshutdown()` stellt aller-

dings sicher, dass der Betrieb nur aus dem System entfernt werden kann, solange der Nikolaus nicht auf Tour ist.

Die beiden rot gekennzeichneten Kanäle `:sendOnTour()` und `:consumeGift(thing)` simulieren die möglichen Interaktionen des Anwenders mit dem Nikolausbetrieb in diesem Beispiel. Auf Basis dieser Kanäle kann später eine einfache Steuerung des Netzsystems ermöglicht werden, welche die Rundreise anfordern und die Geschenke nach Auslieferung wieder aus den Stiefeln herausnehmen kann.

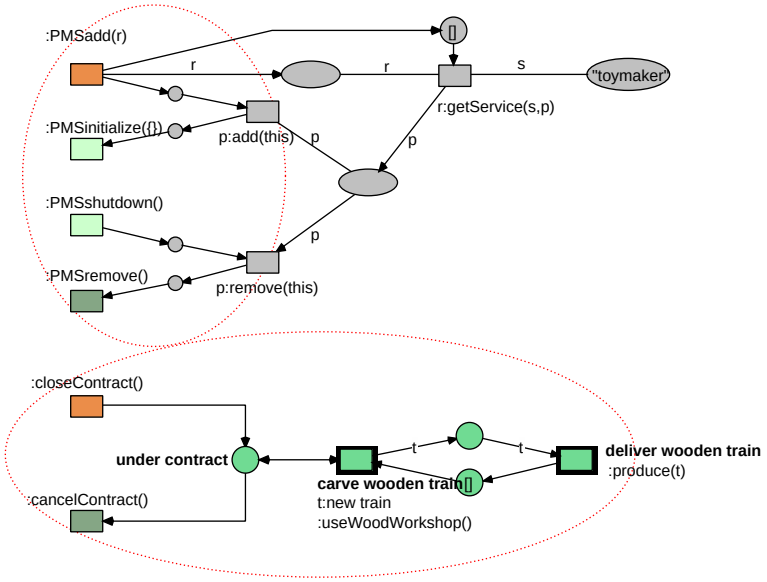



Abbildung 5.13: Ein Spielzeughersteller-Plugin (`trainMakerPlugin.rnw`)

In Abbildung 5.13 ist ein exemplarisches Plugin dargestellt, welches als Erweiterung des Nikolausbetriebs die Herstellung einer Holzeisenbahn ermöglicht. Zunächst implementiert das Netz genauso wie jedes andere Plugin die vier Kanäle des vom PMS vorgegebenen Lebenszyklus im oberen linken Bereich. In diesem Fall wird – wie bereits in Abbildung 4.20 konzeptionell dargestellt – die Spielzeugherstellerverwaltung des Nikolausbetriebs als Dienst angefordert, um das Plugin als Erweiterung zu konfigurieren.

Die eigentliche Funktionalität des Plugins ist an den Lebenszyklus der Erweiterungsspezifikation gekoppelt und im unteren Bereich des Netzes dargestellt. Nur solange, wie das Plugin als Erweiterung in den Nikolausbetrieb integriert ist, wird die Holzwerkstatt des Betriebs genutzt, um Holzeisenbahnen herzustellen. Die fertig gestellte Holzeisenbahn wird anschließend an den Betrieb übergeben. Die komplementäre Stelle stellt sicher, dass die Erweiterung nicht mehr Spielzeuge produziert als sie in den Geschenksack stecken kann.

Alternative Spielzeughersteller-Plugins können die Struktur des Netzes variieren. So kann beispielsweise auf mehrere Werkstätten nacheinander zur Herstellung eines

aufwendigeren Spielzeugs zurückgegriffen werden. Oder der Spielzeughersteller stellt ein kleines Lager bereit, um ein paar Exemplare seines Spielzeugs auf Vorrat fertigen zu können. Ein Spielzeughersteller kann auch einen eigenen Werkzeugkasten mitbringen, um nicht auf die Werkstätten des Nikolaus' angewiesen zu sein, oder um andere Materialien bearbeiten zu können.

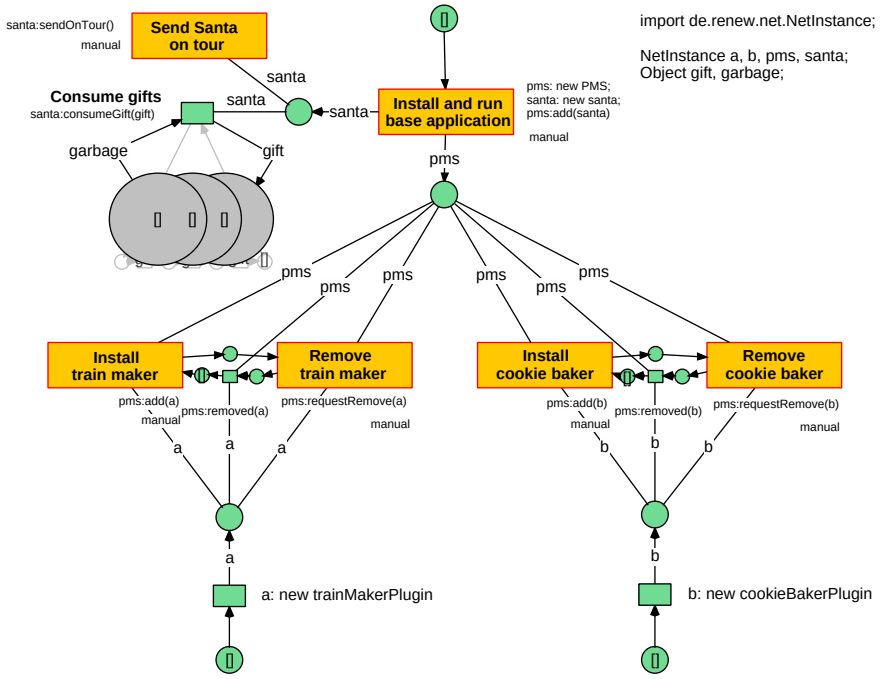



Abbildung 5.14: Die Interaktionsmöglichkeiten des Anwenders (`system.rnw`)

Die bisher vorgestellten Netze des Prototypen verfügen noch über keine Mensch-Maschine-Schnittstelle. Für einen einfachen, schnell zu implementierenden Prototypen bietet es sich an, die Implementierungssprache beizubehalten. Dies ist möglich, weil die grafische Ausgabe einer Renew-Simulation mit Hilfe von `manual`-Transitionen zur interaktiven Steuerung eines Netzsystems eingesetzt werden kann.

Abbildung 5.14 zeigt das Steuerungsnetz für den hier beschriebenen Prototypen eines Plugin-Systems. Die groß gezeichneten Transitionen erlauben dem Anwender die Steuerung des Systems per Mausklick, da sie als `manual` gekennzeichnet sind. Die Konfigurationsmöglichkeiten des Plugin-Systems nehmen dabei einen großen Raum ein, die eigentliche „Funktionalität" der Anwendung bietet dem Anwender nur eine einzige Interaktionsmöglichkeit, nämlich das Anstoßen der Rundreise des Nikolaus' (links oben im Bild). Das „Verbrauchen" der vom Nikolaus verteilten Geschenke geschieht im Steuerungsnetz automatisch, die drei großen grauen Stellen im oberen linken Bereich zeigen jeweils die letzten drei aus den Stiefeln gefischten Geschenke grafisch an.

Der Steuerungsblock zur Installation und Deinstallation eines konkreten Plugins ist für alle Plugins gleich aufgebaut. Unten wird beim Start der Simulation eine Instanz des Plugin-Netzes erzeugt, die als „das Plugin“ in Konfigurationsoperationen eingesetzt wird. Dem Anwender werden zwei Operationen, „Install“ und „Remove“, angeboten, welche auf die Konfigurationsschnittstellen `:add(p)` und `:requestRemove(p)` des zuvor beschriebenen PMS abgebildet werden. Wie bereits in Unterabschnitt 4.3.8.2 diskutiert, erfordert der zweiphasige Deinstallationsvorgang einen zweiten nach außen sichtbaren Kanal `:removed`, sofern die Umgebung des Systems an einer aktuellen Quittung des erfolgreichen Deinstallationsvorgangs interessiert ist. Dieser Kanal wird, da er keine Interaktion des Anwenders erfordert, in der kleinen Transition zwischen den beiden zur Auswahl stehenden Operationen aktiviert.

Wie das konzeptionelle Modell abstrahiert dieser in Referenznetzen implementierte Prototyp von der Installation von Plugins im Dateisystem. Stattdessen erzeugt das Steuerungsnetz von jeder Systemkomponente im Vorwege eine Instanz, die dynamische Rekonfiguration des Systems befasst sich dann nur noch mit der Kopplung und Entkopplung dieser durch Netzinstanzen repräsentierten Plugins. Das Steuerungsnetz beschränkt sich für dieses Beispiel auf zwei Plugins, welche zusätzlich zum immer installierten Nikolausbetrieb ins System integriert werden können. Diese Beschränkung ist jedoch keine konzeptionelle. Da das Steuerungsnetz die Interaktion des Anwenders mit dem System repräsentiert, kann es zur Aufnahme weiterer Plugins um weitere Steuerungselemente ergänzt werden. In Kombination mit Java-Anschriften und dem Renew-Netzlader wäre auch eine Installation beliebiger Plugins implementierbar, die vom Anwender nur die Angabe des zu instantiierenden Plugin-Netzes verlangt.

Abbildung 5.15 zeigt das in Ausführung befindliche System. Das Plugin-Verwaltungssystem (PMS) ist links direkt unter der Renew-Werkzeugleiste zu sehen, die Stelle für aktive Plugins enthält zwei Netzmarken: den Nikolausbetrieb `santa` (als Netzinstanz rechts unten im Bild) und das bereits integrierte `trainMakerPlugin` (als Netzinstanz links unter dem PMS sichtbar). In der Erweiterungsverwaltungsstelle des Nikolausbetriebs ist ebenfalls eine Referenz auf das integrierte `trainMakerPlugin` zu sehen. Links unten ist ein Ausschnitt des `cookieBakerPlugins` erkennbar, welches aber momentan nicht ins System integriert ist. Im Steuerungsnetz (rechts oben) sind Referenzen auf sämtliche Systemkomponenten vorrätig, um beim Anstoßen von Konfigurationsänderungen die betroffenen Plugins benennen zu können. Die letzten drei aus den Stiefeln gezogenen Geschenke sind ebenfalls zu sehen, darunter eine Holzeisenbahn, die zuvor vom Holzeisenbahn-Experten in der Holzwerkstatt des Nikolaus' produziert wurde.

### 5.3.3 Integration mit dem Java-Prototypen

Um das netzbasierte Plugin-System mit dem Java-Prototypen zu koppeln, wird zum einen auf die im konzeptionellen Modell prinzipiell vorhandene Möglichkeit der Schachtelung von Plugin-Verwaltungssystemen (PMS-en) zurückgegriffen. Jedes PMS kann auch als Plugin betrachtet werden und somit die Konfigurationsschnittstellen eines anderen PMS nutzen. Zum anderen wird eine kombinierte Implementierung aus Java- und Netz-Anteilen verwendet, welche sowohl die Kopplung an das Java-PMS als auch die schnelle Implementierung von Datenmanipulationsfunktionen in zustandslosen Hilfs-

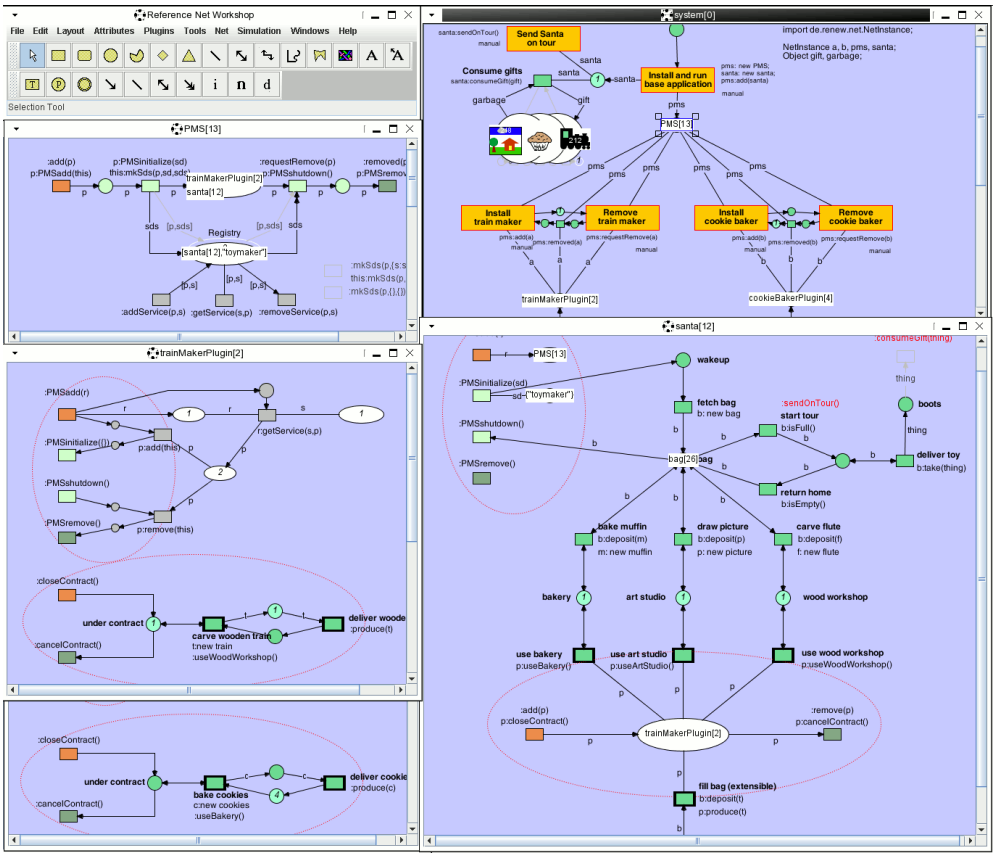



Abbildung 5.15: Das Plugin-System in Ausführung

klassen erlaubt. Letztere Technik ist vom Autor bereits in früheren Arbeiten [siehe Duv01, Duv02] verwendet und dokumentiert worden.

Zunächst startet Renew wie gewohnt auf Basis des in Java implementierten Plugin-Systems, wie es in Abschnitt 5.2 vorgestellt wurde. Im Rahmen des Renew-Simulators wird dann das netzbasierte PMS ausgeführt, welches zuerst die aktuelle Systemkonfiguration vom Java-PMS erfragt und die Markierung seiner Verwaltungsstellen entsprechend anpasst. Anschließend können die Konfigurationsschnittstellen des Netz-PMS genutzt werden, um weitere Plugins ins System zu integrieren oder wieder daraus zu entfernen.

In Abbildung 5.16 ist eine auf die Integration mit dem Java-System ausgelegte Variante des PMS-Netzes dargestellt.[3] Die blass dargestellte Transition in der oberen rechten Ecke des Netzes extrahiert bei der Initialisierung des PMS die Systemkonfigu-

[3] Die grafische Darstellung der Netze in diesem Unterabschnitt entspricht derjenigen aus dem Papier [CDMR05], da diese Netze zum Zweck der Demonstration im zugehörigen Vortrag gezeichnet wurden. Die Abweichungen zur Darstellung der im Rahmen dieser Arbeit präsentierten Netze beschränken sich im Wesentlichen auf die Farbgebung der Elemente sowie die Benennung einiger

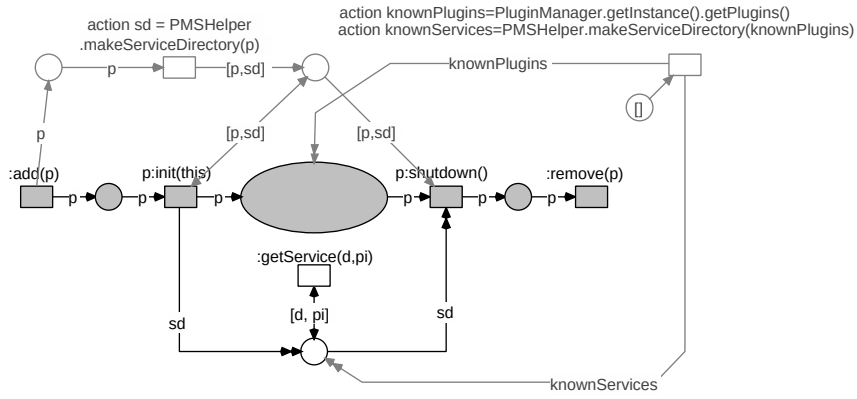



Abbildung 5.16: Integriertes Plugin-Verwaltungsnetz (`PMS.rnw`)

ration sowie das aktuell bestehende Angebot an Diensten aus dem Java-PMS. Die in Abbildung 5.11 als Schatten hinter das Dienstverzeichnis gelegte Buchführungsstelle mit den Dienstangeboten jedes Plugins ist in Abbildung 5.16 mittig oben eingezeichnet. Die Kanäle zum dynamischen An- und Abmelden von Diensten sind in diesem PMS nicht vorhanden, da der zugrunde liegende Java-Prototyp über diese Fähigkeit ebenfalls nicht verfügt. Der den Netz-Plugins auferlegte Lebenszyklus umfasst in diesem Prototypen lediglich zwei Zustandsübergänge, welche durch die Kanäle `:init(pms)` und `:shutdown()` kommuniziert werden.

Zusätzlich schreibt das Netz-PMS vor, die Methode `getProperties()` des im Java-Prototypen definierten Interfaces `IPlugin` zu implementieren, welche das Auslesen und Ändern der Konfigurationsparameter des Plugins erlaubt. Diese dienen – wie bereits in Unterabschnitt 5.2.5 erläutert – zum einen der Parametrisierung der Plugin-Funktionalität, zum anderen der Ablage von Verwaltungsinformationen durch das Java-PMS. Diese werden in den blass dargestellten Netzelementen im linken oberen Bereich des PMS-Netzes ausgewertet, um das Dienstangebot eines hinzuzufügenden Plugins zu ermitteln. Die eigentliche Auswertung der Properties, um das Dienstangebot zu ermitteln, geschieht in einer zustandslosen Hilfsmethode, welche in Java implementiert ist. Damit ersetzt das Java-Interface `IPlugin` den im vorigen Netz-Prototypen vorgesehenen Kanal `:PMSinitialize(sd)` zur Übergabe der Dienstbeschreibung.

In den Abbildungen 5.17 und 5.18 ist ein konkretes Plugin dargestellt, welches durch das Netz-PMS verwaltet werden kann. Es ist als Netz-Stub implementiert [siehe KWD06b, Abschnitt 3.9], so dass die Netzinstanz zugleich ein Objekt vom Typ `IPlugin` ist und die im Interface `IPlugin` geforderten Methoden in Form synchroner Kanäle implementiert. Wie am Netz in Abbildung 5.17 zu sehen, werden nicht alle Methoden des Interfaces korrekt implementiert, da die im Stub vorgesehenen Kanal-Uplinks im Netz fehlen. Dies ist darauf zurückzuführen, dass zur Verwaltung

Kanäle. Unterschiede, die sich aus der Kopplung mit dem Java-PMS ergeben, werden im Text erläutert.

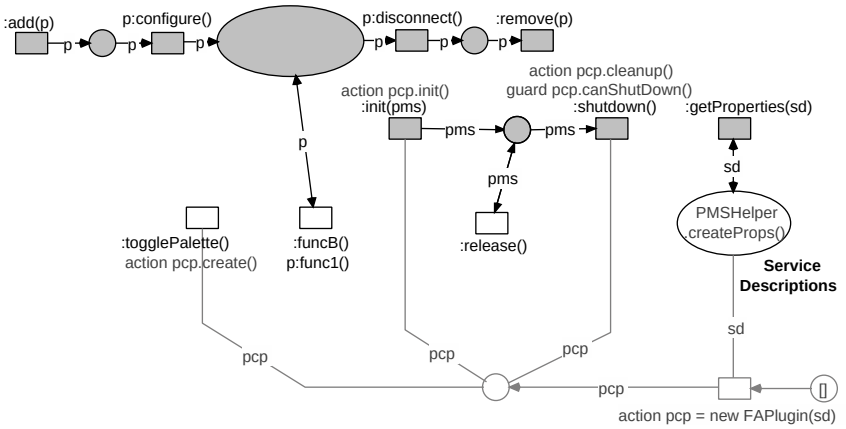



Abbildung 5.17: Beispiel-Plugin zur Erweiterung der Renew-GUI (`FAPlugin.rnw`)

```
package samples.plugin;
class FAPlugin for net FAPlugin
implements
  de.renew.plugin.IPlugin
{
  java.lang.String getName() {
    this:getName(return);
  }
  de.renew.plugin.PluginProperties getProperties() {
    this:getProperties(return);
  }
  void init() {
    this:init();
  }
  boolean cleanup() {
    this:cleanup(return);
  }
  java.lang.String getAlias() {
    this:getAlias(return);
  }
  boolean canShutDown() {
    this:canShutDown(return);
  }
}
```

Abbildung 5.18: Stub-Definition zum Plugin aus Abbildung 5.17 (`FAPlugin.stub`)

durch das Netz-PMS keine Methoden der Lebenszyklusverwaltung des Java-PMS außer `getProperties()` benötigt werden.

Die vom FAPlugin bereitgestellte Funktionalität umfasst eine Werkzeugleiste zum Zeichnen von endlichen Automaten, welche in den RENEW-Editor integriert werden kann. Die eigentliche Werkzeugpalette ist in Form von Java-Code in einer Klasse namens `FAPlugin` implementiert, deren Instanz im Netz in der unteren Stelle verwahrt wird. Die eigentliche Funktionalität ist daher nicht direkt im Netz sichtbar. Sobald das Plugin in RENEW integriert ist, wird die zusätzliche Werkzeugleiste im RENEW-Hauptfenster eingeblendet. Das Plugin bietet allerdings über das PMS den „Dienst“ an, diese Werkzeugleiste ein- oder auszublenden (Kanal `:togglePalette()`). Die zur Veröffentlichung dieses Dienstes notwendige Dienstbeschreibung wird in einer zustandslosen Java-Hilfsmethode erzeugt und als Properties-Objekt in die Stelle am rechten Rand des Netzes gelegt.

Der obere Bereich des Netzes ist unmittelbar aus dem Modell des Grundkonzepts eines erweiterbaren Systems übernommen und entspricht der Darstellung aus [CDMR05]. Weder die Erweiterungsverwaltung noch die weiß dargestellte Transition mit Kanalanschrift nach dem Schema `:funcX()` wird vom FAPlugin benötigt, da dieses Plugin in seiner derzeitigen Version keine Erweiterungsmöglichkeiten bietet. Die grauen Elemente in der zweiten Zeile implementieren den vom Netz-PMS vorgegebenen Lebenszyklus. Der Anstoß zum Entfernen des Plugins aus dem System wird in dieser Lebenszyklusvariante vom Plugin aus gegeben, da der in Unterabschnitt 4.3.8.2 besprochene äußere Kanal `:requestRemove(p)` im PMS fehlt. Stattdessen bietet das Plugin selber einen Kanal `:release()`, welcher vom Anwender aufgerufen werden kann, um den Konfigurationsvorgang anzustoßen.

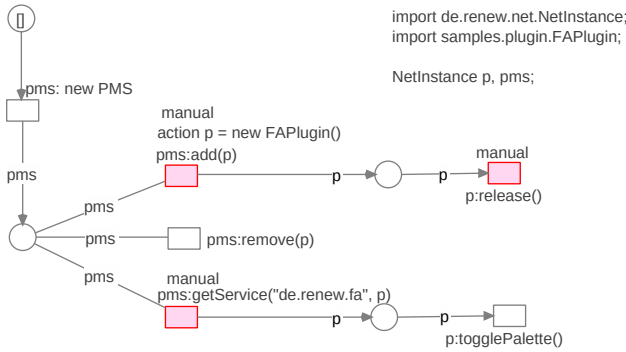



Abbildung 5.19: Steuerungs-/Interaktionsnetz zum integrierten Prototypen (`UI.rnw`)

Das Steuerungsnetz, welches beim Start der RENEW-Simulation instantiiert wird und dem Anwender wie in der vorigen Variante des Netz-Prototypen eine „Ein-Klick-Oberfläche“ zum Integrieren und Entfernen des Plugins bietet, ist in Abbildung 5.19 dargestellt. Die rot gefärbten Transitionen erlauben dem Anwender das Hinzufügen und Entfernen einer frisch instantiierten Instanz des Plugins sowie die Nutzung des vom Plugin angebotenen Dienstes, welcher die Werkzeugpalette ein- und ausblenden

kann. Der Dienst wird hierbei über die Verwaltungsschnittstelle des Netz-PMS erfragt und in einem zweiten automatisierten Schritt genutzt. Der `:remove(p)`-Kanal im Steuerungsnetz schaltet nach Bedarf, sobald das Plugin seine Entfernung wie zuvor erläutert angestoßen hat.

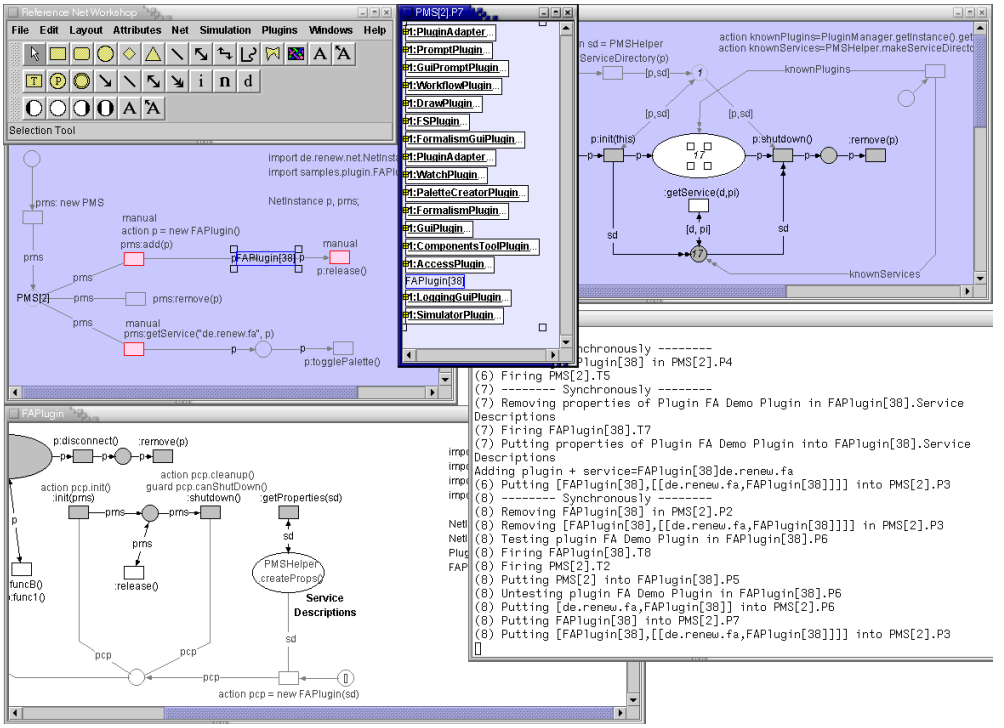



Abbildung 5.20: Der kombinierte Prototyp in Ausführung mit integriertem FAPlugin

In Abbildung 5.20 ist das System in Ausführung zu sehen. Das FAPlugin ist bereits ins System integriert und zwischen die anderen Plugins in der zentralen Stelle des Netz-PMS einsortiert. Die zusätzliche Werkzeugpalette ist im Renew-Hauptfenster sichtbar.

