Ein Einschubsystem für die musikwissenschaftliche Analyse

basierend auf einem Ansatz zur bruchlosen Modellierung von Anwendungsfamilien mit Rahmenwerken und Komponenten

Dissertation

zur Erlangung des Doktorgrades
am Fachbereich Informatik
der Universität Hamburg

vorgelegt von

Andreas Kornstädt
aus Bensberg

Hamburg 2002

Genehmigt vom Fachbereich Informatik der Universität Hamburg auf Antrag von

Prof. Dr. Heinz Züllighoven (Universität Hamburg)

Prof. Dr. Thomas Grey (Stanford University)

Prof. Dr. Berndt Neumann (Universität Hamburg)

Hamburg, den 19. März 2002

Prof. Dr. Siegfried Stiehl (Dekan)

Für meinen Vater,
dem ich so viel verdanke

Kurzzusammenfassung

Ausgehend von dem wiederholt in der Musikwissenschaft vorgetragenen Wunsch nach effektiver, flexibler Rechnerunterstützung bei einer Vielfalt von Analysearten, erarbeite ich in dieser Dissertation sowohl softwaretechnische als auch musikdatenverarbeitungsbezogene Lösungen, um die in der Literatur diskutierten Probleme zu überwinden. Die Ergebnisse sind (1) eine einheitliche Beschreibung der Anforderungen an eine Anwendungsfamilie von Analysesystem bezüglich der Entwicklungskontexte *Anwendungsbereich*, *Handhabung & Präsentation* und *verwendete Technik*, (2) ein Ansatz zur bruchlosen Modellierung von Anwendungsfamilien von der Analyse bis hin zur für den Anwender verständlichen sitzungsweisen Konfiguration sowie (3) zwei Konstruktionskonzepte, mit denen komplexe technische Subsysteme – beispielsweise zur Musikdatenverarbeitung – auf harmonische Weise in anwendungsorientierte Softwaresysteme eingegliedert werden können, ohne die fachliche Modellierung aufweichen zu müssen. Entsprechend der Sichtweise von Software als sozial eingebettetem System validiere ich alle drei Ergebnisse konstruktiv anhand des *JRing*-Systems zur Unterstützung der musikwissenschaftlichen Analyse, das ich nach dem Werkzeug- und Material-Ansatz WAM zusammen mit Musikwissenschaftlern bis hin zur Produktreife entwickelt habe.

Zunächst konkretisiere ich den Begriff der Anwendungsfamilie und identifiziere dabei als wesentliche Bestandteile das Kernsystem und Komponenten, mit denen das Kernsystem an festgelegten Anpassungsstellen konfigurierbar ist. Anschließend betrachte ich den Anwendungsbereich der musikwissenschaftlichen Analyse und spezifiziere ausgehend von den Gemeinsamkeiten und Unterschieden in den Anforderungen eine Anwendungsfamilie von Analysesystemen samt Kernsystem, Anpassungsstellen und Komponenten. Dabei arbeite ich heraus, dass sich die Anforderungen von Sitzung zu Sitzung ändern können und somit das Kernsystem vor jedem Start durch die Musikwissenschaftler anpassbar sein muss. Darauf aufbauend untersuche ich die softwaretechnischen Eigenschaften, die Kernsysteme und Komponenten aufweisen müssen, um auf verständliche Weise durch ihre Anwender anpassbar zu sein. Als geeignete Grundlage für eine softwaretechnische Realisierung identifiziere ich eine Kombination von Rahmenwerk- und Komponenten-Ansätzen, da reine Rahmenwerk-Ansätze sich als zu inflexibel und reine Komponenten-Ansätze sich als nicht durch Anwender handhabbar erweisen. Da ich das Konzept der Systemmetapher als zentral für die Verständlichkeit und somit der Handhabbarkeit der Auslieferungseinheiten einer Anwendungsfamilie herausarbeite, zeige ich zunächst die Probleme der in der Literatur vorgeschlagene Metaphern auf, in denen Komponenten u.a. als Legosteine, aktive Dokumente, Server oder Geräte einer Stereoanlage versinnbildlicht werden, und stelle anschließend die Systemmetapher von Einschub und Einschubrahmen vor, welche die zuvor beschriebenen Probleme nicht aufweist. Mit dem anschließend vorgestellten Einschub-Ansatz, können Anwendungsfamilien gemäß dieser Metapher konstruiert, ausgeliefert und durch die Anwender konfiguriert werden. Abschließend zeige ich, wie komplexe technische Subsysteme durch Materialien und fachliche Datentypen sowohl in Verbindung als auch unabhängig vom Einschub-Ansatz gekapselt werden können.

Abstract

Musicologists have repeatedly called for effective, flexible computer tools to facilitate the tasks that arise during various types of musicological analyses. This dissertation presents solutions in the field of software engineering as well as in the field of computer-assisted musicology to overcome the obstacles that have stood in the way of this kind of computer assistance so far. These solutions are threefold: (1) a uniform description of a comprehensive application family of analysis systems regarding the software development contexts *application domain*, *handling & presentation*, and *technology*; (2) a design approach that allows for seamless modeling of application families from analysis all the way to user-driven configuration from session to session; (3) two construction methods for integrating complex subsystems – such as existing musicology software – into user-oriented software systems without generating conflicts between user-oriented and technical design objectives. As the success of software solutions only shows in practice, I implemented the *JRing* family of music analysis systems in close cooperation with musicologists following the tools and materials approach. The *JRing* system incorporates all three solutions and attains the quality level of a commercial software product.

Starting with the basic application family concept, I identify its main constituent parts: the core system and components that can be used to configure the core system at predefined adaptation spots. I then apply this categorization to the domain of musicological analysis in order to specify an analysis system application family with (a) a core systems that represents the commonalties and (b) components that represent the variability of different kinds of analysis requirements. As I can demonstrate that all variable requirements can change from session to session, musicologists must be able to re-configure the systems completely at each startup. Starting from these premises I examine those features that core system and components need to have in order to be re-configurable in such manner: Framework-only solutions are too inflexible and components-only solutions are too difficult to be handled by users, wherefore the focus shifts to approaches that combine frameworks with components. As technical feasibility is not enough when users themselves have to do the configuration, I introduce the “system metaphor” concept to discuss various suggestions for describing the ways in which core system and components can be combined. After showing that known metaphors such as Lego blocks, active documents, servers or stereo components are problematic, the “slide-in and slide-in frame” metaphor is introduced. It avoids the previously described problems and forms the basis for the slide-in approach, making it possible to construct and to deploy user-configurable application families. Focussing on the third initial problem, I demonstrate how existing complex subsystems can be encapsulated by means of materials and domain-specific data types. These two methods can be applied in conjunction with or in isolation from the slide-in approach.

Vorwort

Ein Forschungsvorhaben kann nur in einem anregenden, diskussionsfreudigen Umfeld gelingen. Dafür, dass ich ein solches Umfeld sowohl an der Universität Hamburg als auch an der Stanford University angetroffen habe, möchte ich mich herzlich bedanken.

Mein Betreuer Prof. Dr. Heinz Züllighoven stand meinem Vorhaben von seinen ersten Anfängen im Rahmen meiner Diplomarbeit an stets aufgeschlossen gegenüber. Er hat zudem am Arbeitsbereich Softwaretechnik eine Umgebung geschaffen hat, in der aktuelle Themen in Veranstaltungen wie dem Dienstagsseminar und dem Komponentenlesekreis rasch ein Forum finden konnten. Mit dem *JWAM*-Team um Stefan Roock, Henning Wolf und Martin Lippert bestand außerdem eine Runde, in der Konstruktionsvorschläge kompetent besprochen und zügig auf ihre Tauglichkeit hin überprüft werden konnten.