## 5.4 Dritter Prototyp: Java-OSGi

Der OSGi-basierte Prototyp der Anwendung Renew entstand im Rahmen der Diplomarbeit von Felix Simmendinger [Sim07]. Im Gegensatz zu den beiden vorigen Prototypen entfällt in dieser Variante die eigenständige Entwicklung einer Plugin-Verwaltung, da stattdessen auf die vorhandene Infrastruktur des OSGi-Rahmenwerkes aufgesetzt wird. Die in Unterabschnitt 5.1.4 beschriebene Dekomposition wird fortgeführt, um die Softwarearchitektur klarer zu strukturieren und an die in OSGi vorherrschende Beziehungsart der Dienstnutzung anzupassen. Die auf dem OSGi-Prototypen basierende Version von Renew kann wiederum zur Steuerung der Konfiguration des OSGi-

Laufzeitsystems eingesetzt werden [siehe SCDK07]. Das OSGi-Rahmenwerk könnte in einem zukünftigen Renew-Release den in Abschnitt 5.2 präsentierten ersten Java-basierten Prototypen eines Plugin-Systems ablösen.

### 5.4.1 Zielsetzung

Der in Abschnitt 5.2 vorgestellte Prototyp einer in Java implementierten Plugin-Verwaltung weist einige Unzulänglichkeiten im Bezug auf das dynamische Laden und Konfigurieren eines modularen Systems auf, welche im OSGi-Rahmenwerk detailliert und ausgereift gelöst sind. Es bietet sich daher an, anstelle einer eigenen PMS-Implementierung mit einem aufwendig zu perfektionierenden dynamischen Modulladememechanismus auf die bestehenden Lösungen des OSGi-Rahmenwerkes zurückzugreifen.

Ein weiterer Auslöser für die Entwicklung eines OSGi-basierten Prototypen war die Aufgabenstellung der Diplomarbeit von Felix Simmendinger [Sim07], in welcher eine Steuerung des Ablaufs von Simulationsexperimenten mittels Petrinetzen realisiert werden sollte. Die zu steuernden Werkzeuge zum Experimentieren mit Simulationsmodellen waren bereits auf der Eclipse-Plattform aufsetzend implementiert. Eine Integration der Renew-Funktionalität in die Eclipse-Plattform war somit eine Voraussetzung für die Lösung der Aufgabenstellung.

Zudem handelt es sich beim Renew-Plugin-System um einen proprietären Prototypen, der außerhalb der Anwendung Renew unbekannt ist. Gleiches gilt zwar in abgeschwächter Form auch für das Plugin-System der Eclipse-Plattform, nicht jedoch für die dem System zugrunde liegende OSGi-Laufzeitumgebung. Das OSGi-Rahmenwerk (siehe Unterabschnitt 2.2.3 und Abschnitt 3.2) ist seit vielen Jahren in Form mehrerer Implementierungen unabhängiger Hersteller im Einsatz und nicht zuletzt seit der Umstellung der Eclipse-Plattform auf das Rahmenwerk in Java-Entwicklerkreisen weithin bekannt. Ein Wechsel des Rahmenwerkes könnte die einzelnen Renew-Komponenten einem größeren Entwickler- und Anwenderkreis zugänglich machen.

Die Verfügbarkeit mehrerer unabhängiger OSGi-Implementierungen ist ein Grund, das OSGi-Rahmenwerk dem Eclipse-Rahmenwerk vorzuziehen. Obwohl die Eclipse-Plattform den Vorteil böte, sowohl eine ausgereifte Infrastruktur für Erweiterungsbeziehungen als auch ein umfangreiches Rahmenwerk zur Implementierung von Integrierten Entwicklungsumgebungen mitzubringen, wäre eine darauf aufbauende Anwendung nicht ohne die Eclipse-Plattform lauffähig, welche „nur“ von einer einzigen (wenn auch relativ großen und zudem offenen) Entwicklergruppe hergestellt wird. Um die Verwendbarkeit des portierten Renew-Prototypen unter mehreren OSGi-Implementierungen nachzuweisen, kam neben der in Eclipse enthaltenen OSGi-Implementierung „Equinox“ auch eine alternative Implementierung namens „Knopflerfish“ [Kno] als Ausführungsumgebung zum Einsatz.

Dennoch bietet der Wechsel zum OSGi-Rahmenwerk, welches auch als Basis der Eclipse-Plattform dient, die Chance der Integration der Renew-Funktionalität in die Eclipse-IDE. Eine nahtlose Integration der Java-Referenznetze als Programmiersprache im Kontext der Entwicklungsumgebung Eclipse würde die gemischte Verwendung mit der in Eclipse sehr gut unterstützten Programmiersprache Java in Softwareentwicklungsprojekten erleichtern. Dieses Ziel wäre auch mit dem Renew-eigenen PMS erreichbar, wie die Ergebnisse der Diplomarbeit von Klaus Mitreiter [Mit08] zeigen (siehe Unterabschnitt 5.4.5). Die nahtlose Integration der Renew-Funktionalität in

die Eclipse-IDE erfordert zwar eine Neu-Implementierung der Renew-Benutzungsoberfläche im Kontext des Eclipse-Rahmenwerkes, eine Umstellung auf das OSGi-Komponentenmodell erleichtert jedoch die Integration der Simulationsfunktionalität ungemein.

Allerdings ist das OSGi-Rahmenwerk nicht auf intendiert erweiterbare Systeme spezialisiert, sondern versteht sich als dynamisches Modulsystem. Die Infrastruktur zur Kopplung der Module orientiert sich an der Dienstbeziehung (siehe Unterabschnitt 4.3.8.1). Daher müssen entweder die Fassaden der bereits dekomponierten Renew-Plugins auf die Dienstbeziehung hin umgestaltet oder eine Unterstützung der Erweiterungsbeziehung im Kontext des OSGi-Rahmenwerks implementiert werden. Dieser Thematik kommt aufgrund des essentiellen Charakters der intendierten Erweiterbarkeit für Plugin-Systeme eine besondere Bedeutung zu.

Der zweite Schwerpunkt bei der Entwicklung des OSGi-Prototypen betrifft das Laden von Netzen. Wie bereits in Unterabschnitt 5.2.4 festgestellt, verfügt der Renew-Simulator über keinen der Java-Classloader-Architektur gleichwertigen Netzlademechanismus, da dieser ausschließlich global konfigurierbar ist. Die Umstellung der grundlegenden Softwarearchitektur auf einen dynamisches Modulsystem wie das OSGi-Rahmenwerk erfordert eine Anpassung des Netzlademechanismus, sofern Netze als gleichwertige Elemente von Modulen wie Klassen oder andere Ressourcen verwendet werden sollen.

### 5.4.2 Erweiterungsbeziehungen in OSGi

Im konzeptionellen Modell wird die Erweiterungsbeziehung in Form einer in der intendiert erweiterbaren Systemkomponente explizit verwalteten Menge von Erweiterungen realisiert, welche zu einer von der erweiterbaren Systemkomponente vorgegebenen, bidirektionalen Schnittstellenspezifikation konform gehen (siehe Unterabschnitt 4.3.2). Im Java-Prototpyen ist dieses Muster direkt übernommen worden, ohne es durch ein Rahmenwerk zu unterstützen. Eine erweiterbare Systemkomponente kann unter Rückgriff auf diverse Entwurfsmuster Erweiterungen verwalten und mit diesen über mehrere unidirektionale Java-Interfaces, welche von der erweiterbaren Systemkomponente entlang der gewählten Entwurfsmuster definiert sind, kommunizieren (siehe Unterabschnitt 5.2.2). Das Zusammenspiel mehrerer unidirektionaler Java-Interfaces ermöglicht eine bidirektionale Kommunikation wie im konzeptionellen Modell. Die Bidirektionalität ist wichtig, da ein erweiterbares System seinen Erweiterungen in der Regel einen umfangreichen Kontext an nutzbarer Funktionalität anbietet.

Im Zuge der Umstellung der Renew-Plugins auf OSGi bieten sich mehrere Wege zur Abbildung der Erweiterungsbeziehungen an:

**Renew-Erweiterungsbeziehung** Das OSGi-Rahmenwerk kann als reines Modulladesystem verwendet und die in ihm zur Verfügung gestellte Infrastruktur zur Kopplung von Komponenten mittels der Dienstbeziehung ignoriert werden. Stattdessen können die in Java implementierten Renew-Erweiterungsbeziehungen auf der Basis von Entwurfsmustern beibehalten werden. Es wären lediglich die zuvor im Renew-PMS konfigurierten statischen Abhängigkeiten in die passende OSGi-Syntax zu übertragen.

**Erweiterungsverwaltung als Dienst** Im konzeptionellen Modell und im Netz-Prototypen besteht die Möglichkeit, die Erweiterungsverwaltung jeder erweiterbaren Systemkomponente als Dienst zu veröffentlichen (vgl. Unterabschnitt 4.3.8.4 und Unterabschnitt 5.3.2). Die Veröffentlichung dieses Dienstes kann natürlicherweise über das OSGi-Rahmenwerk geschehen. Die eigentliche Erweiterungsbeziehung nebst zugehöriger Interfaces und individueller Verwaltung bleibt wie in der vorigen Variante gegenüber dem Java-Prototypen unverändert.

**Eclipse-Erweiterungsbeziehung** Wie in Unterabschnitt 3.2.1.5 erwähnt, bietet Eclipse ein ausgefeiltes Rahmenwerk zur Deklaration von Erweiterungsbeziehungen zwischen Plugins. Dieses Rahmenwerk könnte auch zur Ausgestaltung der Erweiterungsbeziehungen zwischen RENEW-Plugins herangezogen werden. Es könnte die im Java-Prototypen notwendigen Erweiterungsverwaltungen der einzelnen Plugins ersetzen. Die bereits spezifizierten Interfaces könnten in die Eclipse-Erweiterungspunktdeklarationen eingearbeitet werden.

**Erweiterung als Dienstangebot** Wie bereits in Unterabschnitt 4.1.3.1 angemerkt, kann jede Erweiterungsbeziehung als gegenseitige Dienstbeziehung verstanden werden. Jede Erweiterung bietet einen auf die Erweiterungsspezifikation passenden Dienst mit nur einem einzigen im Vorwege bekannten Nutzer an, der die volle Hoheit über die Nutzung der einzelnen Dienstangebote hat. Umgekehrt kann die vom erweiterbaren System als Kontext angebotene Funktionalität ebenfalls als Dienstnutzung aufgefasst werden. Da hier aber eher eine *statische* anstatt einer *funktionalen*[4] Abhängigkeit vorliegt, kann die Nutzung von Kontext-Funktionalität auch ohne ein explizit instantiiertes Dienstangebot erfolgen. Das erweiterbare System könnte also die Verwaltung installierter Erweiterungen einem `ServiceTracker` [siehe OSG06b, Kapitel 701, S. 335ff.] überlassen und jeweils bei Bedarf einen der Anbieter (d.h. eine Erweiterung) aufrufen.

**Erweiterbarkeit als optionale funktionale Abhängigkeit** Auf dem vorigen Vorschlag aufbauend kann auch anstelle der in Java zu implementierenden Dienstanbieterverwaltung auf die deklarative Dienstbeziehungsverwaltung der *OSGi-Service-Component-Runtime* [SCR, siehe Unterabschnitt 3.2.1.6 und OSG06b, Kapitel 112, S. 281ff.] gesetzt werden. Diese Variante hätte den Vorteil, dass erweiternde Plugins nur aktiviert werden, falls ihr Dienstangebot tatsächlich benötigt wird. Allerdings ist darauf zu achten, in den SCR-Deklarationen keine unbedingt zu erfüllenden funktionalen Abhängigkeiten [*mandatory cardinality*, siehe OSG06b, Abschnitt 112.3.2, S. 290f.] zu spezifizieren. Notwendige funktionale Abhängigkeiten bestehen bei einer Erweiterungsbeziehung ja gerade *nicht*, da das intendiert erweiterbare System auch ohne passende Erweiterung verwendbar sein soll. Zweitens sollte eine erweiterbare Systemkomponente dynamisch auf Konfigurationsänderungen reagieren können [*dynamic policy*, siehe OSG06b, Abschnitt 112.3.3, S. 291f.] Bei einer entsprechenden Deklaration können nicht

[4] Das Adjektiv „*funktional*“ wird in weiten Teilen dieser Arbeit nicht im mathematischen Sinne verwendet. Eine „*funktionale Abhängigkeit*“ bedeutet eine Abhängigkeit vom Vorhandensein bestimmter Funktionalität im System (siehe auch Unterabschnitt 3.4.2).

nur die beziehungsdefinierenden Interfaces, sondern auch die bereits in den Renew-Plugins existierenden Methoden, welche auf Konfigurationsänderungen reagieren, weiterverwendet werden. Die Steuerung des Lebenszyklus der Erweiterungsbeziehung wird vollständig von der OSGi-Laufzeitumgebung übernommen.

Die zusammen mit Felix Simmendinger im Kontext seiner Diplomarbeit [Sim07, Kapitel 6] erstellte prototypische OSGi-basierte Version von Renew nutzt mehrere der zuvor genannten Möglichkeiten mit Ausnahme des Eclipse-Rahmenwerkes für Erweiterungsbeziehungen. Wie bereits im vorigen Unterabschnitt erläutert, soll die Funktionalität von Renew nicht an die Eclipse-Plattform gebunden werden, sondern unter beliebigen OSGi-Implementierungen zur Verfügung stehen.

Um schnell zu einer lauffähigen Version des Gesamtsystems zu kommen, wurden zunächst keine Änderungen an den Schnittstellen vorgenommen, sondern lediglich die notwendigen Deklarationen statischer Abhängigkeiten erzeugt, um die Renew-Plugins auf der OSGi-Plattform ausführen zu können. Die Kopplung der Plugins geschieht weiterhin über das PMS des Java-Prototypen, welches lediglich die Classloader-Funktionalität nicht mehr bereitstellen muss. Bereits in dieser Phase stellten sich wesentliche Unterschiede zum Modulladem echanismus des Renew-PMS heraus, welche in Unterabschnitt 5.4.4 besprochen werden.

Für den in Simmendingers Diplomarbeit relevanten Einsatzzweck musste zunächst der nicht-grafische Simulationskern nebst Formalismen und Netzlader überarbeitet werden. Hier kamen sowohl der `ServiceTracker` als auch die *Service Components* zum Einsatz, wobei sowohl Art der Beziehung als auch die Rollen der betroffenen Plugins im Vergleich zum Beziehungskonzept des Java-Prototypen angepasst wurden.

Die Umgestaltung der Formalismen und des Netzladers wird im folgenden Unterabschnitt besprochen werden. Viele Erweiterungsbeziehungen der grafischen Oberfläche sind bisher noch nicht an das neue Beziehungskonzept angepasst. Hier gilt weiterhin, dass das Renew-PMS zur Konfiguration der Beziehungen benötigt wird.

### 5.4.3 Simulations-Bundles

Anlässlich der Umstellung auf das OSGi-Rahmenwerk als zugrunde liegendes Modulladesystem der Renew-Softwarearchitektur stellten sich einige Fragen der Dekomposition in einzelne Plugins bzw. Bundles erneut, die schon in Unterabschnitt 5.1.4 angesprochen wurden. Die im ersten Prototypen bereits angedachte, aber nicht vollendete Abtrennung individueller Formalismen von der zentralen Formalismenverwaltung wurde wieder aktuell, da insbesondere der Java-Referenznetzformalismus im Java-Prototypen auf den `BottomClassloader` angewiesen ist, um Typen und Objekte korrekt referenzieren oder instantiieren zu können. Aufgrund der in Unterabschnitt 5.4.4 angesprochenen Unterschiede der Klassenlader-Hierarchie muss der `BottomClassloader` im OSGi-Prototypen ersetzt werden.

Ein zweiter Grund für die Neugestaltung ist die bisher in Renew verwendete Technik zum Festlegen des Petrinetzformalismus eines Schattennetzes. Der Formalismus wird im Schattennetzsystem in Form eines Java-Factory-Objektes referenziert, welches auf Anfrage den zugehörigen Compiler instantiiert. Diese Technik führt dazu, dass der Simulator, welcher letztendlich das Übersetzen der Schattennetze anstößt,

die Klassen des speziellen Formalismus-Plugins nutzt. Die statische Abhängigkeit im Java-Prototypen besteht aber in Gegenrichtung, da jeder Formalismus aus spezialisierten Klassen des Simulator-Plugins besteht. Diese Verwendung von Klassen in Gegenrichtung der statischen Abhängigkeit ist prinzipiell möglich, da der Simulator lediglich das Interface `ShadowCompiler` deklariert und nutzt, so dass er die konkret vorhandenen Formalismen nicht kennen muss. Allerdings stellt im OSGi-Rahmenwerk das Einlesen der Schattennetze aus Dateien im Rahmen des Netzladers oder beim grafiklosen Betrieb ein Problem dar (siehe Unterabschnitt 5.4.4).

Da das OSGi-Rahmenwerk über ein ausgefeiltes Classloader-Konzept verfügt, bietet es sich an, auf dieses zum Ersatz des Bottom-Classloaders, des Netzladers und zum Referenzieren der Formalismen zurückzugreifen. Ein so genanntes Simulations-Bundle soll sämtliche Informationen zusammenfassen, welche zum Aufsetzen und Betreiben einer konkreten Simulation inklusive aller beteiligten RENEW-Plugins, Formalismen, Netze und Anwendungsklassen benötigt werden.

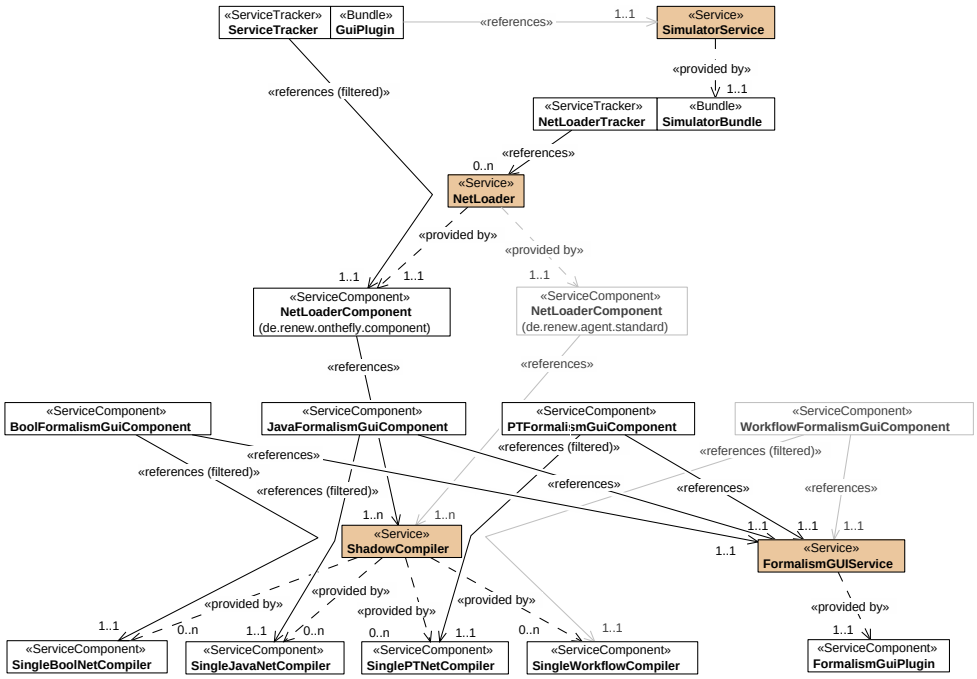



Abbildung 5.21: Dienstbeziehungen im OSGi-Prototypen

Im Ergebnis der Umgestaltung im Kontext von Simmendingers Diplomarbeit [Sim07, Kapitel 6] wurden einige Dienstbeziehungen zwischen Systemkomponenten eingerichtet, um die statischen Abhängigkeiten und Erweiterungsbeziehungen durch ein vom OSGi-Rahmenwerk unterstütztes Konzept zu ersetzen. In Abbildung 5.21 sind die aktuell im Prototypen vorhandenen OSGi-basierten Dienstbeziehungen dargestellt. Das Diagramm umfasst nicht nur die grafiklosen Teile der Simulation, sondern auch die

Schnittstellen zum petrinetzspezifischen Teil des Renew-Editors. Die blassgrau markierten Elemente stellen konsequente Weiterverwendungen derselben Beziehungsmuster dar, welche aber im Prototypen aktuell noch nicht umgesetzt sind.

Die Dienstbeziehungen sind – bis auf wenige experimentelle Ausnahmen – in Form von *Service Components* entsprechend der *Declarative Services Specification* der OSGi [OSG06b, Kapitel 112] umgesetzt. Dies bedeutet, dass jedes dienstanbietende oder -nutzende Bundle eine Klasse als *Service Component* ausweist, deren Lebenszyklus direkt vom Rahmenwerk gesteuert wird. Die gestrichelten Kanten weisen von einer Dienstspezifikation zu jenen *Service Components*, welche den Dienst anbieten. Eine Kardinalität von `0..n` steht für einen als *Service Factory* deklarierten Dienstanbieter, der die Erzeugung beliebig vieler Instanzen nach Bedarf zulässt, eine Kardinalität von `1..1` hingegen bezeichnet einen gewöhnlichen Dienstanbieter mit nur einer Instanz (vgl. Lebenszyklen in Abbildung 3.12). Die durchgezogenen Kanten weisen von einer dienstnutzenden *Service Component* zur jeweiligen Dienstspezifikation oder – falls der Dienstnutzer auf einen bestimmten Dienstanbieter hin filtert – mit einem Knick in der Nähe der genutzten Dienstspezifikation direkt zum passenden Dienstanbieter. Kardinalitäten an diesen Kanten beschreiben die Notwendigkeit und Vielfältigkeit der Beziehung: Während `1..1` für die Notwendigkeit der Bindung an exakt einen Dienstanbieter steht, deutet eine Kardinalität von `0..n` oder `1..n` auf mögliche Bindungen an mehrere verschiedene Dienstanbieter hin (ohne und mit notwendiger Existenz mindestens eines Anbieters). Die Kardinalitätsangaben entsprechen damit der OSGi-Spezifikation [vgl. OSG06a, Abschnitt 112.3.2, S. 290f.].

Zentrales Element des veränderten Beziehungsgeflechts sind die „Simulations-Bundles“. In Abbildung 5.21 sind zwei der für solche Bundles typischen *Service Components* vom Typ `NetLoaderComponent` dargestellt. Bei dem linken Simulations-Bundle (Package `de.renew.onthefly`) handelt es sich um jenes Simulations-Bundle, welches vom grafischen Editor beim Simulationsstart erzeugt, konfiguriert und installiert wird. Das rechte Simulations-Bundle steht exemplarisch für netzbasierte Plugins wie z. B. die Mulan/Capa-Agentenplattform. Beide Varianten von Simulations-Bundles verhalten sich weitgehend gleich und können daher gemeinsam besprochen werden.

Ein Simulations-Bundle umfasst sämtliche Elemente und Parameter, die zur Ausführung eines konkreten Petrinetz-Systems bzw. eines netzbasierten Plugins notwendig sind. Dazu gehören neben den Netzen, welche den Quellcode der netzbasierten Anwendung ausmachen, auch sämtliche anwendungsspezifischen Java-Klassen. Damit ersetzt das Simulations-Bundle sowohl den per `de.renew.classPath` konfigurierten `BottomClassloader` als auch den per `de.renew.netPath` konfigurierten `Netloader` der bisherigen Renew-Versionen.

Ein wesentliches Element zur Konfiguration eines Simulations-Bundles sind gerade die von der OSGi-Laufzeitumgebung aufzulösenden statischen und funktionalen Abhängigkeiten. Diese können und sollen in der Form deklariert werden, dass sämtliche zum Betrieb der *konkreten* Simulation notwendigen Systemkomponenten vom Simulations-Bundle aus referenziert werden. Dazu gehören neben möglichen weiteren Simulations-Bundles mit Netzen und Klassen insbesondere die zum Übersetzen der Netze nötigen Formalismen. Das Referenzieren von Netzen und Klassen aus weiteren Simulations-Bundles unterscheidet sich aufgrund der unterschiedlichen Lademecha-

nismen. Für Klassen reicht eine statische Abhängigkeit aus, da die OSGi-Laufzeitumgebung diese in eine Classloader-Delegation abbildet. Der Netloader-Mechanismus muss jedoch explizit implementiert werden, so dass hier auf funktionale Abhängigkeiten zurückgegriffen wird. Da jedes Simulations-Bundle eine von der SCR verwaltete `NetLoaderComponent` beinhaltet, ist allerdings auch die Deklaration dieser funktionalen Abhängigkeit kaum aufwendiger als die einer statischen Abhängigkeit.

In Abbildung 5.21 ist die funktionale Abhängigkeit zum Formalismus (bezeichnet als `ShadowCompiler`) eingezeichnet, welche *immer* notwendig ist. Bundles mit Netzen in mehreren verschiedenen Formalismen können entsprechend mehrere funktionale Abhängigkeiten deklarieren. Die Compiler der Formalismen werden von der SCR über den *Service Factory*-Mechanismus für jedes Simulations-Bundle erneut instantiiert. Am unteren Rand des Bildes sind einige der in Renew vorhandenen Formalismen aufgeführt.

Damit ist die Erweiterungsbeziehung zwischen Simulator und Formalismus einer funktionalen Abhängigkeit zwischen Simulations-Bundle und Formalismus gewichen. Diese Änderung ist möglich, da die Aufgabe des Kompilierens von Netzen vom Simulator in das Simulations-Bundle verlagert wird. Da der Simulator nicht mehr für das Übersetzen von Netzen zuständig ist, handelt es sich auch in dieser Hinsicht nicht mehr um ein erweiterbares System.

Aus Gründen der Übersichtlichkeit befindet sich in Abbildung 5.21 zwischen den `NetLoaderComponent`s und dem Dienst `ShadowCompiler` eine Reihe von `FormalismGuiComponent`s. Diese Elemente liegen entsprechend der Schichtung der Softwarearchitektur viel weiter oben, haben aber viele direkte Verbindungen zu den Formalismen, welche sonst das gesamte Diagramm hätten durchqueren müssen. Bei den `FormalismGuiComponent`s handelt es sich um die bereits im Java-Prototypen vorgesehenen Erweiterungen des grafischen Editors um Werkzeuge und Zeichnungselemente zu den individuellen Formalismen. Jede `FormalismGuiComponent` nutzt das Dienstangebots nur *eines bestimmten* Compilers. Der passende Compiler wird mittels einer Deklaration konfiguriert, welche die von der OSGi-SCR verwalteten Dienstanbieter auf den einen passenden Dienstanbieter-Typ filtert.

Die Editor-Plugins zu den Formalismen erweitern wie im reinen Java-Prototypen das `FormalismGuiPlugin`. Die Verwaltung der Erweiterungsbeziehung wird hier entsprechend der im vorigen Unterabschnitt als zweite Variante vorgeschlagenen Technik als Dienstleistung des erweiterbaren Bundles veröffentlicht. Die Interfaces der Erweiterungsspezifikation bleiben unverändert.

Im oberen Bereich der Abbildung 5.21 sind zwei weitere Nutzer des von den Simulations-Bundles angebotenen Netzlade-Dienstes dargestellt, das `SimulatorBundle` und der petrinetzspezifische Teil des Editors namens `GuiPlugin`.

Die Beziehung zwischen Simulator und Simulations-Bundle entspricht den letzteren der in Unterabschnitt 5.4.2 vorgestellten Varianten einer Erweiterungsbeziehungsimplementierung innerhalb des OSGi-Rahmenwerkes. Der Simulator definiert die Schnittstellen des `Netloader`-Dienstes und beobachtet mittels eines `ServiceTracker`s die Verfügbarkeit von Netzladern im System. Jeder Netzlader darf sämtliche vom Simulator zur Kodierung von Netzen bereitgestellten Klassen nutzen, da immer eine statische Abhängigkeit vom Netzlader zum Simulator besteht. Der Simulator wird als

erweiterbares System betrachtet, dessen Verhalten durch Netze beeinflusst wird, die über Netzlader bereitgestellt werden.