Diese Arbeit wäre ohne die enthusiastische Unterstützung von Musikwissenschaftlern nicht möglich gewesen. Da sind zunächst Prof. Dr. Uwe Seifert, Prof. Dr. Constantin Floros und Prof. Dr. Peter Petersen am Musikwissenschaftlichen Institut der Universität Hamburg und Prof. Dr. Thomas Grey am Music Department in Stanford, die sich sehr viel Zeit genommen haben, um mit mir den *JRing*-Entwicklungsprozess zu betreiben. Prof. Dr. Walter Hewlett und Prof. Dr. Eleanor Selfridge-Field haben mich für anderthalb Jahre äußerst herzlich am Center for Computer Assisted Research in the Humanities (CCARH) in Stanford aufgenommen, das zu meinem zweiten wissenschaftlichen zu Hause geworden ist. Die Projekte, die ich dort parallel zu meinem Dissertationsvorhaben durchführen konnte, haben mir zahlreiche Anregungen zur Ausgestaltung *JRings* gegeben. Dr. Donald Anthony, Edmund Correia und Frances Bennion haben mir zudem ein sowohl menschlich als auch professionell überaus angenehmes Arbeitsumfeld bereitet.

Prof. Dr. David Huron von der Columbus University und Prof. Dr. Douglas Hofstadter von der University of Indiana at Bloomington standen mir als zeitgleich am CCARH eingeladene Gastwissenschaftler stets für ausführliche Diskussionen zur Verfügung, ohne die diese Arbeit in mancherlei Hinsicht unvollständig geblieben wäre. Stellvertretend für die zahlreichen ideellen Förderer möchte ich Prof. Dr. Tim Crawford vom Kings College in London nennen, der mein Projekt durch die Einladung zu mehreren Konferenzen maßgeblich vorangetrieben hat.

Schließlich hat mir das Musikverlagshaus Schott in Mainz freundlicherweise Notensatzdaten für ein komplettes Werk zur Verfügung gestellt. Ohne diese umfangreichen Daten wäre ein Musikanalysesystem nicht entwickelbar gewesen.

Ihnen allen gilt mein tief empfundener Dank.

Inhaltsverzeichnis

1 Einleitung

W a l t h e r. Wie fang' ich nach der Regel an?
S a c h s. Ihr stellt sie selbst und folgt ihr dann.

Die Meistersinger von Nürnberg, 3. Aufzug, 2. Szene

In den Naturwissenschaften sowie der technischen und wirtschaftlichen Praxis werden Computer in großem Umfang eingesetzt. Anwendern wird dort die Arbeit erleichtert, indem der routinemäßige Tätigkeitsanteil in erheblichem Ausmaß durch Rechner unterstützt wird und sich dadurch mehr Zeit für die Lösung der eigentlichen fachspezifischen Aufgaben und Probleme ergibt. Zum Teil eröffnen sich hierdurch wiederum vollkommen neue Möglichkeiten zur Lösung weitaus komplexerer Probleme, an deren Bearbeitung ohne Rechnereinsatz zuvor gar nicht zu denken war.

In den Geisteswissenschaften bietet sich ein anderes Bild. Außer der auch in anderen Feldern weit verbreiteten allgemeinen Standardsoftware wie Textverarbeitungen, Tabellenkalkulationen, Datenbanken und Statistiksoftware gibt es kaum geeignete Rechnerunterstützung, um Anwendern eine Art von Arbeitserleichterung zu bieten, wie sie in den oben erwähnten Feldern möglich ist. Obwohl der prinzipielle Weg zur Behandlung wichtiger Fragestellungen oft bekannt ist, rückt deren Beantwortung durch einen unüberwindbar hohen Anteil manueller Arbeit meist in unerreichbare Ferne.

In der Musikwissenschaft gibt es eine Reihe solcher Fragestellungen im Bereich der Werkanalyse. Viele der zum Teil seit mehr als hundert Jahren vorliegenden Kompositionen sind in ihrem elementaren Grundaufbau (je nach Epoche z.B. Themen, Leitmotive oder Zwölftonreihen) bis auf den heutigen Tag in ihrer Gesamtheit nur äußerst spärlich durchdrungen, so dass bisherige Untersuchungen in ihrem Umfang oder ihrer Aussagekraft starken Einschränkungen unterworfen sind. Der Wunsch nach Verbesserung dieser Situation wird zwar wiederholt geäußert, seine Realisierung mit Mitteln der herkömmlichen manuellen Analyseweise aber als „enormous task"[Coo68] angesehen. Seit einigen Jahren werden auch vereinzelt Forderungen nach geeigneter Rechnerunterstützung für bestimmte Analyseaufgaben laut, (z.B. [Cas94] für die Set Theory), jedoch gibt es trotz beträchtlicher Ähnlichkeiten großer Teile der Analysetätigkeit weder Standardsoftware noch Individuallösungen.

Ausschlaggebend für diese Situation sind die großen Unterschiede der theoretischen Hintergründe der einzelnen Analysearten, der Partiturarten und der zur Verfügung stehenden Rechner. Da sich alle drei Faktoren unabhängig voneinander bedingt durch wechselnde Analysegegenstände und Arbeitsorte innerhalb heterogen ausgestatteter Institute sogar für ein und denselben Musikwissenschaftler von Sitzung zu Sitzung ändern können, muss ein brauchbares Analysesystem sitzungsweise an die individuellen Anforderungen des Analysators anpassbar sein.

Für die Informatik ergeben sich hier Herausforderungen in den Bereichen *Musikdatenverarbeitung* (MDV) und *Softwaretechnik*.

In der Musikdatenverarbeitung muss zunächst

- eine geeignete Infrastruktur aus Musikdatenformaten und Verarbeitungstechniken konzipiert und realisiert werden, die den Anforderungen einer graphisch interaktiven Analyse Rechnung trägt. Andernfalls bleiben alle weitergehenden Pläne mangels verfügbarer computerverarbeitbarer Partiturdaten pure Theorie.

In der Softwaretechnik gilt es,

- Ansätze zu finden, um einander ähnliche Anwendungssysteme so zu modellieren und zu konstruieren, dass die Anwender selbst sie sitzungsweise an ihre augenblicklichen Anforderungen anpassen können, anstatt für jede Kombination von Anforderungen separate Systeme auszuliefern;

- die für die üblichen naturwissenschaftlich-wirtschaftlichen Anwendungsgebiete konzipierten Softwareentwicklungsmethoden an ein musikwissenschaftliches Umfeld zu adaptieren;

- die speziellen Techniken der Musikdatenverarbeitung entsprechend dem Leitbild des verwendeten softwaretechnischen Ansatzes in das Anwendungssystem einzubetten.

Im Bereich der Musikdatenverarbeitung und der Adaption existierender Softwareentwicklungsmethoden konnte ich bereits im Rahmen meiner Diplomarbeit einige Lösungen aufzeigen. Aufgrund der Einschränkung auf den Spezialfall der leitmotivischen Analyse und eines nur bis ins Stadium eines Handhabungsprototyps entwickelten Systems, mussten die Fragen nach softwaretechnischer Modellierung von einander ähnlichen Anwendungssystemen sowie nach Einbettung musikdatenverarbeitungsspezifischer Techniken unberücksichtigt bleiben. Die Konzeption und konstruktive Umsetzung von Lösungen für diese Fragen ist Gegenstand dieser Dissertation.