Bei der Beziehung zwischen `GuiPlugin` und Simulations-Bundle handelt es sich um eine feste 1:1-Beziehung, welche vom GUI-Plugin gesteuert wird. Bereits im Java-Prototypen wurde der petrinetzspezifische Teil des Editors als Erweiterung des Simulators betrachtet, da ersterer Start- und End-Ereignisse der Simulations-Engine anreichert und Netze in die Simulation injiziert. Das Injizieren von Netzen in den Simulator geschieht im OSGi-basierten Prototypen mittels eines speziellen Simulations-Bundles, welches als *On-the-fly-Bundle* bezeichnet wird. Dieses Bundle wird vom Editor beim Aufsetzen einer Simulation mit den im Editor geöffneten Netzen und einem vom Anwender konfigurierbaren Classpath erstellt und sofort ins laufende OSGi-System integriert (*on the fly*). Die „statischen“ und funktionalen Abhängigkeiten dieses Bundles sollen sich ebenfalls im Editor konfigurieren lassen. Beim Terminieren der Simulation wird das *On-the-fly-Bundle* wieder aus der Systemkonfiguration entfernt.

Die Erweiterungsschnittstelle des Simulators, welche ein Anreichern von Zustandsübergängen wie Simulationsbeginn und -ende erlaubt, ist noch nicht in die Umgestaltung einbezogen worden, da außer dem `GuiPlugin` keine Nutzer dieser Schnittstelle mehr existieren (die netzanbietenden Plugins, welche vormals auf den Simulationsstart reagiert haben, verfügen nun über eine geeignetere Technik). Die zentralen Methoden der Simulator-Fassade zum Aufsetzen, Starten und Stoppen einer Simulation werden nun als eigenständiger Dienst über das OSGi-Rahmenwerk angeboten, so dass sowohl die grafiklose Steuerungskomponente als auch der grafische Editor auf diesen Dienst zugreifen können.

Durch das *On-the-fly-Bundle* kann ein Petrinetz-Entwickler die Anwendung Renew weiterhin als IDE für beliebig komplexe Anwendungen nutzen. Während der Entwickler in früheren Renew-Versionen die für seine Anwendung geeignete Konfiguration in verschiedenen Properties und einer passenden Auswahl an Plugins zu konfigurieren hatte, kann die Konfiguration nun einheitlich über Mechanismen des OSGi-Rahmenwerkes erfolgen. Dieses Vorgehen hat vor allem den Vorteil, dass Konfigurationsfehler wie z. B. nicht-erfüllbare deklarierte Abhängigkeiten von der OSGi-Laufzeitumgebung beim Versuch der Integration des *On-the-fly-Bundles* erkannt und gemeldet werden.

### 5.4.4 Unterschiede zum Renew-PMS

Die Bundle-Verwaltung des OSGi-Rahmenwerkes [*Module layer*, siehe OSG06a, Kapitel 3] ist deutlich stärker auf dynamische Rekonfigurierbarkeit des Systems ausgelegt als die Plugin-Verwaltung des in Abschnitt 5.2 vorgestellten Java-Prototypen, welcher einige vereinfachende Annahmen zugrunde liegen. Während der Entwicklung des OSGi-basierten Renew-Prototypen traten mehrere daraus resultierende Unterschiede zu Tage, welche im Folgenden kurz umrissen werden sollen.

**Aktivierung** Das in Unterabschnitt 5.2.5 beschriebene PMS des Java-Prototypen nutzt die in den Plugins deklarierten statischen Abhängigkeiten nicht nur zum Validieren der Systemkonfiguration, so dass Plugins mit unerfüllten Abhängigkeiten nicht geladen werden, sondern auch zur Bestimmung der Aktivierungsreihenfolge. Dies ist

möglich und nötig, weil jedes korrekt installierte Renew-Plugin beim Start des Gesamtsystems automatisch geladen und initialisiert und damit aktiviert wird. Der vom Renew-PMS vorgegebene Lebenszyklus unterscheidet entsprechend den in der konzeptionellen Modellierung vorgenommenen Abstraktionen nicht zwischen ungeladenen und aktivierten Zuständen.

Dagegen unterscheidet die Bundle-Verwaltung der OSGi-Plattform im Bundle-Lebenszyklus [siehe Abbildung 3.10 und OSG06a, S. 80] unter anderem die Zustände „resolved“ und „active“. Sämtliche installierten Plugins nehmen, sobald ihre statischen Abhängigkeiten aufgelöst wurden, zunächst den Zustand „resolved“ ein. Die Aktivierung der Bundles kann entweder individuell programmiert werden oder durch die Festlegung von Startleveln des Gesamtsystems erfolgen. Eine automatische Berücksichtigung der statischen Abhängigkeiten zur Bestimmung der Aktivierungsreihenfolge ist nicht vorgesehen.

Für Renew-Plugins bieten sich hier zwei Lösungsmöglichkeiten an: Entweder kann die Teilfunktionalität des Renew-PMS, welche die Aktivierung vornimmt, ins OSGi-System übernommen werden. Oder es werden zwischen den Plugins auf Basis der *Declarative Services* zusätzlich funktionale Abhängigkeiten parallel zu den bestehenden statischen Abhängigkeiten definiert. Funktionale Abhängigkeiten können die Aktivierungsreihenfolge bestimmen, da die OSGi-SCR in der Lage ist, Komponenten erst nach Gewährleistung funktionaler Abhängigkeiten zu aktivieren.

Derzeit ist noch keine der beiden Lösungen implementiert, stattdessen muss manuell auf die korrekte Einhaltung der Startreihenfolge geachtet werden. Eine teilweise Automatisierungsmöglichkeit besteht in der Festlegung mehrerer Startlevel und der Zuordnung der Renew-Plugins entsprechend ihrer Abhängigkeiten zu diesen Startleveln. Die festgelegten Startlevel müssen allerdings bei der Installation weiterer Plugins manuell überarbeitet werden.

**Java-Reflection und -Serialisierung** Die im OSGi-Rahmenwerk vorhandene komplexe Vernetzung von Classloadern, welche das Entladen von Bundles ermöglicht und einer der wichtigsten Gründe für die Abkehr vom ersten PMS-Prototypen ist, brachte während der Entwicklung des OSGi-basierten Prototypen einen zuvor unbeachteten Seiteneffekt mit sich.

Während des Kompilierens von Netzen und auch während der Renew-Simulation wird häufig auf den Java-Reflection-Mechanismus zurückgegriffen, um Transitions- oder Kantenanschriften auszuführen, welche mit Java-Objekten arbeiten. Um auch vom Renew-Anwender bereitgestellte Klassen zu berücksichtigen, wird der in Unterabschnitt 5.2.3 beschriebene `BottomClassloader` eingesetzt, welcher einen eigenen Classpath kennt und mittels Delegation auf den `PluginClassloader` die Klassen sämtlicher Plugins zur Verfügung hat. Beim Entwurf des OSGi-Prototypen war klar, dass für den `BottomClassloader` ein dem OSGi-Rahmenwerk angemessener Ersatz geschaffen werden muss. Als Lösung entstand das im vorigen Unterabschnitt beschriebene Simulations-Bundle, welches sowohl Anwender-Klassen aufnehmen als auch zu allen benötigten Plugin-Bundles eine statische Abhängigkeit aufbauen kann.

Überraschend war jedoch, dass die Java-Technik zum Serialisieren und Deserialisieren von Java-Objekten, welches z. B. beim Persistieren eines Simulationszustandes,

aber auch beim Persistieren von Schattennetzen und bei Clipboard-Operationen im grafischen Editor eingesetzt wird, von der Classloader-Delegation betroffen ist. Beim Deserialisieren eines Datenstromes kommt Java-intern ebenfalls Reflection zum Einsatz, welche auf den Classloader zurückgreift, welcher für das den Deserialisierungsvorgang anstoßende Objekt maßgeblich ist. In den beschriebenen Fällen handelt es sich dabei um Objekte, welche auf Klassen zentraler Renew-Bundles wie z. B. dem Simulator oder dem JHotDraw-Editor basieren. Die Classloader dieser Bundles delegieren lediglich in Richtung ihrer statische Abhängigkeiten, kennen aber keine Klassen erweiternder Plugins. So kennt der Classloader des JHotDraw-Bundles keine Klassen des petrinetzspezifischen GUI-Bundles und der Classloader des Simulators keine Klassen der konkreten Formalismen. Im Java-Prototypen des Renew-PMS trat dieses Problem nicht auf, da alle Plugins zum selben Classloader gehörten. Eine Erweiterung der Classloader-Architektur wie in Abbildung 5.8 vorgeschlagen hätte das Problem allerdings auch im Java-Prototypen hervorgerufen.

Beim Deserialisieren eines Simulationszustandes bietet sich der `BottomClassloader` bzw. das Simulations-Bundle als dessen OSGi-Nachfolger als Mittel zur Lösung an, da dieses über die nötigen Classloader-Delegationsbeziehungen verfügt. Diese Lösung verdeutlicht zudem einen Aspekt, der auch im monolithischen Renew schon existierte, dort aber noch nicht innerhalb der Anwendung berücksichtigt wurde: Eine Wiederaufnahme einer persistierten Simulation ist natürlich nur möglich, wenn die Simulationsumgebung passend eingerichtet ist, d.h. insbesondere der anwendungsspezifische Classpath korrekt konfiguriert ist. Gleiches galt schon immer für das Aufsetzen einer Simulation auf Basis persistierter Schattennetze, da dann die Simulationskonfiguration der des Editors beim Erstellen der Netze entsprechen muss.

Im Bereich des grafischen Editors ist momentan noch keine Lösung implementiert. Hier ist das Simulations-Bundle zur Referenzierung aller sichtbaren Klassen ungeeignet, da es eine gänzlich andere Ausrichtung hat. Benötigt wird eine Referenzierung sämtlicher Klassen, welche zur Erweiterung der Editorfunktionalität in die von JHotDraw vorgegebene Klassenhierarchie zur Repräsentierung von Zeichnungselementen eingefügt werden. Aus dieser Formulierung heraus ergibt sich bereits eine mögliche Lösung: Wird die Erweiterung der Klassenhierarchie als *intendierte Erweiterbarkeit* verstanden, so sollte die Erweiterungsspezifikation so angepasst werden, dass alle den Editor erweiternden Bundles ihre Klassen (oder entsprechende Klassenlader) beim erweiterbaren Editor registrieren. So hat dieser jederzeit einen Überblick über alle verfügbaren Klassen für Zeichnungsobjekte.

Wie Simmendinger in [Sim07, Abschnitt 6.3.3] im Kontext des Ladens serialisierter Schattennetze erläutert, bietet gerade das OSGi-Rahmenwerk mehrere Mittel zur Lösung dieser Probleme auf Ebene des Rahmenwerkes anstelle der Ebene der Anwendung an. Dazu zählen die Direktiven `DynamicImport` und `BuddyPolicy` in der Deklaration von Bundle-Manifesten. Die `BuddyPolicy`-Direktive richtet Classloader-Delegationsbeziehungen in Gegenrichtung zu statischen Abhängigkeiten ein [siehe Ble06], wird aber nur in der Eclipse-Version des OSGi-Rahmenwerkes (Equinox) angeboten und ist daher für Renew nicht geeignet. Die `DynamicImport`-Direktive erlaubt einem OSGi-Bundle den Zugriff auf Packages anderer Bundles auch mit einer im voraus deklarierten optionalen anstelle einer statischen Abhängigkeit. Diese Technik ist prinzipiell für optionale Abhängigkeiten gedacht [OSG06a, Abschnitt 3.6.3, S. 42], ließe sich aber

mit einer *Catch-All*-Deklaration für das Referenzieren beliebiger noch unbekannter Klassen missbrauchen. Dieses Vorgehen würde allerdings die sorgsam eingeführten Sichtbarkeitsbeziehungen zwischen Bundles aushebeln und erscheint daher ebenfalls nicht ratsam.

**Persistente Konfiguration und Auslieferungsformat** Wie in Unterabschnitt 5.2.5 beschrieben, ist das PMS des Java-Prototypen darauf ausgelegt, beim Start des Systems die im Dateisystem installierten Plugins und damit eine womöglich außerhalb der Programmlaufzeit veränderte Konfiguration zu erkennen. Sämtliche Konfigurationsänderungen während der Laufzeit sind allerdings flüchtig, so dass sie das Ende der Laufzeit des Gesamtsystems nicht überdauern.

Im Gegensatz dazu werden sämtliche Änderungen an der Konfiguration eines OSGi-basierten Systems prinzipiell mittels Verwaltungsoperationen während der Laufzeit der Plattform durchgeführt (siehe Unterabschnitt 3.2.1.6). Die beim Beenden der Plattform aktuelle Konfiguration wird persistiert, so dass sie beim nächsten Start der Plattform wieder zur Verfügung steht [siehe OSG06a, S. 81 u. 94].

Im Kontext der Eclipse-Plattform besteht allerdings weiterhin die schon vor der Einführung des OSGi-Rahmenwerkes vorhandene Möglichkeit, auch außerhalb der Laufzeit der Plattform Konfigurationsänderungen mittels Dateisystemoperationen durchzuführen. Ein Rückgriff auf diesen Mechanismus würde wiederum die unerwünschte Abhängigkeit zur Eclipse-Plattform einführen und kommt daher nicht in Betracht.

Denkbar ist eine Lösung, welche die Teilfunktionalität des PMS des ersten Prototypen zum Auffinden installierter Plugins übernimmt, mit den im OSGi-Rahmenwerk installierten Bundles abgleicht und gegebenenfalls die notwendigen Konfigurationsänderungen durchführt. Diese Lösung ist allerdings noch nicht implementiert, da das Auslieferungsformat des OSGi-basierten Prototypen noch nicht festgelegt ist. Bisher kann das System lediglich experimentell (d.h. manuell) konfiguriert und gestartet werden.

### 5.4.5 Eclipse-Integration

Obwohl – wie in den vorangegangenen Unterabschnitten mehrfach betont – eine Abhängigkeit der Anwendung Renew von der Eclipse-Plattform nicht ratsam erscheint, so ist doch die direkte Integration der Java-Referenznetze in die Eclipse-IDE ein reizvolles Ziel. Diese Wünsche stehen nicht direkt im Widerspruch – sobald die Renew-Kernfunktionalität in Form von OSGi-Bundles realisiert ist, kann auf dieser eine Eclipse-Integrationsschicht aufsetzen, welche als Alternative zum weiterhin Eclipse-los nutzbaren Simulator oder grafischen Editor verfügbar ist.

Wie bereits in Unterabschnitt 2.1.4 erläutert, können die Java-Referenznetze als ein Java-Dialekt aufgefasst werden, welcher die sequentielle, textuelle Programmiersprache Java um einige intuitive grafische Konstrukte zur Implementierung von Nebenläufigkeit und Synchronisation ergänzt. Der Autor dieser Arbeit war bereits an der Entwicklung mehrerer Anwendungen beteiligt, deren Quellcode teils aus textuellem Java-Code und teils aus Java-Referenznetzen besteht [vgl. Duv01, Duv02, RDK+03]. Während der Entwicklung solcher Anwendungen wäre es eine enorme Arbeitserleichterung, nicht ständig zwischen verschiedenen IDEs (z. B. Eclipse und Renew) wechseln

zu müssen. Zudem besteht zwischen der Sprache Java und der Anschriftensprache der Java-Referenznetze eine sehr große Ähnlichkeit, so dass die vielen intelligenten, Javaspezifischen Hilfsfunktionen der Eclipse-IDE auch bei der Programmierung mit Java-Referenznetzen von Nutzen sein können.

Klaus Mitreiter hat in seiner Diplomarbeit [Mit08] mit fachlicher Betreuung durch den Autor dieser Arbeit verschiedene Versuche unternommen, Renew-Funktionalität in die Eclipse-IDE zu integrieren. Da der Simulator als grafiklose Anwendung wenig Beziehungen zur grafischen Oberfläche der Eclipse-IDE hat, lag der Schwerpunkt in Mitreiters Arbeit auf der Integration der Funktionalität des grafischen Editors.

Eine direkte Einbettung des JHotDraw-basierten Renew-Editors in Eclipse ist aus mehreren Gründen nicht ratsam. Zum einen basiert der JHotDraw-Editor auf der standardmäßig mit Java ausgelieferten Klassenbibliothek für grafische Benutzungsoberflächen namens *Abstract Windowing Toolkit (AWT)*, Eclipse hingegen auf einer konkurrierenden Eclipse-eigenen Klassenbibliothek namens *Standard Widget Toolkit (SWT)* [SWT]. Ein Mischbetrieb beider Klassenbibliotheken ist zwar prinzipiell möglich, erschwert aber die Verschmelzung der intelligenten Funktionen beider Oberflächen. Zudem wird der Mischbetrieb nicht auf allen Plattformen und/oder Java-Versionen unterstützt und führt gelegentlich zu Programmabstürzen. Da viele JHotDraw-Klassen sehr eng an das AWT gekoppelt sind, ist ein Austausch der Klassenbibliothek innerhalb des JHotDraw-Rahmenwerkes nicht leicht möglich, zudem würde dieser Austausch eine Abhängigkeit zu SWT und damit zum Eclipse-Rahmenwerk in den Renew-Editor einführen. [Vgl. Mit08, Kapitel 5]

Da die bestehende grafische Benutzungsoberfläche von Renew nicht einfach in Eclipse integriert werden kann, ist für die Bearbeitung von Java-Referenznetzen innerhalb der Eclipse-IDE die Implementierung eines neuen Editors nötig. Mitreiter hat verschiedene Rahmenwerke für grafische Editoren, welche bereits in das Eclipse-Rahmenwerk integriert sind, verglichen und auf ihre Eignung für einen Petrinetzeditor mit dem von Renew gewohnten Komfort zum Manipulieren von bipartiten Graphen mit textuellen Anschriften an Knoten und Kanten geprüft [vgl. Mit08, Kapitel 6]. Um den Aufwand der Pflege zweier grafischer Oberflächen für die Anwendung Renew zu begrenzen, hat Mitreiter zudem eine Klassenhierarchie zur Rahmenwerk-unabhängigen Repräsentierung von Petrinetz-Zeichnungen prototypisch erstellt, welche in beiden grafischen Oberflächen zum Einsatz kommen und als Grundlage sämtlicher Manipulationsoperationen verwendet werden kann [vgl. Mit08, Abschnitt 7.3].

Grundlage von Mitreiters Arbeit war eine Renew-Version, welche auf dem in Abschnitt 5.2 vorgestellte Java-Prototyp des Plugin-Systems basiert, da der OSGi-basierte Prototyp zu Beginn von Mitreiters Arbeit noch nicht existierte. Damit bestand das Problem, die beiden Eclipse und Renew zugrunde liegenden Plugin-Systeme kombiniert betreiben zu müssen. Die Lösung dieses Problems lieferte der bereits im konzeptionellen Modell skizzierte Ansatz, ein PMS als Plugin im Kontext eines anderen PMS zu installieren. Während die Einordnung der Verwaltungsfunktionalität des Renew-PMS innerhalb der Eclipse-Plattform ohne Schwierigkeiten zu bewerkstelligen war, stellte die Kombination der beiden Classloader-Hierarchien ein Problem dar. Zwar konnte der `PluginClassloader` des Renew-PMS so konfiguriert werden, dass er Plugins relativ zu seiner Position in der Eclipse-Verzeichnishierarchie fand und lud, damit waren aber sämtliche Klassen der Renew-Plugins den Eclipse-Classloadern nicht

bekannt. Somit bestand für den grafischen Netzeditor innerhalb der Eclipse-Architektur kein Zugriff auf die Funktionalität des Renew-Simulators. Als kurzfristige Lösung wurde daher der Renew-Classloader abgeschaltet, so dass alle Klassen der Renew-Plugins über den Eclipse-Classloader geladen werden, obwohl die Erweiterungskonfiguration weiterhin über das Renew-PMS stattfindet.

Der kombinierte Betrieb des Eclipse-Editors von Klaus Mitreiter mit der OSGi-basierten Renew-Version von Felix Simmendinger würde die Classloader-Hürden umschiffen. Die Kombination der beiden Prototypen erfordert aber eine Anpassung des noch auf die Schnittstellen des Java-Prototypen ausgelegten Eclipse-Netzeditors an die Schnittstellen des OSGi-Prototypen. Diese Anpassung wird einer der nächsten Schritte zur Weiterentwicklung des Projekts sein.

## 5.5 Diskussion

Die in diesem Kapitel vorgestellten Prototypen unterscheiden sich sowohl in ihrer Aufgabenstellung als auch in der zur Lösung herangezogenen Vorgehensweise und Programmierwerkzeuge sowie dem jeweils resultiertenden Funktionsumfang. Im Folgenden werden die Vor- und Nachteile der einzelnen Prototypen noch einmal zusammengefasst und vergleichend diskutiert.

**Java-Prototyp** Der stark von der damaligen nicht-dynamischen Eclipse-Architektur inspirierte Java-Prototyp erleichterte das Sammeln von Erfahrungen, um die im Netzmodell erarbeiteten Konzepte zu erkennen. Die Verwendung einer stark getypten, sequentiellen Sprache mit umfangreicher Klassenbibliothek erleichterte die Nachbildung von Plugin-Systemen, welche auf ähnliche Techniken und Möglichkeiten zurückgreifen.

Der in Abschnitt 5.2 präsentierte Prototyp baut ausschließlich auf der Standard-Klassenbibliothek der Java-Umgebung auf. Sämtliche zusätzliche Funktionalität zum dynamischen Laden und Integrieren von Plugins sowie zur Steuerung der Lebenszyklen wird in der Klasse `PluginManager` bzw. dem zugehörigen Java-Package neu implementiert. Die Auslieferung der Plugin-Verwaltung als Teil der letzten öffentlichen Renew-Versionen belegt die prinzipielle Tauglichkeit des Prototypen für den Einsatz in der Praxis, obwohl einige konzeptionelle Anforderungen nicht vollständig erfüllt sind.

Dem Anwender bietet das System eine einfache Konfigurationsmöglichkeit durch Installation von Plugins im Dateisystem außerhalb der Laufzeit des Systems sowie Kommandos zur dynamischen Rekonfiguration, welche lediglich die Benennung des hinzuzufügenden oder zu entfernenden Plugins verlangen. Die durch die Plugin-Verwaltung im System eingeführten Performanzeinbußen beschränken sich auf die Startphase, da hier sämtliche Plugins gefunden, geladen und konfiguriert werden, und auf die Endphase, da hier die Konfiguration wieder aufgelöst wird. Die Mechanismen zum Starten und Beenden des Gesamtsystems erlauben es, mehrere an sich eigenständige Benutzungsoberflächen durch verschiedene Plugins anbieten zu lassen und deren Laufzeiten erwartungskonform zu vereinen. Ein Anwender, der nur eine der Oberflächen nutzt, nimmt keinen durch das Plugin-System hervorgerufenen Unterschied in der Bedienung der Anwendung wahr.

Die Kopplung von Systemkomponenten zur Laufzeit wird in diesem Protypen auf der Basis bekannter Entwurfsmuster realisiert. Diese Herangehensweise präsentiert zwar dem Anwender eine einfache dynamische Rekonfiguration, das fehlende Rahmenwerk bürdet aber jedem Plugin-Entwickler die explizite Implementierung der Konfiguration auf. Insbesondere hat jeder Plugin-Entwickler auf die Symmetrie des Konfigurationsprozesses beim Hinzufügen *und* Entfernen zu achten – Fehler in der Implementierung eines Plugins können den gesamten Systemzustand beeinträchtigen. Der im PMS durchgesetzte Ausschluss von gleichzeitigen Konfigurationsänderungen sorgt für Threadsicherheit in der Verwaltung des Systems, schränkt allerdings die im System prinzipiell mögliche Nebenläufigkeit im Vergleich zum konzeptionellen Modell ein.

Zum dynamischen Laden des Codes und der Ressourcen von Plugins setzt der Java-Prototyp einen Classloader ein, welcher während der Laufzeit zwar neue Plugins aufnehmen, aber ein einmal geladenes Plugin nicht wieder aus dem Speicher entfernen kann. Somit führt das Entfernen eines Plugins lediglich zum Auflösen der Beziehungen zu anderen Plugins, aber nicht zum vollständigen Eliminieren von Code, Ressourcen und etwaiger globaler Zustandsinformation des Plugins. Im Einsatz als eher kurzfristig laufender Editor ist dieses Manko hinnehmbar, da ein Anwender gegebenenfalls die Anwendung beenden und neu starten kann, um die Reste eines entfernten Plugins endgültig zu beseitigen. Im Einsatz als langfristig laufende Ausführungsumgebung für petrinetzbasierte Softwaresysteme stellt dieses Manko jedoch eine erhebliche Beeinträchtigung der Systemstabilität dar.

**Netz-Prototypen** Die in Abschnitt 5.3 vorgestellten, direkt aus dem konzeptionellen Modell abgeleiteten referenznetzbasierten Prototypen illustrieren die bereits in Unterabschnitt 4.2.2 angesprochene Technik der fortschreitenden Verfeinerung von einer abstrakten Modellierung bis hin zur voll funktionstüchtigen Implementierung. Dieses Vorgehen hat den Vorteil, dass kein formaler Bruch zwischen konzeptionellem Modell und Implementierung auftritt, wie er bei einem Wechsel von einer stark auf Nebenläufigkeit ausgelegten Sprache wie den Petrinetzen zu einer stark auf sequentiellen Kontrollfluss ausgelegten Sprache wie den meisten textuellen Programmiersprachen unvermeidbar ist. Ein solcher Bruch führt nicht nur zu der im Java-Prototypen beobachtbaren Einschränkung der Nebenläufigkeit in der Plugin-Verwaltung, sondern birgt auch das Risiko, beim Umgang mit Nebenläufigkeit Fehlverhalten in die Implementierung einzubringen, welche im ursprünglichen Modell nicht vorhanden ist.

Ein Nachteil der Petrinetz-Modellierung ist der im Vergleich zur sequentiellen Programmiersprache stark erhöhte Aufwand bei der Implementierung einfacher sequentieller Algorithmen zur Berechnung von Funktionswerten oder zur Manipulation von Datenstrukturen, da Grundkonzepte wie die Hintereinanderausführung oder Schleifen explizit mit mehreren Transitionen, Stellen und zugehörigen Kanten nachgebildet werden müssen. Dieser Umstand wurde vom Autor bereits in früheren Arbeiten [siehe Duv01, Duv02] erläutert und als Lösung die gemischt-sprachige Implementierung mit Netzen und Java-Klassen vorgeschlagen. Dementsprechend ist keine Implementierung der Funktionalität des dynamischen Modullade-Mechanismus, deren Realisierung zweifelsohne auch in Java-Referenznetzen möglich wäre, im reinen Netz-Prototypen vorhanden. Der kombinierte Prototyp nutzt die Vorteile der gemischt-sprachigen

Implementierung, grafische Netze mit hoher Nebenläufigkeit mit komplexer Funktionalität in kompaktem Quelltext kombinieren zu können. Die Aufgaben der Plugin-Verwaltung werden zwischen Netz- und Java-Code so aufgeteilt, dass die Netze die Ablauf- und Konfigurationssteuerung übernehmen, während der Java-Code grundlegende Funktionalität wie z. B. die Klassenlader bereitstellt.

Da die Netze der Prototypen direkt aus denen des konzeptionellen Modells abgeleitet sind, übernehmen diese Prototypen die meisten Eigenschaften des Modells. Aufgrund des fehlenden Modullademechanismus ist nun gegenüber dem abstrakten Modell, welches diesen Aspekt ausblendet, im reinen Netz-Prototypen eine Einschränkung der Offenheit des Systems festzustellen, da nur die im Steuerungsnetz instantiierten Plugins ins System integriert werden können. Im kombinierten Prototypen ist diese Einschränkung allerdings wieder aufgehoben. Gleiches gilt für die im konzeptionellen Modell abstrakt gehandhabte Eigenschaft der separat deklarierbaren Beziehungen zwischen Plugins, welche ebenfalls nur im kombinierten Prototypen unterstützt wird.

Beide Netz-Prototypen sind auf ein lauffähiges Renew als Ausführungsumgebung angewiesen. Daher ist es nicht möglich, diese Prototypen als Basis für eine Renew-Installation zu verwenden. Diese grundlegende technische Einschränkung ergibt sich aus der zweiseitigen Betrachtung der Anwendung Renew im Kontext eines netzbasierten Prototypen. Zum einen wird die Anwendung Renew benötigt, um die Netze auszuführen, also sollten die Netze in einer Schicht oberhalb der Anwendung in der Softwarearchitektur angeordnet sein. Zum anderen soll die Anwendung Renew durch die vom Prototypen bereitgestellte Plugin-Verwaltung flexibilisiert werden, also sollten die Verwaltungsnnetze in einer Schicht unterhalb der Anwendung in der Softwarearchitektur angeordnet sein. Dieser Zyklus ließe sich aufbrechen, indem der Teil von Renew, welcher die Ausführung von Netzen ermöglicht, aus der Anwendung herausgelöst und zu einer reinen Laufzeitumgebung nahe der Ebene der virtuellen Java-Maschine umgeformt würde. Diese Laufzeitumgebung müsste genauso wie die Java-Klassenladerhierarchie ein dynamisches Laden und Entfernen von Netzen (Klassen) in verschiedenen Namensräumen bieten. Eine derartige Umstellung der Softwarearchitektur war leider im Kontext dieser Arbeit nicht zu bewältigen, da die interne Organisation des Renew-Simulators noch nicht auf eine unabhängige Verwaltung mehrerer Namensräume und unterschiedlicher Lebenszyklen für Netze ausgelegt ist.

**OSGi-Prototyp** Die primäre Motivation für die Erstellung eines weiteren in der Programmiersprache Java verfassten Prototypen auf der Basis des OSGi-Rahmenwerkes war der Wunsch, die im Modullademechanismus des ersten Java-Prototypen vorhandenen Unzulänglichkeiten durch Rückgriff auf ein ausgefeiltes, auf dynamische Rekonfigurierbarkeit ausgelegtes Modulverwaltungssystem zu beheben. Wie bereits in Abschnitt 5.4 angemerkt, fiel die Wahl auf das OSGi-Rahmenwerk auch aufgrund seiner Hersteller-unabhängigen Verfügbarkeit. Zudem besteht die Möglichkeit, jede OSGi-basierte Anwendung einfach in die Eclipse-IDE integrieren zu können.

Der OSGi-basierte Prototyp der Anwendung Renew befindet sich bisher in einem im Vergleich zu den anderen Prototypen sehr experimentellen Zustand. In ihm werden an unterschiedlichen Punkten im System verschiedene Wege ausprobiert, wie sich eine an der Beziehung der intendierten Erweiterbarkeit ausgerichtete Architektur in

einem Rahmenwerk abbilden lässt, welches sich an der Dienstbeziehung orientiert. Es ist sowohl möglich, nur die Modulladefunktionalität des OSGi-Rahmenwerkes zu nutzen und die darüber hinausgehende Verwaltungsfunktionalität wie im Java-Prototypen selber zu organisieren, als auch, die OSGi-unterstützten Dienstbeziehungen in verschiedenen Graden zur Verwaltung oder gar unmittelbaren Abbildung von Erweiterungsbeziehungen einzusetzen.

Aufgrund des experimentellen Status' des Prototypen erfordert die Konfiguration durch den Anwender derzeit detaillierte Kenntnisse über die Präzedenzen in der Startreihenfolge der Plugins, so dass die Anforderung der einfachen Konfigurierbarkeit momentan nicht erfüllt ist. Dieses Manko sollte sich allerdings relativ leicht durch eine Steuerung seitens des Renew-PMS oder durch eine Abbildung auf funktionale Abhängigkeiten beheben lassen. In [SCDK07] wird bereits angedeutet, dass sich die von Renew ausgeführten Petrinetze zur Steuerung der Konfiguration des OSGi-basierten Gesamtsystems einsetzen lassen könnten.

Aufgrund des ausgefeilten Modullademechanismus in der OSGi-Laufzeitumgebung sind zudem Probleme in der Definition der Abhängigkeiten zwischen den Renew-Plugins sichtbar geworden, welche aufgrund der Unzulänglichkeiten des ersten Java-Prototypen nicht zu Tage traten. Diese machten eine Überarbeitung der Dekomposition der in Abschnitt 5.1 beschriebenen Renew-Softwarearchitektur nötig. Als Ergebnis ergibt sich eine generell für integrierte Entwicklungsumgebungen (IDEs) wie Renew geeignete Technik, welche die Installation eines zur Laufzeit erstellten OSGi-Bundles nutzt, um den frisch bearbeiteten Code ins System zu integrieren und zur Ausführung zu bringen. Die statischen und funktionalen Abhängigkeiten dieses Bundles dienen zugleich der Konfiguration der zur Ausführung der gerade entwickelten Anwendung benötigten Plugins. Dabei wird der Unterschied zwischen Renew-Funktionalität und der Funktionalität anderer Bibliotheken auf der technischen Ebene aufgehoben.

Für den Entwickler von Plugins gilt, dass eine im Kontext der Plugin-Verwaltung angebotene Unterstützung zum Etablieren und Auflösen von Erweiterungsbeziehungen (oder auch Dienstbeziehungen) sowohl die Erstellung von Plugins erleichtert als auch das Risiko von Fehlern im Konfigurationsprozess der einzelnen Plugins senkt. Eine Unterstützung in Form von Entwurfsmustern, Netzkomponenten oder generischen Artefakten im Rahmenwerk ist daher sinnvoll. Während der im Java-Prototypen verfolgte Ansatz der Implementierung entlang bekannter Entwurfsmuster dennoch von jedem Entwickler eine Neu-Implementierung verlangt, können Netzkomponenten mit der immer gleichen Grundstruktur der Verwaltung und Konfiguration von Erweiterungen dem Entwickler redundante Arbeit ersparen. Die deklarative Konfiguration, wie sie beispielsweise von der OSGi-SCR unterstützt wird, übernimmt vollständig die Steuerung der Lebenszyklen. Aber auch diese bürdet noch jedem Plugin-Entwickler die konkrete Ausgestaltung der Reaktionen auf Lebenszyklus-Ereignisse auf, welche aufgrund des unterschiedlichen Charakters vieler Erweiterungsspezifikationen sehr vielfältig ausfallen.

Tabelle 5.1 fasst die Ergebnisse der Diskussion aller in diesem Kapitel vorgestellten Prototypen im Hinblick auf die in Abschnitt 4.1 erarbeiteten Anforderungen an ein Plugin-Konzept zusammen. Die Tabelle umfasst dieselben Spalten wie Tabelle 4.1, so dass die Ergebnisse direkt vergleichbar sind. Insgesamt ergibt sich eine relativ voll-

| System | offenes System | dynamisch rekonf. System | einfache Konfiguration | deklarative Konfiguration | Intendierte Erweiterbarkeit | Dienstnutzung | Kapselung auf Modulebene | formal definierter Lebenszyklus | mehrere Verwaltungsebenen | Verwaltungsebenenkopplung | unterstützt Nebenläufigkeit | Verwaltung threadsicher | visuelle Modellierung | präzise Semantik |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Petrinetzmodell | x | x | x | (x) | x | x | (x) | x | x | x | x | x | x | x |
| Renew-PMS (Java) | x | (x) | x | x | x | – | (x) | (x) | x | x | – | x | – | – |
| Reiner Netz-Prototyp | (x) | x | x | – | x | x | (x) | x | x | x | x | x | x | x |
| Netz-PMS mit Java | x | x | x | x | x | x | (x) | x | x | x | x | x | x | x |
| OSGi-basiert | x | x | (x) | x | (x) | x | x | x | x | x | ? | x | – | (x) |

(x): Die Eigenschaft ist nur eingeschränkt vorhanden (siehe Text).

Tabelle 5.1: Erfüllung der Anforderungen an ein Plugin-Konzept durch die Prototypen

ständige Abdeckung der Anforderungen durch alle Prototypen. Die in der Tabelle erkennbaren Schwachstellen der einzelnen Prototypen wurden bereits in den vorangegangenen Absätzen diskutiert. Ein System, das sämtliche Anforderungen erfüllt, könnte sich aus der Weiterentwicklung jedes dieser Prototypen ergeben, würde allerdings einen im Rahmen dieser Arbeit nicht leistbaren Aufwand zum Schließen der jeweiligen Lücken erfordern.

# 6 Schluss

Diese Arbeit betrachtet Plugin-Systeme aus einem konzeptionellen Blickwinkel heraus. Das Konzept des Plugins ist trotz häufiger Verwendung in der Praxis eher unscharf gefasst und in vielen Verwendungsfällen nicht präzise definiert. Eine oftmals systeminhärent vorhandene oder durch Plugins einführbare Nebenläufigkeit wird selten thematisiert. Ziel dieser Arbeit ist daher, eine intuitiv verständliche, präzise und auf nebenläufige Nutzung ausgelegte Semantik hinter den Begriff des Plugins zu legen. Die praktische Umsetzbarkeit der erarbeiteten Konzepte wird zudem am Beispiel Renew in mehreren Prototypen ausgetestet. Die Ergebnisse dieser Betrachtung sollen in den folgenden Abschnitten noch einmal zusammengefasst und diskutiert sowie Möglichkeiten zur Weiterführung der Gedanken aufgezeigt werden.

## 6.1 Zusammenfassung

Das Thema dieser Arbeit ergibt sich in Kapitel 1 aus der Beobachtung, dass viele Softwaresysteme existieren und weite Verbreitung finden, welche mittels so genannter Plugins erweiterbar sind oder deren Softwarearchitektur gar vollständig aus Plugins aufgebaut ist. Eine einheitliche und präzise Definition des Plugin-Konzepts fehlt jedoch. Das Konzept eines Plugins ergibt sich meist aus technischen Gegebenheiten heraus, eine Struktur- oder Verhaltensspezifikation muss oft aus einer in einer sequentiellen Programmiersprache verfassten Implementierung herausgearbeitet werden. Zwar tragen einige Forschungsdisziplinen der Informatik Konzepte zur Beschreibung von Plugin-Systemen bei, behandeln aber dieses aus der Praxis nicht mehr wegzudenkende Konzept nur am Rande. Sowohl die praktisch eingesetzten als auch die konzeptionell beschriebenen Plugin-Systeme beachten die in modernen Softwaresystemen immer stärker zu beobachtende systeminhärente Nebenläufigkeit nicht. Da Plugin-Systeme offene Systeme sind, kann aber jederzeit Nebenläufigkeit eingeführt werden – beispielsweise durch Plugins mit parallel zur erweiterbaren Anwendung arbeitender Hintergrundfunktionalität.

Um sich dem Plugin-Konzept zu nähern, werden in Kapitel 2 zunächst einige Grundlagen erarbeitet. Diese umfassen vor allem Softwareentwicklungsparadigmen und Modellierungsansätze für Softwarearchitekturen, welche zur Beschreibung von Plugin-Architekturen eingesetzt werden oder sich zu diesem Zweck eignen. Es werden folgende Paradigmen und Modellierungsansätze betrachtet:

- Referenznetze (siehe Abschnitt 2.1) eignen sich als objektbasierter Petrinetzformalismus zur Modellierung von dynamisch rekonfigurierbaren, quasi-hierarchisch aufgebauten, hochgradig nebenläufig arbeitenden Systemen. Petrinetze bieten eine intuitiv verständliche, kompakte und formal spezifizierte Semantik zur

Modellierung von Zuständen, Ressourcen und Prozessen, welche inhärent nebenläufigen Systemen gerecht wird. Das den Referenznetzen zugrunde liegende Paradigma der *Netze-in-Netzen* folgt zudem dem Paradigma der objektorientierten Softwareentwicklung, so dass objektbasierte Abstraktions- und Dekompositionskonzepte zur Modellierung von Systemstrukturen zur Verfügung stehen.[1] Mit den Java-Referenznetzen, welche eng in die objektorientierte Programmiersprache Java integriert und durch das Werkzeug Renew modellier- und ausführbar sind, besteht zudem die Möglichkeit der Implementierung von Softwaresystemen mit den vorgenannten Eigenschaften. Einige konservative Erweiterungen der Petrinetzsyntax erlaubten zudem die Erstellung relativ kompakter Modelle ohne Beschränkung der Nebenläufigkeit.

- Das direkt auf der Objektorientierung aufbauende Paradigma der komponentenbasierten Softwareentwicklung (CBSE, siehe Abschnitt 2.2) wird in einigen Forschungsarbeiten als das Plugin-Systemen zugrunde liegende Paradigma gesehen. Es betont die Wiederverwendbarkeit generischer Softwarekomponenten durch Dritte als Bausteine in der Montage von Softwaresystemen, thematisiert aber nicht die Konfiguration des Systems beim Endanwender, wie sie in Plugin-Systemen die Regel ist.

- Der Ansatz einer Architekturbeschreibungssprache (ADL, siehe Abschnitt 2.3) zielt auf die Beschreibung der Komposition eines Softwaresystems aus Komponenten, welche untereinander verdrahtet und ineinander geschachtelt werden können. Als ADL kann neben vielen akademischen, formal fundierten Ansätzen auch ein Teilbereich der weit verbreiteten, aber weniger präzisen Unified Modeling Language (UML) bezeichnet werden. Zwar erlauben viele akademische ADLs nicht nur die formale Beschreibung der Systemstruktur, sondern auch des Verhaltens von Komponenten und Verbindern, um anhand der formalen Beschreibung Systemeigenschaften verifizieren zu können. Einige ADLs sind zudem in der Lage, die dynamische Rekonfiguration einer Architektur zur Laufzeit zu formalisieren, so dass sie sich zur Modellierung von Plugin-Systemen (mit einer zuvor bekannten Menge von Plugins) eignen. Die grafischen Ausdrucksmöglichkeiten beschränken sich aber in der Regel auf Strukturdiagramme, welche genau eine Systemkonfiguration beschreiben. Einige wenige ADLs nutzen zwar Petrinetze als Ausdrucksmittel und beinhalten somit eine grafisch intuitive Verhaltensbeschreibung, erlauben aber keine dynamische Rekonfiguration der Softwarearchitektur. Eine ADL, welche eine grafisch intuitive Modellierung offener, dynamisch rekonfigurierbarer Systeme bietet, existiert nicht.

- In der Objektorientierung hat sich die Verwendung von Entwurfsmustern als Mikroarchitekturen für wiederkehrende Situationen bei Entwurf und Implementierung einer Softwarearchitektur etabliert (siehe Abschnitt 2.4). Viele dieser Entwurfsmuster eignen sich aufgrund der von ihnen beförderten Lockerung der

[1] Das objektorientierte Softwareentwicklungsparadigma wird in dieser Arbeit nicht eigenständig behandelt, da es als längst etablierter Stand der Technik ohnehin als Grundlage sämtlicher in dieser Arbeit besprochenen Paradigmen und Ansätze dient. Elementare Konzepte der Objektorientierung werden lediglich begleitend zu den Konzepten der *Netze-in-Netzen* vorgestellt.

Kopplung von Objekten zur Definition von Erweiterungsschnittstellen in Plugin-Systemen, obwohl sie nicht explizit auf diesen Einsatzzweck festgelegt sind. Darüber hinaus definieren einige Autoren das Konzept des Plugins selber als Entwurfsmuster, setzen dabei allerdings zu unterschiedliche Schwerpunkte. In einem Vorgriff auf die spätere Untersuchung existierender Plugin-Systeme lassen sich Entwurfsmuster auf einer höheren Abstraktionsebene ausmachen, welche wiederkehrende Situationen beschreiben, die beim Einsatz von Plugins auftreten.

- In der agentenorientierten Softwareentwicklung (AOSE, siehe Abschnitt 2.5) werden auf mehreren Abstraktionsebenen Konzepte eingeführt, welche die Flexibilisierung von Softwaresystemen unterstützen, indem den einzelnen Systemkomponenten (Agenten) eine höhere Autonomie als in der objektorientierten Softwareentwicklung zugestanden wird. Zudem werden in einigen agentenorientierten Methoden wie z. B. der Paose Interaktionsprozesse als gleichrangige Architekturelemente zu strukturierenden Konzepten wie der Rolle betrachtet. Das Paradigma der AOSE beschäftigt sich zunächst überhaupt nicht mit der Modellierung oder Implementierung von Plugin-Systemen, es werden allenfalls gelegentlich Plugin-Systeme zur Konfiguration von Agentenplattformen oder individuellen Agenten eingesetzt. Allerdings berücksichtigt die AOSE die systeminhärente Nebenläufigkeit moderner Softwaresysteme, da sie sich mit verteilten und proaktiv agierenden Systemkomponenten beschäftigt und mehrere Interaktionsprozesse unabhängig voneinander stattfinden können. Dieser Aspekt fehlt in anderen, etablierten Softwareentwicklungsparadigmen oder -ansätzen.

Die den jeweiligen Paradigmen oder Ansätzen zugrunde liegenden architekturprägenden Konzepte werden in Abschnitt 2.6 vergleichend diskutiert, noch ohne eine endgültige Wertung bezüglich der Eignung zur Modellierung von Plugin-Systemen vorzunehmen.

Der nächste Schritt zur Annäherung an die zentralen Konzepte von Plugin-Systemen besteht in der Untersuchung diverser Anwendungen, welche ein solches System beinhalten. Die Auswahl der betrachteten Softwaresysteme beschränkt sich zum einen auf Open-Source-Software. Zum anderen ergibt sich die Auswahl aus dem Anspruch, mit möglichst wenig verschiedenen zu untersuchenden Anwendungen einen repräsentativen Querschnitt durch die vorhandene Vielfalt von Plugin-artigen Softwarearchitekturen zu bekommen. Die detaillierte Untersuchung wird daher dominiert von zwei komplexen Anwendungen, der Mozilla-Familie (siehe Abschnitt 3.1) und der Eclipse-Plattform (siehe Abschnitt 3.2), welche beide bereits eine längere Entwicklungszeit hinter sich haben und mehrere verschiedene Architekturen zur Integration von Plugins in sich vereinen.

Die in der Mozilla-Familie enthaltene Netscape-Plugin-API (NPAPI) steht stellvertretend für die Klasse aller Anwendungen, welche eine fachlich spezialisierte Erweiterungsschnittstelle definieren und den vom Betriebssystem angebotenen Modulladememechanismus flexibilisieren. Zudem weist die Dokumentation der NPAPI auf die Einhaltung bestimmter Konventionen zur Wahrung der Threadsicherheit hin, welche die Problematik der Nebenläufigkeit bereits in derart einfachen Plugin-Systemen verdeutlicht. Das zweite in der Mozilla-Familie enthaltene Plugin-System erlaubt mit so

genannten „Extensions“ eine beliebige Erweiterung der Anwendung über mehrere verschiedene Schnittstellen und Verwaltungen, welche jede für sich nicht die Bezeichnung Plugin-Architektur rechtfertigen. Die Kombination der Möglichkeiten in Verbindung mit einer einheitlichen Installationsroutine und Konfigurationsverwaltung ergibt jedoch eine Plugin-Architektur. Beide Plugin-Architekturen der Mozilla-Familie sind ursprünglich auf Konfigurationsänderungen außerhalb der Laufzeit der Anwendungen ausgelegt. Eine dynamische Rekonfiguration ist angedacht und teilweise implementiert, aber in vielen Fällen aufgrund von Beschränkungen der beteiligten Softwarekomponenten nicht nutzbar.

Die in Java implementierte Eclipse-Plattform basiert ursprünglich auf einer Softwarearchitektur, welche ausschließlich Plugins als Bausteine vorsieht und die deklarative Konfiguration von Erweiterungsbeziehungen und statischen Abhängigkeiten zwischen diesen Plugins unterstützt. Da diese Softwarearchitektur keine dynamische Rekonfiguration zuließ, wurde der Plugin-Lademechanismus später durch eine zum OSGi-Rahmenwerk kompatible Implementierung ersetzt. Obwohl das OSGi-Rahmenwerk auf die Dienstbeziehung als zentrales Konzept zur Kopplung der Systemkomponenten setzt, dominiert in der Eclipse-Plattform weiterhin die Erweiterungsbeziehung die Architektur.

In Abschnitt 3.3 wird die Untersuchung ausgeweitet, um weitere Variationen des Plugin-Konzepts aufzuzeigen. Es werden einige akademische und praxisnahe Plugin-Systeme angesprochen, welche die Konzepte von Eclipse-Plugins variieren oder präzisieren. Unabhängig davon stellt das NetBeans-Rahmenwerk eine flexible Softwarearchitektur basierend auf einfacheren Grundelementen bereit, welche im Ergebnis ebenfalls als Plugin-System bezeichnet werden kann. Eine einzige Arbeit (MagicBeans, siehe Unterabschnitt 3.3.4 und [Cha05]) nimmt ähnlich der Zielsetzung dieser Arbeit eine Formalisierung des Plugin-Konzepts vor, geht aber nicht auf Nebenläufigkeit ein. Im Kontext der AOSE werden Plugin-Systeme häufig eingesetzt, um die konzeptionelle Flexibilität von Agenten technisch zu implementieren, eine Nutzung agentenorientierter Konzepte ist bei der Implementierung dieser Plugin-Systeme aber nicht zu bemerken.

Ergänzend werden einige Forschungsdisziplinen vorgestellt, welche sich thematisch mit Fragen auseinandersetzen, die auch im Kontext von Plugin-Systemen relevant sind. Hierzu zählen Fragen der Kapselung auf Modulebene und der Unterstützung von dynamischer Konfiguration in der Programmiersprache Java, des dynamischen Ladens von Code sowie das Lösen von Versions- und Namenskonflikten in verteilt entwickelten, offenen Systemen. Diese Fragen und Konzepte stehen nicht im Fokus dieser Arbeit. Sie werden in der Forschung ausreichend behandelt, so dass auf verschiedene Lösungsstrategien zurückgegriffen werden kann. Zudem werden zwei Fragestellungen angesprochen, welche für die konzeptionelle Modellierung von Relevanz sind. Zum einen ist die Integration mehrerer Plugin-Systeme nicht trivial, falls dieselbe Funktionalität in mehreren Rahmenwerken zur Verfügung gestellt werden soll. Zum anderen lassen sich einige Konzepte aus dem Bereich der Betriebssysteme auf Plugin-Systeme übertragen.

Ein wiederkehrendes Element der Untersuchung existierender Plugin-Systeme stellt die Definition der von den Plugin-Verwaltungen vorgesehenen Plugin-Lebenszyklen dar, da diese später mit der konzeptionellen Modellierung abgeglichen werden. Einige

andere Eigenschaften der untersuchten Plugin-Systeme werden in Abschnitt 3.4 tabellarisch gegenübergestellt. Während bezüglich einiger Attribute eine starke Ähnlichkeit der Systeme auszumachen ist, tritt bezüglich anderer Attribute eine große Vielfalt an Konzepten zutage. Plugin-Systeme sind tendenziell flach organisiert und betonen eine einfache Systemkonfiguration („Installation“ von Plugins) durch Endanwender. Dagegen lassen sich mehrere verschiedene zur Kopplung der Systemkomponenten herangezogene Beziehungsarten identifizieren, darunter die Erweiterungs- und die Dienstbeziehung. Die in den Plugin-Systemen angetroffenen Eigenschaften lassen sich in die zuvor besprochenen Softwareentwicklungsparadigmen und Modellierungsansätze einordnen, da zu jedem der Ansätze eine teilweise Übereinstimmung besteht. Keiner der Ansätze überdeckt jedoch die Eigenschaften von Plugin-Systemen vollständig. Diese Erkenntnis bereitet den Grund für die in Kapitel 4 folgende inhaltliche Diskusion und Modellierung des Plugin-Konzepts.

Aus der vorangegangenen Untersuchung von Plugin-Systemen ergeben sich einige Definitionen sowie mehrere Anforderungen, die das Konzept eines Plugin-Systems erfüllen sollte (siehe Abschnitt 4.1). Hierzu zählen die grundlegende Eigenschaft der (dynamischen) Rekonfigurierbarkeit eines offenes Systems, welches das Hinzufügen und Entfernen von Systemkomponenten beherrscht, und die Einfachheit der Konfiguration, da sie durch den Endanwender in der Einsatzphase des Softwaresystems erfolgen soll. Für Plugin-Systeme ist die Beziehungsart der intendierten Erweiterbarkeit charakteristisch, welche in hybriden Systemen durch die Beziehungsart der Dienstnutzung ergänzt werden kann. Eine starke Kapselung der einzelnen Systemkomponenten erlaubt erst das Hinzufügen und Entfernen in der Einsatzphase, welches zudem einen Lebenszyklus definiert, der von der Plugin-Verwaltung auf verschiedene Weisen ausgestaltet werden kann und muss. Ferner hat sich aus der Untersuchung existierender Systeme ergeben, dass neben der rein technischen, zentralisierbaren Verwaltung von Plugins eine fachliche Verwaltung der Erweiterungen jeder einzelnen fachlichen Erweiterungsspezifikation erforderlich ist. Die Anforderungen an ein konzeptionelles Modell umfassen zudem eine Fähigkeit zum Umgang mit systeminhärenter Nebenläufigkeit sowie eine eingängige visuelle Modellierung mit präziser Semantik.

Ein Plugin wird als Modul definiert, welches während der Einsatzphase auf einfache Weise zu einem offenen, intendiert erweiterbaren System hinzugefügt und auch wieder daraus entfernt werden kann. Dieser Begriff wird gegen Objekte, Komponenten und Agenten abgegrenzt. Allen vier Begriffen gemeinsam sind die Eigenschaften der Kapselung, der separaten Kompilierbarkeit und der Kopplung zur Laufzeit. Objektinstanzen einer Klasse basieren auf demselben Code – mehrere Instanzen desselben Plugin-Codes sind in der Regel nicht von Nutzen. Zudem haben Objekte in der Regel eine kleinere Granularität als Plugins. Komponenten teilen mit Plugins zwar die Individualität und Granularität, sollen aber anders als auf ein bestimmtes erweiterbares System hin getrimmte Plugins generisch wiederverwendbar und handelbar sein. Der Agentenbegriff zielt eher auf Eigenschaften wie Flexibilität und Proaktivität als auf einfache Integrierbarkeit ab. Zudem zielt die Definition von Plugins im Softwareentwurf stärker auf eine fachlich abgeschlossene Funktionalität als es bei Objekten, Komponenten oder Agenten der Fall ist.

Zur Schärfung der zunächst textuell umrissenen Anforderungen und Eigenschaften von Plugin-Systemen erfolgt eine konzeptionelle Modellierung auf der Basis von Refe-

renznetzen. Die Konventionen und Eigenheiten der referenznetzbasierten Modellierung werden in Abschnitt 4.2 erläutert. Charakteristisch ist eine Modellierung durch fortschreitende Verfeinerung, welche aus der Konstruktion heraus zu hoher Nebenläufigkeit in den Modellen führt und mit bidirektionalen Schnittstellenbeschreibungen arbeitet. Von technischen Details realer Plugin-Systeme wird in den Referenznetzmodellen abstrahiert. Hierzu zählen beispielsweise der einem Plugin-System zugrunde liegende Modullademechanismus oder die konkrete fachliche Funktionalität der Anwendung, da diese Aspekte das zu modellierende Konzept offener, intendiert erweiterbarer Systeme nicht wesentlich beeinflussen. Die Modelle lassen sich somit als Schablonen interpretieren, welche zu konkreten Plugin-Systemen verfeinert werden können.

Das Modell des Plugin-Konzepts wird in Abschnitt 4.3 in mehreren Schritten aufgebaut und durch Verfeinerung weiterentwickelt. Die grundlegenden Eigenschaften des offenen Systems und der intendierten Erweiterbarkeit werden durch die Konzepte der unabhängigen und der rekursiven Erweiterbarkeit ergänzt, um sowohl hierarchische als auch nicht-hierarchische Beziehungsgeflechte zwischen Plugins beschreiben zu können. Zur Trennung der technischen Konfiguration der Systemkomposition von der fachlichen Integration von Funktionalität wird das Konzept des Plugin-Verwaltungssystem (PMS) eingeführt, welches in verschiedenen alternativen Varianten eine mehr oder weniger stark zentralisierte Konfiguration der Beziehungen im System ermöglicht. Das PMS nutzt dabei dieselben Konzepte zur Schnittstellenspezifikation und Verwaltung von Plugins wie sie schon für die grundlegende Modellierung der intendierten Erweiterbarkeit verwendet werden. Um die zentrale technische als auch die dezentralen fachlichen Erweiterungs-Verwaltungen untereinander zu koppeln oder die Konfiguration zu flexibilisieren, können Lebenszyklen und Dienstbeziehungen ins Modell eingearbeitet werden.