1.1 Die untersuchten Fragestellungen

Mengen von Anwendungssystemen, die sich in bestimmten Teilen ähnlich sind und in anderen unterscheiden, werden in der Softwaretechnik seit langem als Produkt- oder Anwendungsfamilien klassifiziert (vgl. [Par76]) Zur ihrer Modellierung werden in der Literatur je nach Phase des Entwicklungsprozesses verschiedene partielle Ansätze vorgeschlagen:

(1) Spezielle *commonality/variability*-Analyseverfahren, um die Ähnlichkeiten und Unterschiedlichkeiten des *fachlichen Modells des betrachteten Anwendungsbereichs* zu ermitteln (vgl. [CHW98], [CW98] und [GJKW97]),

(2) objektorientierte Anwendungsrahmenwerke, um denjenigen Teil des *softwaretechnischen Modells des Anwendungssystems* zu erstellen, der die Gemeinsamkeiten verkörpert (vgl. [GHJV98], [Pre97]), und

(3) Komponenten als die die Unterschiedlichkeiten verkörpernden, rekombinierbaren *Auslieferungseinheiten*, aus denen konkrete Anwendungssysteme zusammengesetzt werden (vgl. [Gri98] und [Szy98]).

Dadurch, dass die Modelle bzw. Auslieferungseinheiten einmal unter fachlichen, dann softwaretechnischen und schließlich technologischen Gesichtspunkten strukturiert und partitioniert werden, ergeben sich die folgenden Probleme:

- Es treten Strukturbrüche zwischen den Modellen bzw. den Modellen und den Auslieferungseinheiten auf, die das Verständnis der Anwendungsfamilie erschweren, so dass sie nur mit hohem Aufwand erstell- und weiterentwickelbar sind.

- Wenn die Struktur und Handhabbarkeit der Komponenten nicht für die Anwender verständlich ist, sind sie nicht in der Lage, die notwendigen sitzungsweisen Anpassungen eigenverantwortlich durchzuführen, sondern sind stets auf die Hilfe der Entwickler oder von Systemadministratoren angewiesen.

Obwohl es seit einiger Zeit Vorschläge für die Vermeidung von Strukturbrüchen zwischen dem fachlichen und dem softwaretechnischen Modell in Form von anwendungsorientierten Ansätzen gibt (vgl. [Bäu98] und [WAM98]), ist es weiterhin problematisch, Rahmenwerke und Komponenten in einem gemeinsamen softwaretechnischen Modell zu vereinen.

Des Weiteren werden zunehmend Schwierigkeiten registriert, wenn Anwendungsfamilien alleine mit Anwendungsrahmenwerken oder ausschließlich mit Komponenten realisiert werden:

- Als Anwendungsrahmenwerke in einem Stück ausgelieferte Anwendungsfamilien werden zu sogenannten „Monolithen“ oder „fatware“[Szy98, S. 126], da sie alle Anpassungsmöglichkeiten bereits bei der Auslieferung beinhalten müssen, auch wenn der individuelle Anwender nur einen Bruchteil davon benötigt. Andererseits ist es nicht möglich, zusätzliche Anpassungsmöglichkeiten hinzuzufügen, ohne das gesamte System erneut auszuliefern.

- Unabhängig voneinander entwickelte Komponenten lassen sich aufgrund verschiedener Schnittstellen nicht sicher zu Anwendungssystemen zusammenfügen. Besonders problematisch ist hierbei die Frage, welche Komponente die Steuerung des Kontrollflusses übernimmt. Da Komponenten technologisch motivierte, sprachunabhängige Einheiten sind, ist die Kombination mit objektorientierten Rahmenwerken, die die Kontrollflusssteuerung übernehmen könnten, ein „largely unresolved problem“[Szy98, S. 13].