Alle bei der Untersuchung existierender Plugin-Systeme vorgefundenen Beziehungsarten und Lebenszyklusspezifikationen werden in Abschnitt 4.4 auf das konzeptionelle Modell abgebildet, soweit sie nicht ohnehin schon direkt Gegenstand der Modellierung waren. Zu diesem Zweck werden die Lebenszyklusspezifikationen (wie zuvor schon die Beziehungsarten) auf Gemeinsamkeiten untersucht und klassifiziert. In Abschnitt 4.5 werden das konzeptionelle Modell und die untersuchten Plugin-Systeme tabellarisch gegenübergestellt und mit den zuvor erarbeiteten Anforderungen an Plugin-Systeme abgeglichen. Keines der untersuchten Plugin-Systeme deckt wie das in dieser Arbeit erstellte Konzept sämtliche Anforderungen ab. Am nächsten kommt den Anforderungen das in Eclipse verwendete OSGi-Rahmenwerk, welches allerdings nicht auf die Beziehungsart der intendierten Erweiterbarkeit ausgelegt ist.

Die modellierten Konzepte verbinden viele Eigenschaften, welche jeweils einzelnen der eingangs vorgestellten Softwareentwicklungsparadigmen und -techniken zu eigen sind. Zudem genügt die verwendete Modellierungstechnik teilweise den Anforderungen an eine Architekturbeschreibungssprache.[2]

Die konzeptionelle Modellierung wurde parallel von der Erstellung mehrerer Prototypen unter verschiedenen technischen Voraussetzungen und Umgebungen begleitet, welche in Kapitel 5 vorgestellt werden. Als Anwendungsbeispiel dient das Petrinetz-

[2] Eine tiefergehende Zusammenfassung und Evaluation der Ergebnisse dieser Arbeit ist Gegenstand des Abschnitts 6.2.

werkzeug Renew, welches zunächst in mehrere Plugins zergliedert wird. Ein rein Java-basierter Prototyp implementiert ein Plugin-System mit möglichst geringen Performanzeinbußen zur Laufzeit und Dateisystem-basierter Konfigurationsmöglichkeit für Endanwender. Die im konzeptionellen Modell ausgeblendeten technischen Details des Modullademechanismus und der konkreten Funktionalität sind in diesem Prototypen umgesetzt. Dieser Prototyp nimmt allerdings einige Einschränkungen im Vergleich zu den im konzeptionellen Modell realisierten Anforderungen in Kauf, um möglichst wenig Verwaltungs-Overhead ins System einzuführen. Ein rein petrinetzbasierter Prototyp illustriert die durch die operationale Semantik des Referenznetzmodells mit geringem Aufwand erreichbare, funktionsfähige Implementierung eines Plugin-Systems. Die Netze lassen sich mit dem Java-Prototypen kombinieren, so dass Plugins beider Prototypen interagieren können. Zuletzt wird ein auf dem OSGi-Rahmenwerk aufsetzender Prototyp vorgestellt, an welchem experimentell verschiedene Möglichkeiten der Abbildung der Erweiterungsbeziehung auf die im OSGi-Rahmenwerk unterstützte Dienstbeziehung diskutiert werden. Auch in diesem Prototyp ist wiederum eine Kombination der in Java implementierten OSGi-Verwaltung mit einer in Netzen implementierten Steuerung denkbar.

## 6.2 Ergebnisse

Diese Arbeit leistet einen Beitrag zur Bewältigung der Komplexität heutiger Software-Systeme, indem sie einige Lücken im Stand der Technik und der Forschung im Bereich der Plugin-Systeme schließt. Die Arbeit reicht ein durchgängiges Konzept für den in der Praxis häufig auftretenden und in der informatischen Forschung nur am Rande betrachteten Begriff des Plugins nach. Das Konzept vereint in sich mehrere Eigenschaften, die von den individuellen praktischen Lösungen und den bisher mit dem Thema befassten oder verwandten Forschungsdisziplinen nicht oder nicht in dieser Kombination abgedeckt werden.[3]

**Architektur- und Entwurfsmuster** Das konzeptionelle Modell identifiziert die architektonischen Elemente des erweiterbaren Systems und der Erweiterung sowie die Beziehung zwischen diesen beiden Elementen als charakteristische Struktur von Plugin-Systemen. Damit beschreibt das Plugin-Konzept ein Architektur- oder Entwurfsmuster im Sinne der in den Abschnitten 2.3 und 2.4 vorgestellten Begriffe.[4] Einige der im Unterabschnitt 2.4.3 bereits vorgestellten Entwurfsmuster mit der Bezeichnung „Plugin“ nehmen allerdings eine deutlich technischere Sichtweise als das hier präsentierte Muster ein, welches explizit die Notwendigkeit technischer Verwaltung *und* fachlicher Integration beschreibt.

Das in dieser Arbeit vorgestellte Konzept geht über den üblichen Umfang von Architektur- und Entwurfsmustern hinaus, da es durch die Modellierung mit

[3] Die in diesem Abschnitt zusammengetragenen Ergebnisse wurden im Kontext der einzelnen Kapitel bereits diskutiert. Es sei hier auf die Abschnitte 2.6, 3.4, 4.1.9.1, 4.4.3, 4.5 und 5.5 verwiesen.

[4] Die Einordnung des Plugin-Konzepts als Architektur- oder Entwurfsmuster hängt von seiner Relevanz für die Softwarearchitektur ab. Beide Sichtweisen lassen sich begründen und kommen im praktischen Einsatz vor.

Petrinetzen eine vollständige operationale Semantik erhält und unmittelbar ausgeführt werden kann.

**Dynamische Komponenten** Das Konzept des Plugins weist eine große Ähnlichkeit zum Konzept der Komponente auf, wie es in den Disziplinen der CBSE (Abschnitt 2.2) und der Architekturbeschreibungssprachen (siehe Abschnitt 2.3) verwendet wird. In beiden Konzepten werden eine starke Kapselung und gut dokumentierte Schnittstellen benötigt. Einige ADLs sowie einige Forschungszweige der CBSE beschäftigen sich zudem wie das Plugin-Konzept mit dynamisch rekonfigurierbaren Systemen. Die enge Integration in ein bestimmtes erweiterbares System, der angestrebte fachliche Zusammenhang eines Plugins und die Integration durch Endanwender stellen jedoch drei Merkmale dar, welche das Plugin-Konzept vom Konzept der generisch wiederverwendbaren Komponente als Konstruktionselement eines vor Auslieferung montierten Systems unterscheiden.

Das modellierte Plugin-Konzept weist hinsichtlich der Dynamik eine Nähe zur Disziplin der AOSE (siehe Abschnitt 2.5) auf, welche weder der CBSE noch den ADLs zu eigen ist. Die Variation der Organisation wird allerdings durch die fest definierten Schnittstellen von Plugins in einem engeren Rahmen kanalisiert als dies in einem beliebig flexibel organisierten Multiagentensystemen der Fall ist.

**Visuelle Modellierung** Die grafische Aufbereitung des Plugin-Konzepts nach dem Paradigma der *Netze-in-Netzen* stellt eine eingängige Beschreibung des elementaren Mechanismus der unabhängigen und rekursiven intendierten Erweiterbarkeit dar. Die Netze visualisieren sowohl die Systemstruktur aus der Sicht jeder einzelnen Systemkomponente als auch die zum Etablieren, Nutzen und Auflösen der Beziehungen zwischen Systemkomponenten definierten Schnittstellen und Prozesse. Die visuelle Darstellung von Architekturbeschreibungssprachen (siehe Abschnitt 2.3) konzentriert sich in der Regel nur auf die Struktur des Systems, da Prozesse in der jeweiligen Sprache nicht oder zumindest nicht im selben Diagramm darstellbar sind. Allerdings ist die volle Eignung der in dieser Arbeit präsentierten Modellierungstechnik als Architekturbeschreibungssprache noch nicht abschließend untersucht worden.

**Präzise Semantik von Konfigurationsänderungen** Die Operationen Hinzufügen und Entfernen von Plugins wurden schon von Chatley in seiner Dissertation (siehe Unterabschnitte 2.3.4 und 3.3.4) mit einer präzisen operationalen Semantik hinterlegt. Er betrachtet jedoch ausschließlich zentral durchgeführte, atomare und global wirkende Konfigurationsänderungen im System. Das konzeptionelle Modell dieser Arbeit kann bei entsprechender Vorgabe der Lebenszyklusnetze auch dezentral gesteuerte, zeitbehaftete und lokal wirkende Konfigurationsänderungen beschreiben.

**Nebenläufigkeit** Das konzeptionelle Modell erlaubt allein durch eine geeignete Konstruktion aus Petrinetzen eine sehr hohe Nebenläufigkeit in der Verwaltung des offenen, dynamisch rekonfigurierbaren Plugin-Systems. Zudem berücksichtigt es grundsätzlich die Möglichkeit weitgehend unabhängig voneinander, also nebenläufig agierender Systemkomponenten. Eine ähnliche Betonung von Nebenläu-

figkeit findet sich in den betrachteten Disziplinen nur in der agentenorientierten Softwareentwicklung (siehe Abschnitt 2.5), welche allerdings weniger Vorgaben zur Strukturierung des Systems, also seiner Elemente und Beziehungen bietet.

**Praxistauglichkeit** Bei den in dieser Arbeit herausgearbeiteten Eigenschaften von Plugin-Systemen und ihrer konzeptionellen Modellierung handelt es sich nicht nur um ein rein theoretisches Konzept. Die Praxistauglichkeit des Modells wird anhand mehrerer Prototypen belegt. Der Einsatz eines Prototypen in der Anwendungssoftware Renew erlaubt einen Vergleich des Konzepts mit den vielen in der Praxis existierenden Softwaresystemen, welche ebenfalls eine Plugin-Architektur beinhalten und täglich erneut ihren Nutzen in der Praxis beweisen (siehe Kapitel 3). Diesen Systemen fehlt aber oftmals das konzeptionelle Fundament, welches mit dieser Arbeit nachgereicht wird.

Jede der vorgenannten Eigenschaften wird in der Regel nur in einem, selten auch in mehreren, aber nie in allen der vorgestellten Paradigmen, Techniken oder Systeme behandelt. Das in dieser Arbeit entwickelte Plugin-Konzept vereint hingegen diese Eigenschaften in einem Modell, welches flexibel an den jeweiligen Bedarf angepasst werden kann.

Entsprechend der Natur eines *konzeptionellen* Modells werden diverse technische Eigenschaften nicht detailliert behandelt, welche in den *in der Praxis* eingesetzten Plugin-Systemen einen nennenswerten Anteil der Implementierung der Plugin-Verwaltung (PMS) ausmachen. Hier ist insbesondere die zur Flexibilisierung der Systemkomposition notwendige Verwaltung zu nennen, welche dem von der Laufzeitumgebung bzw. dem Betriebssystem bereitgestellten Modulladememechanismus vorgeschaltet wird. Die vorgeschaltete Verwaltung überprüft statische Abhängigkeiten und findet Module im Dateisystem, um Fehlermeldungen des Modulladememechanismus aus dem Wege zu gehen. Eine solche vorgeschaltete Verwaltung ist allerdings in sämtlichen offenen Systemen notwendig und ihre Realisierung daher nicht speziell im Bereich der Plugin-Systeme zu betrachten. Weitere im konzeptionellen Modell nicht behandelte Themen sind das Repräsentationsformat zur Spezifikation der Eigenschaften und Beziehungen der Plugins sowie zentrale Vorgaben zur einheitlichen Gestaltung einer Benutzungsoberfläche. Da diese Themen in praktisch eingesetzten Systemen nicht mehr ausgeblendet werden können, haben konsequenterweise die implementierten Plugin-System-Prototypen im Kontext der Anwendung Renew entsprechende Lösungen parat.

Das in dieser Arbeit gewonnene konzeptionelle Modell von Plugin-Systemen erfüllt die in Abschnitt 1.2 genannten Eigenschaften an ein nützliches Konzept. Durch die Dominanz der Beziehungsart der intendierten Erweiterbarkeit wird das System klar strukturiert – sowohl die Rollen der einzelnen Systemkomponenten, ihre zugeordneten Aufgaben und Zuständigkeiten als auch ihre Beziehungen zueinander sind klar erkennbar. Die grafische Modellierung entlang des *Netze-in-Netzen*-Paradigmas liefert eine intuitiv verständliche Darstellung des Rahmens, den jedes erweiterbare System seinen Erweiterungen bietet. Diese Darstellung verfügt zudem Petrinetz-typisch über eine formal fundierte, operationale Semantik.

Die Netze der einzelnen Entwicklungsschritte des konzeptionellen Modells folgen alle einem einheitlichen Schema, die jeweils vorhandenen Eigenschaften sind aber beliebig

durch eine entsprechende Abwandlung der Netze kombinierbar. Somit eignet sich das konzeptionelle Modell im Sinne eines anpassbaren Entwurfsmusters zur Wiederverwendung in verschiedensten Varianten von Plugin-Systemen. Die grafische Ausgestaltung von Erweiterungsbeziehung, Lebenszyklusschnittstelle oder Dienstanbieterverzeichnis sorgt für den gewünschten Wiedererkennungswert der Modelle, auch wenn sie – wie im Netz-Prototypen – mit konkreter Funktionalität verzahnt sind.

Insgesamt liefert die Arbeit damit die nötigen Definitionen und Anforderungen zur Charakterisierung von Plugin-Systemen und stellt deren Zusammenspiel im konzeptionellen Modell dar. Die praktische Erprobung der Konzepte in mehreren Prototypen führte darüber hinaus zu einer auslieferungsfähigen, Plugin-fähigen Version der Anwendung Renew. Der Erfolg der Einführung des Plugin-Systems lässt sich nicht zuletzt daran ausmachen, dass seit der Einführung viele zusätzliche Plugins mit verschiedenen Formalismen und Werkzeugen zur Unterstützung der Paose entwickelt wurden.

## 6.3 Ausblick

Die Ergebnisse dieser Arbeit können sowohl auf der praktischen Ebene der Prototypen als auch auf der konzeptionellen Ebene in mehrere Richtungen weiterentwickelt werden.

Zunächst einmal handelt es sich bei der zur Entwicklung der Prototypen notwendigen Flexibilisierung der Software-Architektur der Anwendung Renew um einen kontinuierlichen Prozess. Die bereits durchgeführte Dekomposition der Anwendung in mehrere Plugins hat mit einer relativ groben Granularität entlang der offensichtlichen Erweiterungsschnittstellen stattgefunden. Mit jedem Plugin, welches zusätzlich ins System integriert werden soll, kann ein Bedarf nach zusätzlichen Erweiterungsspezifikationen entstehen, welche bisher monolithisch innerhalb eines Plugins realisierte Funktionalität flexibilisieren und nach außen konfigurierbar gestalten. Diese Rückkopplung von der Plugin-Entwicklung zur Architektur der erweiterbaren Anwendung ist möglich, da die meisten Renew-Plugins aus Forschungs- und Lehreprojekten des Arbeitsbereichs TGI heraus entstehen, an welchem auch der Renew-Kern entwickelt wird.

Daneben bildet der noch sehr experimentelle OSGi-Prototyp eine interessante Alternative zur Weiterentwicklung von Renew. Die Integration in eine weit verbreitete, von verschiedenen Herstellern implementierte Laufzeitumgebung verspricht den Wirkungskreis der Kernfunktionalität des Petrinetzsimulators zu vergrößern. Eine denkbare Steuerung der Konfigurationsprozesse im OSGi-System mittels ausführbarer Petrinetze öffnet zudem ein neues Anwendungsfeld, von welchem sämtliche Nutzer des OSGi-Rahmenwerkes profitieren.

Darüber hinaus bildet die Integration von Renew-Kernfunktionalität in das OSGi-Rahmenwerk eine gute Grundlage zur Verschmelzung von Renew-IDE und Eclipse-IDE. Beide IDEs haben einen teilweise redundanten Funktionalitätsumfang, dessen Pflege in Folge einer Verschmelzung leichter fallen sollte. Renew und der Java-Referenznetzformalismus könnten durch die Eclipse-Plattform möglicherweise eine größere Popularität und weitere Verbreitung erfahren. Im Gegenzug würde eine weit verbrei-

tete Java-IDE um eine nahtlos integrierte, intuitive grafische Programmiersprachenergänzung zum Umgang mit Nebenläufigkeit und Synchronisation angereichert werden. Wie bereits in Unterabschnitt 5.4.5 erläutert, existiert mit der Arbeit von Mitreiter [Mit08] bereits ein erster Prototyp eines in die Eclipse-Oberfläche integrierten Petrinetz-Editors, welcher allerdings noch ohne den OSGi-Unterbau arbeitet.

Auf der Ebene des Referenznetzmodells kann sich die detaillierte Betrachtung der gegen Ende der schrittweisen Modellierung als Ergänzung zur Erweiterungsbeziehung ins Modell integrierten Dienstbeziehung lohnen. Ähnlich wie bei der Erweiterungsbeziehung lassen sich auch bei der Dienstbeziehung verschiedene Modellierungsvarianten mit individuellen Vor- und Nachteilen diskutieren. Die im Modell verwendete Übergabe von Dienstanbieterreferenzen an interessierte Dienstnutzer kann z. B. durch eine vermittelte Kommunikation wie bei der Modellierung der Modulbeziehung ersetzt werden. Alternativ könnte auch eine Übergabe von Dienstnutzerreferenzen an Dienstanbieter oder eine direkte Verwaltung in einer der beteiligten Systemkomponenten statt in einer zentralen Registry erfolgen. Dieses Spektrum an Modellierungsmöglichkeiten konnte im Rahmen dieser Arbeit nicht in der gleichen Ausführlichkeit wie bei der Erweiterungsbeziehung erarbeitet werden. Im Kontext anderer Arbeiten am Arbeitsbereich TGI werden bereits Referenznetzmodelle zur Spezifikation und Implementierung von dienstbasierten Systemen – insbesondere WebServices – entwickelt [siehe MOO05, MOO04, Off03]. Die im Kontext dieser Arbeiten schwerpunktmäßig betrachtete Komposition von Diensten unter Berücksichtigung der zulässigen Interaktionsprozesse ergänzt einen dynamischen Verhaltensaspekt, der im hier diskutierten Plugin-Konzept durch die Festlegung von Erweiterungsspezifikationen statisch spezifiziert wird. Eine Integration dieser Modelle in die hier präsentierte Modellierung wäre denkbar und reizvoll.

Die im Rahmen dieser Arbeit begonnene Unterscheidung existierender Plugin-Systeme nach Dynamik, unterstützten Beziehungsarten und Lebenszyklen kann zu einer strengeren Klassifizierung von Plugin-Systemen ausgebaut werden. Zudem kann die Analyse auf die vielen fachlichen Erweiterungsschnittstellen innerhalb „reiner“ Plugin-Systeme ausgedehnt werden, um hier Entwurfsmuster, Lebenszyklen und Konfigurationsschnittstellen feiner zu unterscheiden. Mit den in Abschnitt 4.1 aufgestellten Anforderungen an Plugin-Systeme ist ein Grundstein für ein entsprechendes Klassifikationsrahmenwerk gelegt. Wie die Diskussion in Unterabschnitt 4.5.2 zeigt, bedienen schon die untersuchten Plugin-Systeme nicht alle der Anforderungen. Eine nützliche Klassifikation sollte neben grundlegenden Anforderungen auch Merkmale mit variierbaren Ausprägungen enthalten. In Tabelle 4.1 stellen die Unterscheidungen alternativer Konfigurationstechniken und unterstützter Beziehungsarten solche Merkmale dar.

Die in dieser Arbeit verwendete Modellierungstechnik zur Beschreibung von Softwarearchitekturen, welche das Paradigma der *Netze-in-Netzen* unter Referenzsemantik einsetzt, zeigt ihre Stärken gerade bei der Modellierung dynamischer, nebenläufiger und hierarchisch strukturierter Systeme, in denen Prozesse den gleichen Stellenwert wie die Struktur des Systems einnehmen. Die Verwendung dieser Modellierungstechnik ist daher nicht auf Plugin-Systeme beschränkt, sondern sollte sich für beliebige andere Softwarearchitekturen mit ähnlichen Eigenschaften eignen. Die beiden in dieser Arbeit zum Vergleich besprochenen Paradigmen, sowohl die CBSE als auch die AOSE, können von der Modellierungstechnik profitieren. Beide Paradigmen dienen der Komple-

xitätsbewältigung in großen Systemen, in welchen oft mit inhärenter Nebenläufigkeit zu rechnen ist (im Fall der AOSE wird sie sogar vorausgesetzt). Die präzise und intuitive Semantik einer petrinetzbasierten Modellierungssprache zur Beschreibung von Nebenläufigkeits- und Synchronisationsbereichen wird von den konventionell in den beiden Paradigmen zur Modellierung eingesetzten Sprachen nicht geboten.

Die Anwendbarkeit der Modellierungstechnik auf die Konzepte der AOSE lässt sich anhand des Mulan-Modells illustrieren, welches mit einer ähnlichen Modellierungstechnik erstellt wurde. Im Gegensatz zu den Modellen dieser Arbeit versteht sich die von Rölke in [Röl04] vorgestellte Mulan-Architektur allerdings als Referenzarchitektur für Multiagentensysteme, deren mittlere Ebenen (Plattform und Agent) für sämtliche Systeme gleichförmig, also ohne Anpassung an konkrete fachliche Funktionalität, nutzbar sind. Die Implementierung von Funktionalität erfolgt in Mulan ausschließlich in den Unternetzen des Agenten (z. B. Wissensbasis und Protokolle, siehe Abschnitt 2.5.2). Diesen Arten von Netzen sind jeweils statische Schnittstellen vorgegeben, welche keinerlei Bezug zur konkreten fachlichen Funktionalität des Agenten haben.

Ein Vergleich der Netze der Mulan-Referenzarchitektur mit den Netzen des Plugin-Konzepts wird in [CDMR06] skizziert. Es lässt sich eine mehrfache Analogie erkennen. So weist jede der drei äußeren Mulan-Ebenen eine strukturelle Ähnlichkeit zum Konzept des erweiterbaren Systems auf, so dass die Netzinstanzen der jeweils nächstinneren Ebene als Erweiterung betrachtet werden können. In der hierarchischen Variante der Mulan-Architektur werden genau diese Beziehungen zu einer beliebig tiefen Rekursion ähnlich der in dieser Arbeit besprochenen rekursiven Erweiterbarkeit zusammengefaltet. Eine Diskussion hierarchischer geschachtelter Plattformen im Vergleich zum azyklischen Beziehungsgraphen zwischen Plugins könnte daher die Analogie verstärken.

Aus der erkannten Ähnlichkeit heraus stellt sich die Frage, ob zwischen den Modellen tatsächlich eine strukturelle Verwandtschaft besteht. Im Konzept des Plugin-Systems wird bereits auf die Autonomie und Proaktivität der Plugins, wie sie in Multiagentensystemen auftreten, eingegangen. Weitere Ähnlichkeiten, aber insbesondere auch vorhandene Unterschiede sind zweifelsohne eine interessante Diskussion wert. Die Beziehungen zwischen Systemkomponenten werden in Multiagentensystemen nicht als Plattform-Agent-Beziehungen sondern als Agent-Agent-Beziehungen auf einer höheren Abstraktionsebene betrachtet. Diese Beziehungen werden nicht durch einzelne Netze, sondern durch die Gesamtheit aller zwischen zwei (oder mehr) Rollen zur Kommunikation eingesetzten Interaktionsprotokolle definiert. Durch diese zweite Ebene wird in der AOSE eine Flexibilität in den Beziehungen zwischen Systemkomponenten erreicht, welche im Plugin-Konzept zunächst nicht vorhanden ist. Der Preis, der dafür zu zahlen ist, besteht in einem hohen Overhead nicht nur beim Auf- und Abbau, sondern auch während der Kommunikation entlang der Beziehungen. Die Bedeutung von Migration in Plugin-Systemen muss in dieser Arbeit ebenfalls ausgeklammert bleiben.

Die Analogie zwischen Plugin-Konzept und MAS-Konzept ist eine detaillierte Betrachtung wert, da beide Konzepte wie eben erläutert sowohl gemeinsame als auch orthogonale Eigenschaften mitbringen. Aus der Betrachtung können sich daher Ansatzpunkte zur Übertragung von agentenorientierten Konzepten auf das Plugin-Konzept als auch in Gegenrichtung ergeben, von denen letztendlich beide Konzepte zur

Gestaltung von Softwarearchitekturen profitieren können. Um mit den Konzepten beider Ansätze zugleich arbeiten und die Ähnlichkeiten ausnutzen zu können, bietet es sich an, die beiden Konzepte zum Konzept des Plugin-Agenten zu verschmelzen. Erste Überlegungen in dieser Richtung finden sich in [Sch05, SCMD08].

Eine weitere Verschmelzung und Vereinheitlichung von Konzepten wäre im Sinne der von Moldt entwickelten Einheitentheorie denkbar [siehe Tel05, TM05, Hew09]. Die Einheitentheorie lässt sich ebenfalls durch geschachtelte Petrinetze beschreiben, wobei je nach gewünschter Ausdrucksstärke der Modellierung unterschiedliche Petrinetzformalismen zum Einsatz kommen. Im Gegensatz zur im Bereich der Architekturbeschreibungssprachen betonten grundlegenden Trennung zwischen Komponenten und Verbindern existiert in der Einheitentheorie nur ein einziges grundlegendes Konzept – die Einheit. Jede Einheit verfügt nicht nur über einen flexiblen Satz an Verhaltensweisen wie es bei Agenten der Fall ist, sondern auch über frei wählbare und dynamisch veränderliche Schnittstellen und innere Struktur. Durch unterschiedliche Schnittstellenmengen entstehen so unterschiedliche Typen von Einheiten. In diesem Rahmen wäre es einfachst möglich, Einheiten verschiedener Typen (also z. B. Plugin-Einheiten, Agenten-Einheiten usw.) zu einem System zu verbinden. Die von Benjamin Schleinzer in [Sch07] begonnene Verallgemeinerung der Mulan-Hierarchie arbeitet mit ähnlichen Konzepten. Hier unterscheiden sich Agenten, Plattformen und Protokolle äußerlich nur in der Menge ihrer verfügbaren Schnittstellen und können diese Menge während der Laufzeit anpassen. Die Integration von Plugin-Konzepten ist in diesem Kontext durch einfaches Hinzufügen einer Erweiterungsschnittstelle möglich. Die fachliche Bedeutung einer solchen Mischung oder Verschmelzung von Konzepten ist jedoch noch auszuformulieren.

Für den praktischen Nutzen von Konzepten und einen nennenswerten Erkenntnisgewinn während der Modellierung ist es zunächst sinnvoller, weniger flexible und dafür stärker strukturierende Konzepte zu verwenden. Genau diesen Ansatz verfolgt das in dieser Arbeit präsentierte Modell der Konzepte von Plugin-Systemen. Die Festlegung auf die Beziehungsart der intendierten Erweiterbarkeit mag zwar die Architekten und Entwickler eines Softwaresystems in ihren Freiheiten beschränken, befördert aber eine klare, einfach nachvollziehbare und dennoch flexible Systemstruktur.

# Abkürzungsverzeichnis

| | |
|---|---|
| ABI | Binärschnittstelle (*Application Binary Interface*) |
| ACL | *Agent Communication Language* |
| ACM | *Association for Computing Machinery* |
| ADL | Architekturbeschreibungssprache (*Architecture Description Language*) |
| ADT | Abstrakter Datentyp (*Abstract Data Type*) |
| AIP | Agenteninteraktionsprotokoll (*Agent Interaction Protocol*) |
| AMS | *Agent Management System* |
| AOP | *Agent-Oriented Programming* |
| AOSE | Agentenorientierte Softwareentwicklung (*Agent-Oriented Software Engineering*) |
| API | Programmierschnittstelle (*Application Programming Interface*) |
| APT | *Advanced Packaging Toolkit* |
| ARPA | *Advanced Research Projects Agency* |
| AUML | *Agent* UML |
| AWT | *Abstract Windowing Toolkit* |
| BDI | *Beliefs–Desires–Intentions* |
| bzw. | beziehungsweise |
| ca. | etwa (circa) |
| CAP.NET | *Client Application Platform .NET* |
| CAPA | *Concurrent Agent Platform Architecture* |
| CBD | Komponentenbasierte Softwareentwicklung (*Component-Based Development*) |
| CBSE | Komponentenbasierte Softwareentwicklung (*Component-Based Software Engineering*) |
| CCM | CORBA *Component Model* |
| CID | Eindeutiger Klassenname (*Class ID*) |
| COM+ | *Component Object Model* |
| CORBA | *Common Object Request Broker Architecture* |
| CVS | *Concurrent Versioning System* |
| DF | *Directory Facilitator* |
| DLL | *Dynamic Link Library* |
| DOM | *Document Object Model* |
| DSO | *Dynamic Shared Object* |
| DVM | *Dynamic Classes-enabled Virtual Machine* |
| EDM | *Enterprise Deployment Model* |
| EJB | *Enterprise Java Beans* |
| etc. | und so weiter (et cetera) |
| et al. | und andere (et alii) |

| | |
|---|---|
| evtl. | eventuell |
| Ext. | *Extension* |
| f. | und folgende Seite |
| ff. | und folgende Seiten |
| Fig. | *Figure* |
| FIPA | *Foundation for Intelligent Phyisical Agents* |
| ggf. | gegebenenfalls |
| GI | Gesellschaft für Informatik |
| GUI | Grafische Benutzungsoberfläche (*Graphical User Interface*) |
| Hrsg. | Herausgeber |
| HTML | *Hypertext Markup Language* |
| HTTP | *Hypertext Transfer Protocol* |
| ID | Eindeutiger Name (*Identifier*) |
| IDE | Integrierte Entwicklungsumgebung (*Integrated Development Environment*) |
| IDL | Schnittstellendefinitionssprache (*Interface Definition Language*) |
| IEEE | *Institute of Electrical and Electronic Engineers* |
| IID | Eindeutiger Schnittstellenname (*Interface ID*) |
| IP | *Internet Protocol* |
| ISO | *International Standard Organization* |
| J2SE | *Java 2, Standard Edition* |
| JADE | *Java Agent Development Environment* |
| JAR | Java-Archivdatei (`.jar`) |
| JCP | *Java Community Process* |
| JDK | *Java Development Kit* |
| JDT | *Java Development Tools* |
| JIT-Compiler | *Just-In-Time-Compiler* |
| JPF | *Java Plugin Framework* |
| JSR | *Java Specification Request* |
| JVM | Java-Laufzeitumgebung (*Java virtual machine*) |
| Kap. | Kapitel |
| KBE | *Knowledge Base Editor* |
| KI | Künstliche Intelligenz (*Artificial Intelligence, AI*) |
| KIF | *Knowledge Interchange Format* |
| KQML | *Knowledge Query and Manipulation Language* |
| KSE | *Knowledge Sharing Effort* |
| LNCS | *Lecture Notes in Computer Science* |
| MadKit | *Multi-agent Development Kit* |
| MAS | Multiagentensystem (*Multi-agent System*) |
| MASIF | *Mobile Agent System Interoperability Facility* |
| MDA | *Model Driven Architecture* |
| MDC | *Mozilla Developer Center* |
| MIL | *Module Interconnection Language* |
| MIME | *Multipurpose Internet Mail Extensions* |
| MIT | *Massachusetts Institute of Technology* |
| MULAN | Multiagentennetze (*Multi-agent Nets*) |

| | |
|---|---|
| NPAPI | *Netscape Plugin Application Programming Interface* |
| Nr. | Nummer |
| OCL | *Object Constraint Language* |
| OMG | *Object Management Group* |
| OO | Objektorientierung (als Sammelbegriff) |
| OOSE | Objektorientierte Softwareentwicklung (*Object-oriented Software Engineering*) |
| OpenCSA | *Open Composite Services Architecture* |
| OSGi | *Open Service Gateway Initiative* |
| PAOSE | Petrinetzbasierte/Prozessorientierte AOSE |
| PDE | *Plugin Development Environment* |
| PDF | *Portable Document Format* |
| PIM | *Platform Independent Model* |
| PMS | Pluginverwaltung (*Plugin Management System*) |
| Prov. | *Provider* |
| PSM | *Platform Specific Model* |
| RCP | *Rich Client Platform* |
| Renew | *Reference Net Workshop* |
| RNW | Renew-Zeichnungsdatei (`.rnw`) |
| RPC | Entfernter Methodenaufruf (*Remote Procedure Call*) |
| RPM | RPM *Package Manager* |
| S. | Seite |
| SCA | *Service Component Architecture* |
| SCR | *Service Component Runtime* |
| SIG | Fachgruppe (*Special Interest Group*) |
| SL | *Semantic Language* |
| SNS | Renew-Schattennetzsystem (*Shadow Net System*, `.sns`) |
| SO | *Shared Object* |
| SOA | dienstorientierte Architektur *Service-oriented Architecture* |
| SOSE | dienstorientierte Softwareentwicklung *Service-oriented Software Engineering* |
| SPI | *Service Provider Interface* |
| SR | Diensteverzeichnis (*Service Registry*) |
| SWT | *Standard Widget Toolkit* |
| TCP | *Transmission Control Protocol* |
| UDP | *User Datagram Protocol* |
| UI | Benutzungsoberfläche (*User Interface*) |
| UML | *Unified Modeling Language* |
| usw. | und so weiter |
| vgl. | vergleiche |
| VKI | Verteilte Künstliche Intelligenz (*Distributed Artificial Intelligence*) |
| vs. | gegenüber (versus) |
| WfMC | *Workflow Management Coalition* |
| WSDL | *Web Service Definition Language* |
| XML | *Extensible Markup Language* |
| XPCOM | *Cross-Platform Component Object Model* |

| | |
|---|---|
| XPI | XPInstall-Archivdatei (`.xpi`) |
| XPInstall | *Cross-Platform Install* |
| XPT | *Cross-Platform Type Library* |
| XUL | XML *User Interface Language* |
| z.B. | zum Beispiel |

# Literaturverzeichnis


[Aal97] Wil M. P. van der Aalst: *Verification of Workflow Nets*. In: Pierre Azéma und Gianfranco Balbo (Herausgeber) *Applications and Theory of Petri Nets1997: 18th International Conference, ICATPN'97, Toulouse, France, June 1997. Proceedings*, Band 1248 von *LNCS*, S. 407–426, Springer-Verlag, Berlin, 1997, ISBN 3-540-63139-9.

[Aal00] Wil van der Aalst: *Inheritance of Interorganizational Workflows: How to Agree to Disagree Without Loosing Control*. BETA Working Paper Series, WP 46, Eindhoven University of Technology, Eindhoven, 2000.

[ABH+07a] Wil M. P. van der Aalst, Michael Beisiegel, Kees M. van Hee, Dieter König und Christian Stahl: *An SOA-based architecture framework*. International Journal of Business Process Integration and Management (IJBPIM), Band 2 (2007), Nr. 2: S. 91–101, doi:10.1504/IJBPIM.2007.015132.

[ABH+07b] Wil M. P. van der Aalst, Michael Beisiegel, Kees M. van Hee, Dieter König und Christian Stahl: *An SOA-based architecture framework*. Computer Science Report 07/02, Technische Universiteit Eindhoven, The Netherlands, Januar 2007.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno und Vijay Machiraju: *Web Services: Concepts, Architectures and Applications*. Data-centric systems and applications, Springer-Verlag, Berlin, 2004, ISBN 3-540-44008-9.

[ACM03] Deepak Alur, John Crupi und Dan Malks: *Core J2EE Patterns: Best Practices and Design Strategies*. Core design series, Prentice Hall Professional Technical Reference (PH PTR), 2. Auflage, 2003, ISBN 0-13-142246-4.

[ADCR01] Gul Agha, Fiorella De Cindio und Grzegorz Rozenberg (Herausgeber): *Advances in Petri Nets: Concurrent Object-Oriented Programming and Petri Nets*, Band 2001 von *LNCS*. Springer-Verlag, Berlin, 2001.

[AISJ77] Christopher Alexander, Sara Ishikawa, Murray Silverstein und Max Jacobson: *A Pattern Language: Towns, Buildings, Construction*. Nummer 2 in Center for Environmental Structure series, Oxford Univ. Press, New York, 1977, ISBN 0-19-501919-9.

[AKA08] *Arbeitskreis Architektur- und Entwurfsmuster (AKAEM)*. Gesellschaft für Informatik, Fachgruppe Softwarearchitektur, 2008. Im Internet: http://www.architekturmuster.de/, gesichtet mit letzter Änderung vom 7. Mai 2008, 5:51 Uhr (der Arbeitskreis wurde im Mai 2008 aufgelöst).

[All] *The Alloy Analyzer*. Im Internet: http://alloy.mit.edu/.

[And00] Jesper Andersson: *A deployment system for pervasive computing*. In: *ICSM 2000. International Conference on Software Maintenance, San Jose, CA, USA. Proceedings*, S. 262–270, IEEE Computer Society, Los Alamitos, Oktober 2000, doi:10.1109/ICSM.2000.883058.

[And03] Andreas Andresen: *Komponentenbasierte Softwareentwicklung: mit MDA, UML und XML*. Hanser, München [u.a.], 2003, ISBN 3-446-22282-0.

[App01] Hedley Apperly: *Configuration Management and Component Libraries*. In: Heineman und Councill [HC01], Kapitel 29, S. 513–526.

[Arc] Institute for Software Research, University of California: *ArchStudio – Software and Systems Development Environment*. Im Internet: http://www.isr.uci.edu/projects/archstudio/.

[Arn04] Karine Arnout: *From Patterns to Components*. Dissertation, ETH Zürich, März 2004.

[Bai00] Edward C. Bailey: *Maximum RPM*. RedHat, Inc., Durham, USA, 1. Auflage, 2000, ISBN 1-888172-78-9. Im Internet: http://www.redhat.com/docs/books/max-rpm/, gesichtet am 25. Januar 2008, 15:30 Uhr, Seitenangaben beziehen sich auf die PDF-Version.

[Bak03] Mitchell Baker: *Browser Innovation, Gecko and the Mozilla Project*. Mozilla.org, März 2003. Im Internet: http://www.mozilla.org/browser-innovation.html, gesichtet mit letzter Änderung vom 16. September 2004.

[Bar07] Neill Bartlett: *A Comparison of Eclipse Extensions and OSGi Services*. Eclipse community article, EclipseZone, Februar 2007. Im Internet: http://www.eclipsezone.com/articles/extensions-vs-services/.

[Bas01] Len Bass: *Software Architecture Design Principles*. In: Heineman und Councill [HC01], Kapitel 21, S. 389–403.

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel und Michel Wermelinger: *A survey of self-management in dynamic software architecture specifications*. In: *Self-Managed Systems. 1st ACM SIGSOFT workshop, WOSS '04, Newport Beach, California. Proceedings*, S. 28–33, ACM, New York, 2004, ISBN 1-58113-989-6, doi:10.1145/1075405.1075411.

[BCK03] Len Bass, Paul Clements und Rick Kazman: *Software Architecture in Practice*. Addison-Wesley, Boston, New York, München u.a., 2. Auflage, 2003.

[Bir05] Dorian Birsan: *On Plug-ins and Extensible Architectures*. ACM Queue, Band 3 (2005), Nr. 2: S. 40–46, ISSN 1542-7730, doi:10.1145/1053331.1053345.

[BK03] Morgan Björkander und Cris Kobryn: *Architecting Systems with UML 2.0*. IEEE Software, Band 20 (2003), Nr. 4: S. 57–61, ISSN 0740-7459, doi:10.1109/MS.2003.1207456.

[BL07] Hongyu Pei Breivold und Magnus Larsson: *Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles*. In: *Software Engineering and Advanced Applications. 33rd EUROMICRO Conference, SEAA'2007, Lübeck, Germany. Proceedings*, S. 13–20, IEEE Computer Society, Los Alamitos, August 2007, ISSN 1089-6503, doi:10.1109/EUROMICRO.2007.25.

[Ble06] Alex Blewitt: *Eclipse – a tale of two VMs (and many classloaders)*. Eclipse community article, EclipseZone, Mai 2006. Im Internet: http://www.eclipsezone.com/articles/eclipse-vms/.

[Bol03] Azad Bolour: *Notes on the Eclipse Plug-in Architecture*. Eclipse corner article, Eclipse, Juli 2003. Im Internet: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.

[Boo94] Grady Booch: *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings series in object-oriented software engineering, Benjamin/Cummings, Redwood City, Calif. [u.a.], 2. Auflage, 1994, ISBN 0-8053-5340-2.

[Bra04] Jeremy S. Bradbury: *Organizing Definitions and Formalisms of Dynamic Software Architectures*. Technischer Bericht 2004-477, Queen s University, März 2004. Im Internet: http://faculty.uoit.ca/bradbury/research.html.

[BRP+] F. Bellifemine, G. Rimassa, A. Poggi, T. Trucco, G. Caire und F. Bergenti: *Java Agent Development Framework (JADE)*. Im Internet: http://jade.tilab.com/.

[BTW07] Tim Boudreau, Jaroslav Tulach und Geertjan Wielenga: *Rich Client Programming: Plugging into the NetBeans Platform*. Prentice Hall, 2007, ISBN 978-0-132-35480-6.

[Buf04] Robert Buffone: *Divided Loyalty: Create Plugins that Work in Eclipse and NetBeans*. DevX (Jupitermedia Corporation), September 2004. Im Internet: http://www.devx.com/opensource/Article/21904, gesichtet am 30. Januar 2008, 9:50 Uhr.

[Cab02] Lawrence Cabac: *Entwicklung von geometrisch unterscheidbaren Komponenten zur Vereinheitlichung von Mulan-Protokollen*. Studienarbeit, Universität Hamburg, Fachbereich Informatik, 2002.

[Car03] Timo Carl: *Evaluation und beispielhafte Erweiterung einer referenznetzbasierten Agentenumgebung*. Studienarbeit, Universität Hamburg, Fachbereich Informatik, August 2003.

[CD92] Søren Christensen und Niels Damgaard Hansen: *Coloured Petri Nets Extended with Channels for Synchronous Communication*. Technischer Bericht DAIMI PB-390, Computer Science Department, Aarhus University, 1992.

[CD07] Lawrence Cabac und Till Dörges: *Tools for Testing, Debugging and Monitoring Multi-agent Applications*. In: Moldt et al. [MKH+07], S. 209–213.

[CDD+07] Lawrence Cabac, Till Dörges, Michael Duvigneau, Christine Reese und Matthias Wester-Ebbinghaus: *Application Development with Mulan*. In: Moldt et al. [MKH+07], S. 145–159.

[CDMR05] Lawrence Cabac, Michael Duvigneau, Daniel Moldt und Heiko Rölke: *Modeling Dynamic Architectures Using Nets-within-Nets*. In: Gianfranco Ciardo und Philippe Darondeau (Herausgeber) *Applications and Theory of Petri Nets 2005. 26th International Conference, ICATPN 2005, Miami, USA, June 2005. Proceedings*, Band 3536 von *LNCS*, S. 148–167, 2005, ISBN 978-3-540-26301-2, doi:10.1007/11494744_10.

[CDMR06] Lawrence Cabac, Michael Duvigneau, Daniel Moldt und Heiko Rölke: *Applying Multi-agent Concepts to Dynamic Plug-in Architectures*. In: Joerg Mueller und Franco Zambonelli (Herausgeber) *Agent-Oriented Software Engineering VI: 6th International Workshop, AOSE 2005, Utrecht, Netherlands, July 21, 2005. Revised Selected Papers*, Band 3950 von *LNCS*, S. 190–204, Springer-Verlag, Berlin, Juni 2006, doi:10.1007/11752660_15.

[CDR06a] Lawrence Cabac, Ragna Dirkner und Heiko Rölke: *Modelling Service Dependencies for the Analysis and Design of Multi-Agent Applications*. In: Moldt [Mol06], S. 291–298.

[CDR06b] Lawrence Cabac, Michael Duvigneau und Heiko Rölke: *Net Components Revisited*. In: Moldt [Mol06], S. 87–102.

[CEK+04] Robert Chatley, Susan Eisenbach, Jeff Kramer, Jeff Magee und Sebastián Uchitel: *Predictable Dynamic Plugin Systems*. In: Michel Wermelinger und Tiziana Margaria (Herausgeber) *Fundamental Approaches to Software Engineering. 7th International Conference, FASE 2004 (part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004), Barcelona, Spain, March/April 2004. Proceedings*, Band 2984 von *LNCS*, S. 129–143, Springer-Verlag, Berlin, 2004, ISBN 3-540-21305-8, doi:10.1007/b95935.

[CEM03] Robert Chatley, Susan Eisenbach und Jeff Magee: *Modelling a Framework for Plugins*. In: *Specification and Verification of Component-Based Systems. Workshop SAVCBS'03 at ESEC/FSE '03, Helsinki, Finnland, September 2003. Proceedings*, S. 49–57, 2003. Im Internet: http://www.cs.iastate.edu/~leavens/SAVCBS/2003/ (vollständige Proceedings).

[CEM04] Robert Chatley, Susan Eisenbach und Jeff Magee: *MagicBeans: a Platform for Deploying Plugin Components*. In: Emmerich und Wolf [EW04], S. 97–112, doi:10.1007/b98000.

[CH01] Bill Councill und George T Heineman: *Definition of a Software Component and Its Elements*. In: Heineman und Councill [HC01], Kapitel 1, S. 5–19.

[Cha05] Robert Chatley: *Predictable Dynamic Plugin Architectures*. Dissertation, University of London, Imperial College London, Department of Computing, Februar 2005. Im Internet: http://www.chatley.com/articles/.

[CJ02] Cristiano Castelfranchi und W. Lewis Johnson (Herausgeber): *Autonomous Agents and Multiagent Systems, First International Joint Conference, AAMAS 2002, Bologna, Italy. Proceedings*, ACM Press, New York, 2002, ISBN 1-58113-480-0.

[CK07] Lawrence Cabac und Nicolas Knaak: *Process Mining in Petri Net-based Agent-oriented Software Development*. In: Moldt et al. [MKH+07], S. 7–21.

[CKMR06] Lawrence Cabac, Nicolas Knaak, Daniel Moldt und Heiko Rölke: *Analysis of Multi-Agent Interactions with Process Mining Techniques*. In: *Multiagent System Technologies. 4th German Conference, MATES 2006 Erfurt, Germany. Proceedings*, Band 4196 von *LNCS*, S. 12–23, Springer-Verlag, Berlin, 2006, ISBN 978-3-540-45376-5, doi:10.1007/11872283_2.

[CMR03] Lawrence Cabac, Daniel Moldt und Heiko Rölke: *A Proposal for Structuring Petri Net-Based Agent Interaction Protocols*. In: Wil van der Aalst und Eike Best (Herausgeber) *Applications and Theory of Petri Nets2003: 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 2003. Proceedings*, Band 2679 von *LNCS*, S. 102–120, Springer-Verlag, Berlin, 2003.

[CR06] Eric Clayberg und Dan Rubel: *Eclipse: building commercial-quality plug-ins*. The Eclipse series, Addison-Wesley, Boston, New York, München u.a., 2. Auflage, 2006, ISBN 0-321-42672-X.

[CSA] *OASIS Open Composite Services Architecture (CSA)*. Im Internet: http://www.oasis-opencsa.org/.

[Cul99] Timothy Culp: *Industrial Strength Pluggable Factories*. The C++ Report, Band 11 (1999), Nr. 9: S. 21–26, ISSN 1040-6042.

[CW01] Paolo Ciancarini und Michael J. Wooldridge (Herausgeber): *Agent-Oriented Software Engineering. First International Workshop, AOSE 2000, Limerick, Ireland, June 2000. Revised Selected Papers*, Band 1957 von *LNCS*, Springer-Verlag, Berlin, 2001, ISBN 3-540-41594-7, doi:10.1007/3-540-44564-1.

[Dar] *Darwin – An Architecture Description Language*. Im Internet: `http://www-dse.doc.ic.ac.uk/Software/Darwin/`.

[Deb02] *APT HOWTO*. Debian.Org, 2002. Im Internet: `http://www.debian.org/doc/manuals/apt-howto/index.en.html`, gesichtet mit letzter Änderung vom August 2005 (Version 1.8.11, englisch).

[DeL05] Scott A. DeLoach: *Engineering Organization-Based Multiagent Systems*. In: Alessandro F. Garcia, Ricardo Choren, Carlos José Pereira de Lucena, Paolo Giorgini, Tom Holvoet und Alexander B. Romanovsky (Herausgeber) *Software Engineering for Multi-Agent Systems IV. Research Issues and Practical Applications (Post-Proceedings of the 4th International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, SELMAS'05)*, Band 3914 von *LNCS*, S. 109–125, Springer-Verlag, 2005, ISBN 3-540-33580-3, doi:10.1007/11738817_7.

[DF06] Friedrich Delgado Friedrichs: *Integration eines Model Checking Tools in die Petrinetzentwicklungsumgebung Renew*. Studienarbeit, Universität Hamburg, Department Informatik, September 2006.

[DF07] Friedrich Delgado Friedrichs: *Referenznetze mit Anschriften in Scheme*. Diplomarbeit, Universität Hamburg, Department Informatik, September 2007.

[Dir06] Ragna Dirkner: *Roundtrip-Engineering im PAOSE-Ansatz*. Diplomarbeit, Universität Hamburg, Department Informatik, 2006.

[DK06] Linda DeMichiel und Michael Keith: *JSR 220: Enterprise JavaBeans$^{TM}$, Version 3.0. EJB Core Contracts and Requirements*. Sun Microsystems, 2006.

[DMR02] Michael Duvigneau, Daniel Moldt und Heiko Rölke: *Concurrent Architecture for a Multi-agent Platform*. In: Fausto Giunchiglia, James Odell und Gerhard Weiß (Herausgeber) *Agent-Oriented Software Engineering. 3rd International Workshop, AOSE 2002, Bologna. Proceedings*, S. 147–159, ACM Press, Juli 2002.

[DMR03] Michael Duvigneau, Daniel Moldt und Heiko Rölke: *Concurrent Architecture for a Multi-agent Platform*. In: Fausto Giunchiglia, James Odell und Gerhard Weiß (Herausgeber) *Agent-Oriented Software Engineering III. Third International Workshop, Agent-oriented Software Engineering (AOSE) 2002, Bologna, Italy, July 2002. Revised Papers and Invited Contributions*, Band 2585 von *LNCS*, S. 59–72, Springer-Verlag, Berlin, 2003.

[DRW06] Michael Duvigneau, Heiko Rölke und Frank Wienberg: *Informal Introduction to the Feature Structure Nets Tool – A Tool for Process and Information Modeling*. In: Daniel Moldt (Herausgeber) *Proceedings of the*

*13th Workshop Application and Tools for Petri Nets. AWPN'06*, Nummer FBI-HH-B-267/06 in Report of the Department of Informatics, S. 85–91, Universität Hamburg, Department Informatik, 2006.

[Duv01] Michael Duvigneau: *Referenznetze bei der Erstellung der verteilten Anwendung des Börsenspiels*. Studienarbeit, Universität Hamburg, Fachbereich Informatik, 2001.

[Duv02] Michael Duvigneau: *Bereitstellung einer Agentenplattform für petrinetzbasierte Agenten*. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Dezember 2002.

[DWE98] Sophia Drossopoulou, David Wragg und Susan Eisenbach: *What is Java binary compatibility?* In: Freeman-Benson und Chambers [FBC98], S. 341–361, doi:10.1145/286936.286974.

[EBP01] Claudia Ermel, Roswitha M. Bardohl und Julia Padberg: *Visual Design of Software Architecture and Evolution based on Graph Transformation*. Electronic Notes in Theoretical Computer Science, Band 44 (2001), Nr. 4: S. 52–66, ISSN 1571-0661, doi:10.1016/S1571-0661(04)80943-5.

[Ecl] *Eclipse*. Im Internet: http://www.eclipse.org/.

[EDH06] Clemènt Escoffier, Didier Donsez und Richard S. Hall: *Developing an OSGi-like Service Platform for .NET*. In: *Consumer Communications and Networking. 3rd IEEE Conference, CCNC 2006, Las Vegas, USA. Proceedings*, Band 1, S. 213–217, IEEE Computer Society, Los Alamitos, Januar 2006, ISBN 1-4241-4244-0085-6, doi:10.1109/CCNC.2006.1593018.

[Esk99] Philip Eskelin: *Component Interaction Patterns*. In: *Pattern Languages of Programs (PLoP '99). Proceedings*, 1999. Im Internet: http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/.

[EW04] Wolfgang Emmerich und Alexander L. Wolf (Herausgeber): *Component Deployment. Second International Working Conference, CD 2004, Edinburgh, UK, May 2004. Proceedings*, Band 3083 von *LNCS*, Springer-Verlag, Berlin, 2004, ISBN 3-540-22059-3, doi:10.1007/b98000.

[FBC98] Bjorn Freeman-Benson und Craig Chambers (Herausgeber): *Object-Oriented Programming Systems, Languages and Applications. 13th ACM SIGPLAN Conference, OOPSLA '98, Vancouver, Canada. Proceedings*, Nummer 10 in ACM SIGPLAN Notices, ACM Special Interest Group on Programming Languages, ACM Press, New York, Oktober 1998, ISBN 1-58113-005-8.

[FDH06] Yujian Fu, Zhijiang Dong und Xudong He: *Modeling, Validating and Automating Composition of Web Services*. In: *Web Engineering. 6th international conference, ICWE '06. Proceedings*, S. 217–224, ACM, New York, 2006, ISBN 1-59593-352-2, doi:10.1145/1145581.1145626.

[Fer01] Jaques Ferber: *Multiagentensysteme – Eine Einführung in die Verteilte Künstliche Intelligenz*. Addison-Wesley, Boston, New York, München u.a., 2001. Deutsche Übersetzung von Stefan Kirn (frz. Original von 1995).

[FG98] Jaques Ferber und Olivier Gutknecht: *A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems*. In: *Multi Agent Systems. International Conference, ICMAS 1998, Paris. Proceedings*, S. 128–135, Juli 1998, ISBN 0-8186-8500-X, doi:10.1109/ICMAS.1998.699041.

[FH08] Peter Fischer und Peter Hofer: *Lexikon der Informatik*. Springer-Verlag, Berlin, 14. Auflage, 2008, ISBN 978-3-540-72549-7, doi:10.1007/978-3-540-72550-3.

[FIP] *Foundation for Intelligent Physical Agents (FIPA)*. Im Internet: http://www.fipa.org.

[FIP00a] Foundation for Intelligent Physical Agents: *FIPA Agent Management Specification*. 28. August 2000. Im Internet: http://www.fipa.org/specs/fipa00023/.

[FIP00b] Foundation for Intelligent Physical Agents: *FIPA Agent Management Support for Mobility Specification*. 30. Juni 2000. Im Internet: http://www.fipa.org/specs/fipa00087/.

[FIP01a] Foundation for Intelligent Physical Agents: *FIPA Abstract Architecture Specification*. 29. August 2001. Im Internet: http://www.fipa.org/specs/fipa00001/.

[FIP01b] Foundation for Intelligent Physical Agents: *FIPA Communicative Act Library Specification*. 29. Januar 2001. Im Internet: http://www.fipa.org/specs/fipa00037/.

[FMB05] Jaques Ferber, Fabien Michel und J Baez: *AGRE: Integrating Environments with Organizations AGRE: Integrating Environments with Organizations*. In: Danny Weyns, H. Van Dyke Parunak und Fabien Michel (Herausgeber) *Environments for Multi-Agent Systems. First International Workshop, E4MAS 2004, New York, USA, 2004. Revised Selected Papers*, Band 3374 von *LNAI*, S. 48–56, Springer-Verlag, 2005, ISBN 978-3-540-24575-9, doi:10.1007/b106134.

[Fow03] Martin Fowler: *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, New York, München u.a., 2003, ISBN 0-321-12742-0. Enthält Beiträge von David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee und Randy Stafford. Kurzübersicht der Patterns Im Internet: http://martinfowler.com/eaaCatalog.

[Fow04] Martin Fowler: *Inversion of Control Containers and the Dependency Injection pattern*. 2004. Im Internet: http://www.martinfowler.com/articles/injection.html, gesichtet mit letzter Änderung vom 23. Januar 2004.

[FSF99] *Gnu Lesser General Public License*. Free Software Foundation, Inc., Februar 1999. Im Internet: http://www.gnu.org/licenses/lgpl-2.1.txt, Version 2.1.

[FW93] T. Finin und J. Weber: *Draft Specification of the KQML AgentCommunication Language*. The DARPA Knowledge Sharing Initiative, External Interfaces Working Group. Im Internet: http://www.cs.umbc.edu/kqml/, 1993.

[GB03] Erich Gamma und Kent Beck: *Contributing to Eclipse: principles, patterns, and plug-ins*. The eclipse series, Addison-Wesley, Boston, New York, München u.a., 2. Auflage, 2003, ISBN 0-321-20575-8.

[Gen95] Michael R. Genesereth: *Draft Specification of the Knowledge Interchange Format*. ANSI X3T2 Ad Hoc Group on KIF. Im Internet: http://www.cs.umbc.edu/kse/kif/, 1995.

[GFM] Olivier Gutknecht, Jacques Ferber und Fabien Michel: *MadKit (Multi-Agent Development Kit)*. Im Internet: http://www.madkit.net/.

[GH06] Olivier Gruber und Richard S. Hall: *A Java Framework for Building and Integrating Runtime Module Systems*. In: Robert Meersman und Tari Zahir (Herausgeber) *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE. OTM Confederated International Conferences, Montpellier, France. Proceedings, Part II*, Band 4276 von *LNCS*, S. 1772–1789, Springer-Verlag, Berlin, 2006, ISBN 978-3-540-48274-1, doi:10.1007/11914952_50.

[GHJV98] Erich Gamma, Richard Helm, Ralph Johnson und John M. Vlissides (Herausgeber): *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series, Addison-Wesley, Boston, New York, München u.a., 15. Auflage, 1998, ISBN 0-201-63361-2.

[GHM+05] Olivier Gruber, B. J. Hargrave, Jeff McAffer, Pascal Rapicault und Thomas Watson: *The Eclipse 3.0 platform: Adopting OSGi technology*. IBM Systems Journal, Band 44 (2005), Nr. 2: S. 289–299.

[GJSB05] James Gosling, Bill Joy, Guy L. Steele Jr. und Gilad Bracha: *The Java Language Specification*. The Java Series, Prentice Hall Professional Technical Reference (PH PTR), 3. Auflage, 2005, ISBN 0-321-24678-0. Im Internet: http://java.sun.com/docs/books/jls/, Seitenangaben beziehen sich auf die PDF-Version.

[GK94] Michael R. Genesereth und Steven P. Ketchpel: *Software agents*. Communications of the ACM, Band 37 (1994), Nr. 7: S. 48–53, 147, ISSN 0001-0782, doi:10.1145/176789.176794.

[Gor06] Ian Gorton: *Essential Software Architecture*. Springer-Verlag, Berlin, 2006, ISBN 978-3-540-28713-1.

[GP01] Martin L. Griss und Gilda Pour: *Accelerating Development with Agent Components*. Computer, Band 34 (2001), Nr. 5: S. 37–43, ISSN 0018-9162, doi:10.1109/2.920610.

[Gri98] Frank Griffel: *Componentware : Konzepte und Techniken eines Softwareparadigmas*. dpunkt-Lehrbuch, dpunkt-Verlag, Heidelberg, 1. Auflage, 1998, ISBN 3-932588-02-9.

[Gri01a] Frank Griffel: *Verteilte Anwendungssysteme als Komposition klassifizierter Softwarebausteine: ein Komponenten-basierter Ansatz zur Generativen Softwarekonstruktion*. Dissertation, Universität Hamburg, Fachbereich Informatik, 2001.

[Gri01b] Martin L. Griss: *Software Agents as Next Generation Software Components*. In: Heineman und Councill [HC01], Kapitel 36, S. 641–657.

[Gro06] JSR-277 Expert Group: *JSR-277: Java Module System – Early Draft*. Sun Microsystems, Inc. – Java Community Process, 2. Oktober 2006. Im Internet: http://jcp.org/aboutJava/communityprocess/edr/jsr277/index.html.

[GV03] C. Girault und Rüdiger Valk (Herausgeber): *Petri Nets for Systems Engineering: A Guide to Modelling, Verification, and Applications*, Springer-Verlag, Berlin, 2003.

[Han08] Jeff Hanson: *The case for Java modularity – Two newer JSRs tackle an age-old problem*. JavaWorld.com, 26. August 2008. Im Internet: http://www.javaworld.com/javaworld/jw-08-2008/jw-08-java-modularity.html, gesichtet am 5. September 2008.

[HC01] George T. Heineman und William T. Councill (Herausgeber): *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Boston, New York, München u.a., 2001, ISBN 0-201-70485-4.

[HD02] Xudong He und Yi Deng: *A Framework for Developing and Analyzing Software Architecture Specifications in SAM*. The Computer Journal, Band 45 (2002), Nr. 1: S. 111–128, doi:10.1093/comjnl/45.1.111.

[Hew09] Marcin Hewelt: *Unit Editor (UnitEd): Modellierung komplexer Systeme auf Basis eines einfachen Rahmenwerkes*, 28. April 2009. Diplomarbeitsvortrag, Universität Hamburg, Department Informatik.

[HP06] Petr Hnětynka und František Plášil: *Dynamic Reconfiguration and Access to Services in Hierarchical Component Models*. In: Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski und Kurt Wallnau (Herausgeber) *Component-Based Software Engineering. 9th International Symposium, CBSE 2006, Västerås, Sweden, June 2006. Proceedings*, Band 4063 von *LNCS*, S. 352–359, Springer-Verlag, Berlin, 2006.

[HS98] Michael N. Huhns und Munindar P. Singh: *Agents and Multiagent Systems: Themes, Approaches and Challenges*. In: Michael N. Huhns und Munindar P. Singh (Herausgeber) *Readings in Agents*, S. 1–23, Morgan Kaufman, San Francisco, CA, 1998.

[HT03] Petr Hnětynka und Petr Tůma: *Fighting Class Name Clashes in Java Component Systems*. In: László Böszörményi und Peter Schojer (Herausgeber) *Modular Programming Languages. Joint Modular Languages Conference, JMLC 2003, Klagenfurt, Austria, August 2003. Proceedings*, Band 2789 von *LNCS*, S. 106–109, Springer-Verlag, Berlin, 2003, ISBN 3-540-40796-0.

[Hya99] Dave Hyatt: *The XPToolkit Architecture*. Mozilla.org, 1999. Im Internet: `http://www.mozilla.org/xpfe/xptoolkit/`, gesichtet mit letzter Änderung vom 28. Oktober 2006.

[IEE00] Institute of Electrical and Electronics Engineers, New York: *Recommended Practice for Architectural Description of Software-Intensive Systems*. 2000. Identisch zu ISO/IEC Standard 42010:2007.

[Iri04] Luis Iribarne: *Web Components: A Comparison between Web Services and Software Components*. Colombian Journal of Computation (RCC), Band 5 (2004), Nr. 1.

[ISO07] International Standards Organization/International Electrotechnical Commission, Genf: *Systems and software engineering – Recommended practice for architectural description of software-intensive systems*. Juli 2007, doi:10.1109/IEEESTD.2007.386501. Identisch zu IEEE Standard 1471-2000.

[Jac02] Thomas Jacob: *Implementierung einer sicheren und rollenbasierten Workflowmanagement-Komponente für ein Petrinetzwerkzeug*. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 2002.

[JBC+07] Eric Jendrock, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin und Kim Haase: *The Java EE 5 Tutorial – For Sun Java System Application Server 9.1*. Sun Microsystems, September 2007. Im Internet: `http://java.sun.com/javaee/5/docs/tutorial/doc/`, Seitenangaben beziehen sich auf die PDF-Version.

[Jen92] Kurt Jensen: *Coloured Petri Nets: Basic Concepts*. Monographs in Theoretical Computer Science, Springer-Verlag, Berlin, 1992, ISBN 3-540-55597-8.

[Jen00] N. R. Jennings: *On Agent-Based Software Engineering*. Artificial Intelligence, Band 117 (2000), Nr. 2: S. 277–296. Im Internet: `http://www.ecs.soton.ac.uk/~nrj/pubs.html`.

[JKMUN02] Thomas Jacob, Olaf Kummer, Daniel Moldt und Ulrich Ultes-Nitsche: *Implementation of Workflow Systems using Reference Nets – Security*

*and Operability Aspects*. In: Kurt Jensen (Herausgeber) *Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, University of Aarhus, Department of Computer Science, Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark, August 2002. DAIMI PB: Aarhus, Denmark, August 28–30, number 560.

[JPF] *JPF (Java Plugin Framework)*. Im Internet: `http://jpf.sourceforge.net/`.

[JSR06a] *Java Specification Request (JSR) 277: Java Module System*. Im Internet: `http://www.jcp.org/en/jsr/detail?id=277`, gesichtet mit letzter Änderung vom 13. November 2006, 2006.

[JSR06b] *Java Specification Request (JSR) 294: Improved Modularity Support in the Java Programming Language*. Im Internet: `http://www.jcp.org/en/jsr/detail?id=294`, gesichtet mit letzter Änderung vom 9. Mai 2006, 2006.

[JSR07] *Java Specification Request (JSR) 291: Dynamic Component Support for Java SE*. Im Internet: `http://www.jcp.org/en/jsr/detail?id=291`, gesichtet mit letzter Änderung vom 7. August 2007, 2007.

[JSW98] Nicholas R. Jennings, Katia Sycara und Michael Wooldridge: *A Roadmap of Agent Research and Development*. Autonomous Agents and Multi-Agent Systems, Band 1 (1998), Nr. 1: S. 7–38.

[Kno] *Knopflerfish – Open Source OSGi Service Platform*. The Knopflerfish Project. Im Internet: `www.knopflerfish.org`.

[KO05] Michael Köhler und Jan Ortmann: *Formal Aspects for service modelling based on high-level Petri Nets*. In: *International Conference on Intelligent Agents, Web Technologies and Internet Commerce (IAWTIC 2005)*, 2005.

[Köh04] Michael Köhler: *Objektnetze: Definition und Eigenschaften*, Band 1 von *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.

[Kor02] Andreas Kornstädt: *Ein Einschubsystem für die musikwissenschaftliche Analyse: basierend auf einem Ansatz zur bruchlosen Modellierung von Anwendungsfamilien mit Rahmwerken und Komponenten*. Dissertation, Universität Hamburg, Fachbereich Informatik, Universität Hamburg, Fachbereich Informatik, 2002. Im Internet: `http://www.sub.uni-hamburg.de/opus/volltexte/2002/639/` (elektronischer Dissertationsserver der Staats- und Universitätsbibliothek Hamburg, `urn:nbn:de:gbv:18-6396`).

[KQ05] Dia Kharrat und Syed Salman Quadri: *Self-Registering Plug-ins: an Architecture for Extensible Software*. In: *Electrical and Computer Engineering. Canadian Conference, CCECE 2005, Saskatoon, Saskatchewan Canada. Proceedings*, S. 1324–1327, IEEE Computer Society, nur online?, Mai 2005, ISSN 0840-7789, doi:`10.1109/CCECE.2005.1557221`.

[KR04] Michael Köhler und Heiko Rölke: *Properties of Object Petri Nets*. In: J. Cortadella und W. Reisig (Herausgeber) *International Conference on Application and Theory of Petri Nets 2004*, Band 3099 von *LNCS*, S. 278–297, Springer-Verlag, 2004.

[Kru95] Philippe B. Kruchten: *The 4+1 View Model of Architecture*. IEEE Software, Band 12 (1995), Nr. 6: S. 42–50.

[Kum02a] Olaf Kummer: *Referenznetze*. Logos Verlag, Berlin, 2002.

[Kum02b] Olaf Kummer: *Referenznetze*. Disputationsvortrag, Universität Hamburg, Fachbereich Informatik, 14. Juni 2002.

[KW07] Maxim Klenski und Anatolij Willner: *Graphische Informationsmodellierung für Mulan-Agenten*. Diplomarbeit, Universität Hamburg, Department Informatik, 2007.

[KWD] Olaf Kummer, Frank Wienberg und Michael Duvigneau: *Renew – The Reference Net Workshop*. Im Internet: `http://www.renew.de/`.

[KWD01] Olaf Kummer, Frank Wienberg und Michael Duvigneau: *Renew – Architecture Guide*. University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg, release 1.5 Auflage, Mai 2001. Available at: http://www.renew.de/.

[KWD02] Olaf Kummer, Frank Wienberg und Michael Duvigneau: *Renew – The Reference Net Workshop*. Available at: http://www.renew.de/, Oktober 2002. Release 1.6.

[KWD+04] Olaf Kummer, Frank Wienberg, Michael Duvigneau, Jörn Schumacher, Michael Köhler, Daniel Moldt, Heiko Rölke und Rüdiger Valk: *An Extensible Editor and Simulation Engine for Petri Nets: Renew*. In: Jordi Cortadella und Wolfgang Reisig (Herausgeber) *Applications and Theory of Petri Nets2004: 25th International Conference, ICATPN 2004, Bologna, Italy, June 2004. Proceedings*, Band 3099 von *LNCS*, S. 484–493, Springer-Verlag, Berlin, 2004.

[KWD06a] Olaf Kummer, Frank Wienberg und Michael Duvigneau: *Renew – The Reference Net Workshop*. Available at: http://www.renew.de/, Mai 2006. Release 2.1.

[KWD06b] Olaf Kummer, Frank Wienberg und Michael Duvigneau: *Renew – User Guide*. University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg, release 2.1 Auflage, Mai 2006. Available at: http://www.renew.de/.

[Lau99] Anthony Lauder: *Pluggable Factory in Practice*. The C++ Report, Band 11 (1999), Nr. 9: S. 27–32, ISSN 1040-6042.

[LB98] Sheng Liang und Gilad Bracha: *Dynamic class loading in the Java virtual machine*. In: Freeman-Benson und Chambers [FBC98], S. 36–44, doi:10.1145/286936.286945.

[Lee08] Thorsten Leemhuis: *Höhen und Tiefen – Wo es mit 64-Bit-x86-Linux hakt*. c't Magazin für Computertechnik, (2008), Nr. 8: S. 112–115, ISSN 0724-8679.

[Lev00] John R. Levine: *Linkers and Loaders*. Morgan Kaufmann, San Francisco, Calif. [u.a.], 2000, ISBN 1-558-60496-0.

[Lip05a] Martin Lippert: *Classloading in Eclipse*. JavaSpektrum, (2005), Nr. 1. Sigs-Datacom GmbH.

[Lip05b] Martin Lippert: *Was Eclipse im Innersten zusammenhält – Unter der Haube, Teil 1: Ein Blick auf die Eclipse Runtime*. Eclipse-Magazin, Band 5 (2005): S. 19–22. Software & Support Verlag.

[LK05] Martin Lippert und Bernd Kolb: *Neustart ?!? Warum? – Mit Eclipse 3 dynamische Plug-ins implementieren*. EclipseMagazin, Band 3 (2005): S. 28–30. Software & Support Verlag.

[LM08] Martin Lippert und Klaus Marquardt: *Plugin Architectures*. Software Engineering Radio (SE-Radio), Episode 104, 20. Juli 2008. Im Internet: http://www.se-radio.net/podcast/2008-07/episode-104-plugin-architectures.

[Lon01] Andy Longshaw: *Choosing between COM+, EJB, and CCM*. In: Heineman und Councill [HC01], Kapitel 35, S. 621–640.

[LPS03] Sea Ling, Iman Poernomo und Heinz Schmidt: *Describing Web Service Architectures through Design-by-Contract*. In: Adnan Yazici und Cevat Sener (Herausgeber) *Computer and Information Sciences. ISCIS 2003, 18th International Symposium, Antalya, Turkey, November 2003. Proceedings*, Band 2869 von *LNCS*, S. 1008–1018, Springer-Verlag, Berlin, 2003, ISBN 978-3-540-20409-1, doi:10.1007/b14229.

[LS06] David E. Lightfoot und Clemens A. Szyperski (Herausgeber): *Modular Programming Languages. 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 2006. Proceedings*, Band 4228 von *LNCS*, Springer-Verlag, Berlin, 2006, ISBN 978-3-540-40927-4, doi:10.1007/11860990.

[LW07] Kung-Kiu Lau und Zheng Wang: *Software Component Models*. IEEE Transactions on Software Engineering, Band 33 (2007), Nr. 10: S. 709–724, ISSN 0098-5589, doi:10.1109/TSE.2007.70726.

[LY99] Tim Lindholm und Frank Yellin: *The Java Virtual Machine Specification*. The Java Series, Prentice Hall Professional Technical Reference (PH PTR), 2. Auflage, 1999, ISBN 0-201-43294-3. Im Internet: http://java.sun.com/docs/books/jvms/.

[M+08] Scott McCloud et al.: *Google Chrome – Behind the Open Source Browser Project*. Google, 2. September 2008. Im Internet: http://www.google.com/googlebooks/chrome/, gesichtet am 2. September 2008.

[Mag02] Boris Magnusson (Herausgeber): *Object-Oriented Programming. 16th European Conference, ECOOP 2002, Malaga, Spain, June 2002. Proceedings*, Band 2374 von *LNCS*, Springer-Verlag, Berlin, 2002, ISBN 3-540-43759-2.

[Mar06] Klaus Marquardt: *Patterns for Plugins*. In: Dragos Manolescu, Markus Voelter und James Noble (Herausgeber) *Pattern Languages of Program Design 5*, Software patterns series, Kapitel 12, S. 301–336, Addison-Wesley, Boston, New York, München u.a., 1. Auflage, 2006, ISBN 0-321-32194-4.

[McC97] Carma McClure: *Software Reuse Techniques: Adding Reuse to the System*. Prentice Hall, Upper Saddle River, NJ, 1997.

[MCC+04] Fabio Melo, Ricardo Choren, Renato Cerqueira, Carlos José Pereira de Lucena und Marcelo Blois: *Deploying Agents with the CORBA Component Model*. In: Emmerich und Wolf [EW04], S. 234–247, doi:10.1007/b98000.

[McF03] Nigel McFarlane: *Rapid Application Development with Mozilla*. Bruce Perens' Open Source Series, Prentice Hall Professional Technical Reference (PH PTR), November 2003, ISBN 978-0-13-142343-5.

[McI69] M. D. McIlroy: *Mass Produced Software Components*. In: Peter Naur und Brian Randell (Herausgeber) *Software Engineering. Garmisch, Germany, October 1968. Report on a Conference*, S. 138–155, NATO / Science Committee, NATO Scientific Affairs Division, Brussels, 1969.

[MDC] *Mozilla Developer Center*. Im Internet: http://developer.mozilla.org/.

[MDC02] *Install Wizards (aka: Stub Installers)*. Mozilla Developer Center, August 2002. Im Internet: http://developer.mozilla.org/en/docs/Install_Wizards_%28aka:_Stub_Installers%29, gesichtet mit letzter Änderung vom 26. Februar 2007, 09:59 Uhr.

[MDC05] *XUL Overlays*. Mozilla Developer Center, 2005. Im Internet: http://developer.mozilla.org/en/docs/XUL_Overlays, gesichtet mit letzter Änderung vom 20. Januar 2007, 18:08 Uhr.

[MDC06a] *Creating XPI Installer Modules*. Mozilla Developer Center, Februar 2006. Im Internet: http://developer.mozilla.org/en/docs/Creating_XPI_Installer_Modules, gesichtet mit letzter Änderung vom 28. Februar 2007, 18:04 Uhr.

[MDC06b] *Extension Packaging*. Mozilla Developer Center, April 2006. Im Internet: http://developer.mozilla.org/en/docs/Extension_Packaging, gesichtet mit letzter Änderung vom 30. März 2007, 08:43 Uhr.

[MDC06c] *Learn XPI Installer Scripting by Example*. Mozilla Developer Center, Februar 2006. Im Internet: `http://developer.mozilla.org/en/docs/Learn_XPI_Installer_Scripting_by_Example`, gesichtet mit letzter Änderung vom 2. Februar 2007, 22:00 Uhr.

[MDC06d] *Multiple Item Packaging*. Mozilla Developer Center, Mai 2006. Im Internet: `http://developer.mozilla.org/en/docs/Multiple_Item_Packaging`, gesichtet mit letzter Änderung vom 10. April 2007, 10:10 Uhr.

[MDC06e] *XPCOM ABI*. Mozilla Developer Center, März 2006. Im Internet: `http://developer.mozilla.org/en/docs/XPCOM_ABI`, gesichtet mit letzter Änderung vom 16. Juli 2006, 22:34 Uhr.

[MDC07a] *Building an Extension*. Mozilla Developer Center, Juli 2007. Im Internet: `http://developer.mozilla.org/en/docs/Building_an_Extension`, gesichtet mit letzter Änderung vom 29. Juli 2007, 12:27 Uhr.

[MDC07b] *Bundles*. Mozilla Developer Center, März 2007. Im Internet: `http://developer.mozilla.org/en/docs/Bundles`, gesichtet mit letzter Änderung vom 9. März 2007, 16:03 Uhr.

[MDC07c] *Chrome*. Mozilla Developer Center, Juli 2007. Im Internet: `http://developer.mozilla.org/en/docs/Chrome`, gesichtet mit letzter Änderung vom 22. Juli 2007, 19:10 Uhr.

[MDC07d] *Chrome Registration*. Mozilla Developer Center, Juli 2007. Im Internet: `http://developer.mozilla.org/en/docs/Chrome_Registration`, gesichtet mit letzter Änderung vom 9. Juli 2007, 07:08 Uhr.

[MDC07e] *Extension Etiquette*. Mozilla Developer Center, Mai 2007. Im Internet: `http://developer.mozilla.org/en/docs/Extension_Etiquette`, gesichtet mit letzter Änderung vom 27. Mai 2007, 11:58 Uhr.

[MDC07f] *Gecko Plugin API Reference*. Mozilla Developer Center, April 2007. Im Internet: `http://developer.mozilla.org/en/docs/Gecko_Plugin_API_Reference`, gesichtet mit letzter Änderung vom 17. April 2007, 13:13 Uhr.

[MDC07g] *Install Manifests*. Mozilla Developer Center, Juni 2007. Im Internet: `http://developer.mozilla.org/en/docs/Install_Manifests`, gesichtet mit letzter Änderung vom 28. Juni 2007, 02:43 Uhr.

[MDC07h] *Plugins: The First Install Problem*. Mozilla Developer Center, März 2007. Im Internet: `http://developer.mozilla.org/en/docs/Plugins:_The_First_Install_Problem`, gesichtet mit letzter Änderung vom 1. März 2007, 17:43 Uhr.

[MDC07i] *XUL Application Packaging*. Mozilla Developer Center, Mai 2007. Im Internet: `http://developer.mozilla.org/en/docs/XUL_Application_Packaging`, gesichtet mit letzter Änderung vom 10. July 2007, 14:33 Uhr.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach und Jeff Kramer: *Specifying Distributed Software Architectures*. In: Wilhelm Schäfer und Pere Botella (Herausgeber) *Software Engineering. 5th European Conference, ESEC '95, Sitges, Spain. Proceedings*, Band 989 von *LNCS*, S. 137–153, Springer-Verlag, Berlin, 1995, ISBN 3-540-60406-5, doi:10.1007/3-540-60406-5_12.

[MDT07] Nenad Medvidović, Eric M. Dashofy und Richard N. Taylor: *Moving Architectural Description From Under the Technology Lamppost*. Information and Software Technology, Band 49 (2007), Nr. 1: S. 12–31, ISSN 0950-5849, doi:10.1016/j.infsof.2006.08.006.

[Mey97] Bertrand Meyer: *Object-Oriented Software Construction*. Prentice Hall Professional Technical Reference (PH PTR), 2. Auflage, 1997, ISBN 0-13-629155-4.

[MGM02] Pierre Marie-Gervais und Florin Muscutariu: *Towards an ADL for Desigining Agent-Based Systems*. In: Wooldridge et al. [WWC02], S. 263–277.

[MHF06] Sean McDirmid, Wilson C. Hsieh und Matthew Flatt: *A Framework for Modular Linking in OO Languages*. In: Lightfoot und Szyperski [LS06], S. 116–135, doi:10.1007/11860990_9.

[Mit08] Klaus Mitreiter: *Einbetten der grafischen Benutzungsschnittstelle von Renew in Eclipse*. Diplomarbeit, Universität Hamburg, Department Informatik, Januar 2008.

[MKB+04] Ron Morrison, Graham Kirby, Dharini Balasubramaniam, Kath Mickan, Flavio Oquendo, Sorana Cîmpan, Brian Warboys, Bob Snowdon und R. Mark Greenwood: *Support for Evolving Software Architectures in the ArchWare ADL*. In: *Software Architecture. Fourth Working IEEE/IFIP Conference, WICSA 2004. Proceedings*, S. 69–78, Juni 2004, ISBN 0-7695-2172-X.

[MKH+07] Daniel Moldt, Fabrice Kordon, Kees van Hee, José-Manuel Colom und Rémi Bastide (Herausgeber): *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'07)*, Akademia Podlaska, Siedlce, Poland, Juni 2007.

[MMS03] Johannes Mayer, Ingo Melzer und Franz Schweiggert: *Lightweight Plug-In-Based Application Development*. In: Mehmet Aksit, Mira Mezini und Rainer Unland (Herausgeber) *Objects, Components, Architectures, Services, and Applications for a Networked World. International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 2002. Revised Papers*, Band 2591 von *LNCS*, S. 87–102, Springer-Verlag, Berlin, 2003, ISBN 978-3-540-00737-1, doi:10.1007/3-540-36557-5_9.

[MO04] Daniel Moldt und Jan Ortmann: *DaGen: A Tool for Automatic Translation from DAML-S to High-Level Petri Nets*. In: Michel Wermelinger

und Tiziana Margaria-Steffen (Herausgeber) *Fundamental Approaches to Software Engineering: 7th International Conference, FASE 2004. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings*, Band 2984 von *LNCS*, S. 209–213, Springer-Verlag, Berlin, 2004.

[MO05] Myk Melez und Ian Oeschger: *Mozilla Application Framework in Detail.* Mozilla Developer Center, 2005. Im Internet: `http://developer.mozilla.org/en/docs/Mozilla_Application_Framework_in_Detail`, gesichtet mit letzter Änderung vom 14. März 2006, 14:07 Uhr.

[Mol96] Daniel Moldt: *Höhere Petrinetze als Grundlage für Systemspezifikationen.* Dissertation, Universität Hamburg, Fachbereich Informatik, August 1996.

[Mol06] Daniel Moldt (Herausgeber): *Proceedings of the Fourth International Workshop on Modelling of Objects, Components, and Agents. MOCA'06*, Nummer FBI-HH-B-272/06 in Report of the Department of Informatics, Universität Hamburg, Department Informatik, Juni 2006.

[MOO04] Daniel Moldt, Jan Ortmann und Sven Offermann: *A Proposal for Petri Net Based Web Service Application Modeling.* In: Nora Koch, Piero Fraternali und Martin Wirsing (Herausgeber) *Web Engineering - 4th International Conference, ICWE 2004, Munich, Germany, July 26-30, 2004, Proceedings*, Band 3140 von *LNCS*, S. 93–97, Springer-Verlag, 2004, ISBN 3-540-22511-0.

[MOO05] Daniel Moldt, Sven Offermann und Jan Ortmann: *A Petri Net-Based Architecture for Web Services.* In: Lawrence Cavedon, Ryszard Kowalczyk, Zakaria Maamar, David Martin und Ingo Müller (Herausgeber) *Workshop on Service-Oriented Computing and Agent-Based Engineering, SOCABE 2005, Utrecht, Netherland, July 26, 2005. Proceedings*, S. 33–40, 2005.

[Moz02] *Plugin Identifier Specification.* Mozilla.org, Juni 2002. Im Internet: `http://www.mozilla.org/projects/plugins/plugin-identifier.html`, gesichtet mit letzter Änderung vom 18. Juni 2002 21:32 Uhr.

[MPG+00] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr und J. Fritz Barnes: *Runtime Support for Type-Safe Dynamic Java Classes.* In: Elisa Bertino (Herausgeber) *Object-Oriented Programming. 14th European Conference, ECOOP 2000, Sophia Antipolis and Cannes, France, June 2000. Proceedings*, Band 1850 von *LNCS*, S. 337–361, Springer-Verlag, Berlin, 2000, ISBN 3-540-67660-0.

[MRRR02] Nenad Medvidović, David S. Rosenblum, David F. Redmiles und Jason E. Robbins: *Modeling software architectures in the Unified Modeling Language.* ACM Transactions on Software Engineering and Methodology (TOSEM), Band 11 (2002), Nr. 1: S. 2–57, ISSN 1049-331X, doi:10.1145/504087.504088.

[MS02] Erik Meijer und Clemens Szyperski: *Overcoming Independent Extensibility Challenges.* Communications of the ACM, Band 45 (2002), Nr. 10: S. 41–44, ISSN 0001-0782, doi:10.1145/570907.570929.

[MSH02] Hamza Mazouzi, Amal El Fallah Seghrouchni und Serge Haddad: *Open protocol design for complex interactions in multi-agent systems.* In: Castelfranchi und Johnson [CJ02], S. 517–526, doi:10.1145/544862.544866.

[MT97] Nenad Medvidović und Richard N. Taylor: *A Framework for Classifying and Comparing Architecture Description Languages.* Technischer Bericht, University of California, Irvine, California, USA, 1997.

[MT00] Nenad Medvidović und Richard N. Taylor: *A Classification and Comparison Framework for Software Architecture Description Languages.* IEEE Transaction on Software Engineering, Band 26 (2000), Nr. 1: S. 70–93.

[MV03] Klaus Marquardt und Markus Völter: *Plug-Ins – applikationsspezifische Komponenten.* JavaSpektrum, (2003), Nr. 2: S. 20–25. Sigs-Datacom GmbH.

[Mør97] Anders Mørch: *Three Levels of End-User Tailoring: Customization, Integration, and Extension.* In: Morten Kyng und Lars Mathiassen (Herausgeber) *Computers and Design in Context*, S. 51–76, MIT Press, Cambridge, Mass., 1997, ISBN 0-262-11223-X.

[Mül97] Jörg P. Müller: *Control architectures for autonomous and interacting agents: A survey.* In: Lawrence Cavedon, Anand Rao und Wayne Wobcke (Herausgeber) *Intelligent Agent Systems: Theoretical and Practical Issues. Based on a Workshop Held at PRICAI '96, Cairns, Australia, August 1996*, Band 1209 von *LNCS*, S. 1–26, Springer-Verlag, Berlin, 1997, ISBN 978-3-540-62686-2, doi:10.1007/3-540-62686-7_25.

[Net] *NetBeans.* Im Internet: http://www.netbeans.org/.

[Net98] *Plug-in Guide (Communicator 4.0).* Netscape Communications Corporation, Januar 1998. Im Internet: http://developer.netscape.com/docs/manuals/communicator/plugin/index.htm, gesichtet am 13. August 2004 (web.archive.org).

[NKCG05] Xuan Thang Nguyen, Ryszard Kowalczyk, Mohan Baruwal Chhetri und Alasdair Grant: *WS2JADE: A Tool for Run-time Deployment and Control of Web Services as JADE Agent Services.* In: Unland et al. [UKC05], S. 223–251.

[O'C07] John O'Connor: *Creating Extensible Applications With the Java Platform.* Sun Developer Network (SDN), September 2007. Im Internet: http://java.sun.com/developer/technicalArticles/javase/extensible/, gesichtet am 15. Oktober 2007, 16:08 Uhr.

[Oes00] Ian Oeschger: *A XUL Bestiary*. Mozilla Developer Center, 2000. Im Internet: `http://developer.mozilla.org/en/docs/A_XUL_Bestiary`, gesichtet mit letzter Änderung vom 28. Februar 2007, 18:01 Uhr.

[Off03] Sven Offermann: *Ein Kompositionsmodell für Mehrwertdienste auf Basis von Referenznetzen*. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, nov 2003.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Greg Johnson, Nenad Medvidović, Alex Quilici, David S. Rosenblum und Alexander L. Wolf: *An architecture-based approach to self-adaptive software*. IEEE Intelligent Systems and their Applications, Band 14 (1999), Nr. 3: S. 54–62, ISSN 1094-7167, doi:10.1109/5254.769885.

[Ols07a] Dmitry Olshansky: *Core JPF Concepts*. Java Plugin Framework Website, 2007. Im Internet: `http://jpf.sourceforge.net/concepts.html`, gesichtet am 10. September 2007, 9:51 Uhr.

[Ols07b] Dmitry Olshansky: *JPF Questions And Answers*. Java Plugin Framework Website, 2007. Im Internet: `http://jpf.sourceforge.net/qa.html`, gesichtet am 10. September 2007, 10:17 Uhr.

[Ols07c] Dmitry Olshansky: *JPF System Overview*. Java Plugin Framework Website, 2007. Im Internet: `http://jpf.sourceforge.net/about.html`, gesichtet am 10. September 2007, 10:10 Uhr.

[OMG97] Object Management Group (OMG): *Mobile Agent System Interoperability Facilities Specification*. November 1997. Im Internet: `http://www.omg.org/docs/orbos/97-10-05`.

[OMG03] Object Management Group (OMG): *MDA Guide, version 1.0.1*. 2003.

[OMG05a] Object Management Group (OMG): *Unified Modeling Language (UML), Version 2.0: Infrastructure*. 2005.

[OMG05b] Object Management Group (OMG): *Unified Modeling Language (UML), Version 2.0: Superstructure*. 2005.

[OMG06] Object Management Group (OMG): *CORBA Component Model Specification. Version 4.0*. 2006.

[OMT08] Peyman Oreizy, Nenad Medvidović und Richard N. Taylor: *Runtime software adaptation: framework, approaches, and styles*. In: *Software Engineering. 30th International Conference, ICSE 2008, Leipzig, Germany. Companion*, S. 899–910, ACM, ACMPRESS.addr, 2008, ISBN 978-1-60558-079-1, doi:10.1145/1370175.1370181.

[OPB00] James Odell, H. Van Dyke Parunak und Bernhard Bauer: *Extending UML for Agents*. In: Gerd Wagner, Yves Lesperance und Eric Yu (Herausgeber) *Agent-Oriented Information Systems. Workshop AOIS 2000 at*

*the 17$^{th}$ National Conference on Artificial Intelligence (AAAI), Austin. Proceedings*, S. 3–17, Austin, TX, 2000.

[Oqu06] Flavio Oquendo: *Formally modelling software architectures with the UML 2.0 profile for π-ADL.* SIGSOFT Software Engineering Notes, Band 31 (2006), Nr. 1: S. 1–13, ISSN 0163-5948, doi:10.1145/1108768.1108773.

[Ort03] Jan Ortmann: *Prozeß-Ontologien in Multiagentensystemen – Eine prototypische Umsetzung von DAML-S Beschreibungen in Petrinetzen und ihre Verwendung in Mulan.* Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 2003.

[OSG04] OSGi Alliance: *Listeners Considered Harmful: The "Whiteboard" Pattern.* Revision 2.0 Auflage, August 2004. Im Internet: http://www.osgi.org/osgi_technology/index.asp?section=2#Further_Reading.

[OSG05] OSGi Alliance: *About the OSGi Service Platform (Technical Whitepaper).* Revision 4.1 Auflage, November 2005. Im Internet: http://www.osgi.org/osgi_technology/index.asp?section=2#Further_Reading.

[OSG06a] OSGi Alliance: *OSGi Service Platform Core Specification.* Release 4, v4.0.1 Auflage, September 2006. Im Internet: http://www.osgi.org/osgi_technology/download_specs.asp?section=2.

[OSG06b] OSGi Alliance: *OSGi Service Platform Service Compendium.* Release 4, v4.0.1 Auflage, September 2006. Im Internet: http://www.osgi.org/osgi_technology/download_specs.asp?section=2.

[Par72] David Lorge Parnas: *On the criteria to be used in decomposing systems into modules.* Communications of the ACM, Band 15 (1972), Nr. 12: S. 1053–1058, ISSN 0001-0782, doi:10.1145/361598.361623.

[Pet62] Carl Adam Petri: *Kommunikation mit Automaten.* Dissertation, Schriften des IIM 2, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, Bonn, 1962.

[Pos05] Stefan Poslad: *History of FIPA.* Im Internet: http://www.fipa.org/subgroups/ROFS-SG-docs/History-of-FIPA.htm, Dezember 2005.

[PPW02] David Poutakidis, Lin Padgham und Michael Winikoff: *Debugging Multi-Agent Systems Using Design Artifacts: The Case of Interaction Protocols.* In: Castelfranchi und Johnson [CJ02], S. 960–967, doi:10.1145/544862.544966.

[RC02] Barry Redmond und Vinny Cahill: *Supporting Unanticipated Dynamic Adaptation of Application Behaviour.* In: Magnusson [Mag02], S. 205–230.

[RDK+03] Christine Reese, Michael Duvigneau, Michael Köhler, Daniel Moldt und Heiko Rölke: *Agent–based Settler Game.* In: *Proceedings of Agentcities Agent Technology Competition (ATC03), Barcelona, Spain*, Agentcities.NET, Februar 2003.

[Riv07] Jim des Rivières: *Evolving Java-based APIs*. Eclipse Wiki, Februar 2007. Im Internet: `http://wiki.eclipse.org/index.php/Evolving_Java-based_APIs`, gesichtet mit letzter Änderung vom 15. Februar 2007, 16:47 Uhr (Version 1.1).

[Rob04] Christian Robra: *Ein komponentenbasiertes Software-Architekturmodell zur Entwicklung betrieblicher Anwendungssysteme*. Dissertation, Bamberg, Univ., 2004.

[Röl99] Heiko Rölke: *Modellierung und Implementation eines Multi-Agenten-Systems auf der Basis von Referenznetzen*. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1999.

[Röl04] Heiko Rölke: *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, Band 2 von *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.

[Röl05] Heiko Rölke: *Nette Tricks mit Netzen (in Netzen)*. In: Berndt Farwer und Daniel Moldt (Herausgeber) *Object Petri Nets, Processes, and Object Calculi*, Nummer FBI-HH-B-265/05 in Bericht des Fachbereichs Informatik, S. 71–82, Universität Hamburg, Fachbereich Informatik, August 2005.

[RPS02] Ralf Reussner, Iman Poernomo und Heinz Schmidt: *Using the Trust-ME Tool Suite for Automatic Component Protocol Adaptation*. In: Peter M. A. Sloot, Chih Jeng Kenneth Tan, Jack Dongarra und Alfons G. Hoekstra (Herausgeber) *Computational Science. International Conference, ICCS 2002, Amsterdam, The Netherlands, April 2002. Proceedings, Part II*, Band 2330 von *LNCS*, S. 854–863, Springer-Verlag, Berlin, 2002.

[RW06] Jim des Rivières und Beaton Wayne: *Eclipse Platform Technical Overview*. Whitepaper, IBM / Eclipse Foundation, April 2006. Im Internet: `http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf`.

[Sam97] Johannes Sametinger: *Software engineering with reusable components*. Springer-Verlag, Berlin, 1997, ISBN 3-540-62695-6.

[SCA] *Service Component Architecture (SCA)*. Im Internet: `http://www.osoa.org/display/Main/Service+Component+Architecture+Home`.

[SCDK07] Felix Simmendinger, Lawrence Cabac, Michael Duvigneau und Nicolas Knaak: *Controlling OSGi Bundles with Petri Nets*. In: Moldt et al. [MKH+07], S. 220–225.

[Sch03] Jörn Schumacher: *Eine Plugin-Architektur für Renew – Konzepte, Methoden, Umsetzung*. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Oktober 2003.

[Sch05] Benjamin Schleinzer: *Konzepte und Entwicklung von Plug-in Agenten*. Baccalaureatsarbeit, Universität Hamburg, Fachbereich Informatik, 2005.

[Sch07] Benjamin Schleinzer: *Flexible und hierarchische Multiagentensysteme – Modellierung und prototypische Erweiterung von Mulan und Capa*. Diplomarbeit, Universität Hamburg, Department Informatik, Dezember 2007.

[SCMD08] Benjamin Schleinzer, Lawrence Cabac, Daniel Moldt und Michael Duvigneau: *From Agents and Plugins to Plugin-Agents, Concepts for Flexible Architectures*. In: *New Technologies, Mobility and Security, 2008. International Conference, NTMS '08, Tangier, Morocco. Electronical proceedings*, S. 1–5, IEEE Xplore, 5.–7.November 2008, ISBN 978-1-42443547-0, doi:10.1109/NTMS.2008.ECP.49.

[SCS06] Bikram Sengupta, Satish Chandra und Vibha Sinha: *A research agenda for distributed software development*. In: *Software Engineering. 28th International Conference (ICSE '06), Shanghai, China. Proceedings*, S. 731–740, ACM, New York, Mai 2006, ISBN 1-59593-375-1, ISSN 0270-5257, doi:10.1145/1134285.1134402.

[SDN] *Java SE – Overview*. Sun Developer Network (SDN). Im Internet: http://java.sun.com/javase/.

[Sea69] John R. Searle: *Speech acts*. Cambridge Univ. Press, 1969.

[SG96] Mary Shaw und David Garlan: *Software architecture: perspectives on an emerging discipline*. An Alan R. Apt book, Prentice Hall, Upper Saddle River, NJ, 1996, ISBN 0-13-182957-2.

[Sha96] Mary Shaw: *Some Patterns for Software Architectures*. In: John M. Vlissides, James O. Coplien und Norman L. Kerth (Herausgeber) *Pattern languages of program design. Papers presented at Pattern Languages on Programming Conference, PLoP'95, Monticello, USA*, Software Patterns Series, Kapitel 16, S. 255–269, Addison-Wesley, Boston, New York, München u.a., 1996, ISBN 0-201-89527-7.

[Sho93] Yoav Shoham: *Agent-oriented programming*. Artificial Intelligence, Band 60 (1993), Nr. 1: S. 51–92, doi:10.1016/0004-3702(93)90034-9.

[Sim07] Felix Simmendinger: *Referenznetze zur Modellierung von wissenschaftlichen Workflows am Beispiel der Steuerung von Simulationsexperimenten*. Diplomarbeit, Universität Hamburg, Department Informatik, August 2007.

[Ste04] Johnny Stenback: *Scripting Plugins in Mozilla*. Mozilla.org, September 2004. Im Internet: http://www.mozilla.org/projects/plugins/npruntime.html, gesichtet mit letzter Änderung vom 1. Juni 2007, 16:34 Uhr.

[Sto05] Zoran Stojanović: *A Method for Component-Based and Service-Oriented Software Systems Engineering*. Dissertation, Technische Universiteit Delft, Februar 2005.

[Stö05] Harald Störrle: *UML 2 für Studenten*. Pearson Studium, München, 2005, ISBN 3-8273-7143-0.

[SW01] Judith A. Stafford und Alexander L. Wolf: *Software Architecture*. In: Heineman und Councill [HC01], Kapitel 20, S. 371–387.

[SWT] *SWT: The Standard Widget Toolkit*. The Eclipse Foundation. Im Internet: `http://www.eclipse.org/swt/`.

[Szy96] Clemens Szyperski: *Independently Extensible Systems – Software Engineering Potential and Challenges*. Computer Science. 19th Australasian Conference, Melbourne, Australia. Proceedings. Australien Computer Science Communications, Band 18 (1996), Nr. 1: S. 203–212.

[Szy98] Clemens Szyperski: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Boston, New York, München u.a., 1. Auflage, 1998, ISBN 0-201-17888-5.

[Szy02] Clemens Szyperski: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Boston, New York, München u.a., 2. Auflage, 2002, ISBN 0-201-74572-0.

[Tel05] Volker Tell: *Grundlagen für die prototypische Umsetzung eines Multiagentensystem-basierten Leitmodells*. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, März 2005.

[TGML98] M. Tuan Tu, Frank Griffel, Michael Merz und Winfried Lamersdorf: *A Plug-in Architecture Providing Dynamic Negotiation Capabilities for Mobile Agents*. In: Kurt Rothermel und Fritz Hohl (Herausgeber) *Mobile Agents. Second International Workshop, MA'98, Stuttgart, Germany. Proceedings*, Band 1477 von *LNCS*, S. 222–236, Springer-Verlag, 1998, ISBN 3-540-64959-X, doi:10.1007/BFb0057662.

[Tin05] Susan D. Tiner: *Bird's Eye View of the Mozilla Framework*. Mozilla Developer Center, 2005. Im Internet: `http://developer.mozilla.org/en/docs/Bird%27s_Eye_View_of_the_Mozilla_Framework`, gesichtet mit letzter Änderung vom 27. Juni 2007, 00:08 Uhr.

[TM05] Volker Tell und Daniel Moldt: *Ein Petrinetzsystem zur Modellierung selbstmodifizierender Petrinetze*. In: Karsten Schmidt und Christian Stahl (Herausgeber) *Proceedings of the 12th Workshop on Algorithms and Tools for Petri Nets (AWPN 05)*, S. 36–41, Humboldt Universität zu Berlin, Fachbereich Informatik, 2005.

[TMD10] Richard N. Taylor, Nenad Medvidović und Eric M. Dashofy: *Software Architecture: Foundations, Theory, and Practice*. John Wiley and Sons, Februar (2009) 2010, ISBN 978-0-470-16774-8.

[TO03] Doug Turner und Ian Oeschger: *Creating XPCOM Components*. Brownhen Publishing(?)/Mozilla.org, 2003. Im Internet: `http://www.mozilla.org/projects/xpcom/book/cxc/`, Seitenangaben beziehen sich auf die PDF-Version.

[TSL99] M. Tuan Tu, Christian Seebode und Winfried Lamersdorf: *A Dynamic Negotiation Framework for Mobile Agents*. In: *First International Symposium on Agent Systems and Applications, and Third International Symposium on Mobile Agents, Palm Springs, CA, USA. Proceedings*, S. 262–263, IEEE Computer Society, Los Alamitos, Oktober 1999, doi:10.1109/ASAMA.1999.805412.

[Tur03] Michael Turnlund: *Distributed Development – Lessons Learned*. ACM Queue, Band 1 (2003), Nr. 9: S. 26–31, ISSN 1542-7730, doi:10.1145/966789.966801.

[UKC05] Rainer Unland, Matthias Klusch und Monique Calisti (Herausgeber): *Software Agent-Based Applications, Platforms and Development Kits*. Whitestein series in software agent technologies, Birkhäuser, Basel [u.a.], 2005, ISBN 3-7643-7347-4.

[Val98] Rüdiger Valk: *Petri Nets as Token Objects - An Introduction to Elementary Object Nets*. In: Jörg Desel und Manuel Silva (Herausgeber) *19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal*, Nummer 1420 in LNCS, S. 1–25, Springer-Verlag, Berlin, 1998, ISSN 0302-9743.

[Val01] Rüdiger Valk: *Concurrency in Communicating Object Petri Nets*. In: Agha et al. [ADCR01], S. 164–195.

[Val04] Rüdiger Valk: *Object Petri Nets – Using the Nets-within-Nets Paradigm*. In: Jörg Desel, Wolfgang Reisig und Grzegorz Rozenberg (Herausgeber) *Advances in Petri Nets: Lectures on Concurrency and Petri Nets*, Band 3098 von *LNCS*, S. 819–848, Springer-Verlag, Berlin, 2004.

[VHW05] László Zsolt Varga, Ákos Hajnal und Zsolt Werner: *The WSDL2Agent Tool*. In: Unland et al. [UKC05], S. 197–222.

[Vli98] John Vlissides: *Pluggable Factory, Part 1*. The C++ Report, Band 10 (1998), Nr. 10: S. 52–56, ISSN 1040-6042.

[Vli99] John Vlissides: *Pluggable Factory, Part 2*. The C++ Report, Band 11 (1999), Nr. 2: S. 51–55, ISSN 1040-6042.

[VT05] Minh Vu und Craig Thompson: *E2 Agent Plugin Architecture*. In: Craig Thompson und Henry Hexmoor (Herausgeber) *Integration of Knowledge Intensive Multi-Agent Systems. International Conference, KIMAS 2005,*

*Waltham, MA, USA. Proceedings*, S. 26–31, IEEE Computer Society, Los Alamitos, April 2005, ISBN 0-7803-9013-X.

[W3C] *W3C Web Services Activity*. Im Internet: http://www.w3.org/2002/ws/.

[Wae04] Pakorn Waewsawangwong: *A Constraint Architectural Description Approach to Self-Organising Component-Based Software Systems*. In: *Software Engineering. 26th International Conference (ICSE '04), Edinburgh, Scotland. Proceedings*, S. 81–83, IEEE Computer Society, Los Alamitos, Mai 2004, ISBN 0-7695-2163-0, ISSN 0270-5257.

[WD01] Mark F. Wood und Scott A. DeLoach: *An Overview of the Multiagent Systems Engineering Methodology*. In: Ciancarini und Wooldridge [CW01], S. 202–222, doi:10.1007/3-540-44564-1_14.

[WDPM06] Reinhard Wolfinger, Deepak Dhungana, Herbert Prähofer und Hanspeter Mössenböck: *A Component Plug-In Architecture for the .NET Platform*. In: Lightfoot und Szyperski [LS06], S. 287–305, doi:10.1007/11860990_18.

[Wei00] Gerhard Weiß (Herausgeber): *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT Press, Cambridge, Mass., 2000.

[Wen07] Nicole Wengatz: *SCA, OSGi und Spring*. JavaSpektrum, (2007), Nr. 4: S. 41–43. Sigs-Datacom GmbH.

[Wie01] Frank Wienberg: *Informations- und prozeßorientierte Modellierung verteilter Systeme auf der Basis von Feature-Structure-Netzen*. Dissertation, Universität Hamburg, Fachbereich Informatik, Januar 2001.

[Wik07a] *NPAPI*. Wikipedia, Januar 2007. Im Internet: http://en.wikipedia.org/wiki/NPAPI?oldid=103913506, gesichtet mit letzter Änderung vom 28. Januar 2007 20:50 Uhr.

[Wik07b] *Plug-in*. Wikipedia, Juli 2007. Im Internet: http://de.wikipedia.org/wiki/Plug-in?oldid=34583141, gesichtet mit letzter Änderung vom 20. Juli 2007 08:32 Uhr.

[Wik07c] *Plugin*. Wikipedia, Juli 2007. Im Internet: http://en.wikipedia.org/wiki/Plugin?oldid=146370830, gesichtet mit letzter Änderung vom 22. Juli 2007 19:38 Uhr.

[Wik08] *Thread Safety*. Wikipedia, August 2008. Im Internet: http://en.wikipedia.org/wiki/Thread_safety?oldid=231661021, gesichtet mit letzter Änderung vom 13. August 2008 11:26 Uhr.

[WJ95] Michael Wooldridge und Nicholas R. Jennings: *Intelligent Agents: Theory and Practice*. Knowledge Engineering Review, Band 10 (1995), Nr. 2: S. 115–152.

[WJ04] Gerhard Weiß und Ralf Jakob: *Agentenorientierte Softwareentwicklung – Methoden und Tools*. Xpert.press, Springer-Verlag, Berlin, 2004, ISBN 3-540-00062-3.

[WJK00] Michael Wooldridge, Nicholas Jennings und David Kinny: *The Gaia Methodology for Agent-Oriented Analysis and Design*. The International Journal of Autonomous Agents and Multi-Agent Systems, Band 3 (2000), Nr. 3: S. 285–312.

[WS01] Rainer Weinreich und Johannes Sametinger: *Component Models and Component Services: Concepts and Principles*. In: Heineman und Councill [HC01], Kapitel 3, S. 33–48.

[WWC02] Michael J. Wooldridge, Gerhard Weiß und Paolo Ciancarini (Herausgeber): *Agent-Oriented Software Engineering II. Second International Workshop, AOSE 2001, Montreal, Canada, May 2001. Revised Papers and Invited Contributions*, Band 2222 von *LNCS*, Springer-Verlag, Berlin, 2002, ISBN 978-3-540-43282-1.

[YC04] Zhenhua Yu und Yuanli Cai: *Novel architecture description language based on high-level Petri nets*. In: *Information and Communication Technologies: From Theory to Applications. ICTTA'04, International Conference, Damascus, Syria, April 2004. Proceedings*, S. 589–590, IEEE Computer Society, Los Alamitos, 2004, doi:10.1109/ICTTA.2004.1307900.

[YC06] Zhenhua Yu und Yuanli Cai: *Object-Oriented Petri nets Based Architecture Description Language for Multi-agent Systems*. International Journal of Computer Science and Network Security (IJCSNS), Band 6 (2006), Nr. 1B: S. 123–131.

[YCWH05] Zhenhua Yu, Yuanli Cai, Ruifeng Wang und Jiuqiang Han: *$\pi$-Net ADL: An Architecture Description Language for Multi-agent Systems*. In: De-Shuang Huang, Xiao-Ping Zhang und Guang-Bin Huang (Herausgeber) *Advances in Intelligent Computing. International Conference on Intelligent Computing, ICIC 2005, Hefei, China, August 2005. Proceedings, Part II*, Band 3645 von *LNCS*, S. 218–227, Springer-Verlag, Berlin, 2005, ISBN 978-3-540-28227-3, doi:10.1007/11538356_23.

[ZCCK04] Jia Zhang, C.K. Chang, Jen-Yao Chung und S.W. Kim: *WS-Net: a Petri-Net Based Specification Model for Web Services*. In: *Web Services. IEEE International Conference, ICWS 2004, San Diego, CA, USA. Proceedings*, S. 420–427, Juli 2004, doi:10.1109/ICWS.2004.1314766.

[Zei08] Andy Zeigler: *IE8 and Loosely-Coupled IE (LCIE)*. The Windows Internet Explorer Weblog (MSDN IEBlog), 11. März 2008. Im Internet: http://blogs.msdn.com/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx, gesichtet am 3. September 2008.

[ZJW03] Franco Zambonelli, Nicholas Jennings und Michael Wooldridge: *Developing Multiagent Systems: The Gaia Methodology*. ACM Transactions on Software Engineering and Methodology, Band 12 (2003), Nr. 3: S. 317–370.

[ZO04] Franco Zambonelli und Andrea Omicini: *Challenges and Research Directions in Agent-Oriented Software Engineering*. Autonomous Agents and Multi-Agent Systems, Band 9 (2004), Nr. 3: S. 253–283, ISSN 1387-2532, doi:10.1023/B:AGNT.0000038028.66672.1e.

[Zwi05] Olaf Zwintzscher: *Software-Komponenten im Überblick: Einführung, Klassifizierung und Vergleich von JavaBeans, EJB, COM+, .Net, CORBA, UML 2*. IT kompakt, W3L-Verl., Herdecke [u.a], 2005, ISBN 3-937137-60-2.