Das Anliegen dieser Arbeit ist es daher, einen neuen, einheitlichen Modellierungsansatz zu beschreiben, mit dem sich Anwendungsfamilien über den gesamten Softwareentwicklungsprozess hinweg bruchlos strukturieren und konstruieren lassen. Auf diese Weise lassen sich monolithische Rahmenwerke in Komponenten zerlegen, so dass neben dem Kernsystem nur diejenigen Teile an die Anwender ausgeliefert werden müssen, die diese auch tatsächlich benötigen. Benötigen die Anwender zu einem späteren Zeit-

punkt weitere Anpassungen, so müssen sie lediglich die dazu notwendigen Komponenten beziehen und sie auf für sie verständliche Weise zu einem lauffähigen Anwendungssystem zusammenfügen.

Im Einzelnen untersuche ich die folgenden Fragestellungen:

- Welche Eigenschaften muss ein durch seine Anwender sitzungsweise zusammenstellbares Anwendungssystem aufweisen? Inwiefern sind die in der Literatur diskutierten Ansätze geeignet, um ein solches System zu modellieren?

- Wie lassen sich Rahmenwerk- und Komponenten-Konzepte dergestalt aufeinander beziehen, dass Strukturbrüche zwischen dem softwaretechnischen Modell und den Auslieferungseinheiten vermieden werden können?

- Auf welche Weise lassen sich mit dem vorgeschlagenen Ansatz existierende MDV-Systeme kapseln, so dass sie einerseits maximal genutzt werden und andererseits entbehrlich sind, falls sie am jeweiligen Arbeitsplatzrechner nicht zur Verfügung stehen?

Da der beschriebene Ansatz am Entwicklungsprozess einer konkreten Anwendungsfamilie zur Unterstützung der musikwissenschaftlichen Analyse konstruktiv validiert wird, für die es kein Vorgängersystem gibt, werden anfangs auch die folgenden Fragen behandelt:

- Welches sind die speziellen Anforderungen, die ein für die musikwissenschaftliche Analyse geeignetes Anwendungssystem erfüllen muss?

- Lassen sich Anwendungssysteme für unterschiedliche Arten der musikwissenschaftlichen Analyse als Anwendungsfamilien auffassen?

Indem ich diese Fragen beantworte, erschließe ich in Ergänzung zu dem oben skizzierten, neuartigen Ansatz zur Modellierung von Anwendungsfamilien mit den folgenden Ergebnissen wissenschaftliches Neuland:

- Die kombinierte Darstellung der fachlichen und softwaretechnischen Anforderungen an musikwissenschaftliche Analysesysteme und des sich daraus ergebenden Gestaltungsspielraums ist die erste ihrer Art und als Grundlage sowohl für weitere Forschung als auch konkrete Programme geeignet.

- Mit der von mir entwickelten Technik zur Modellierung von Anwendungsgegenständen wie Partituren, Faksimiles und Stadtplänen, die sich nur mit Hilfe von Menschenhand geschaffener Layoutdaten ansprechend darstellen lassen, ergeben sich weniger Kapselungsprobleme als bisher.

- Für die „Fachwert“ genannten fachlichen Datentypen einer Anwendung (vgl. [WAM98, S. 316]), die besonders komplexe Operationen oder einen sehr großen, aber endlichen Wertebereich aufweisen, beschreibe ich eine bisher nicht bekannte Technik, mit der es möglich ist, diese auf schlanke Weise zu realisieren, ohne dabei die Funktionalität auf deren Klienten zu verlagern.

1.2 Einordnung der Arbeit

Mein Beitrag zur Diskussion über die bruchlose Modellierung von Anwendungsfamilien ist geprägt durch meine Forschungsaufenthalte am Arbeitsbereich Softwaretechnik der Universität Hamburg sowie am Center for Computer Assisted Research in the Humanities (CCARH) an der Stanford University. In Hamburg entwickelte ich zunächst aufbauend auf den Grundlagen meiner Diplomarbeit[Kor96a] das *JRing*-System zur Unterstützung der musikwissenschaftlichen Analyse, partizipierte dabei als Mitglied des *JWAM*-Teams an der Konsolidierung des gleichnamigen Rahmenwerks[WPS01] und leitete gemeinsam mit Henning Wolf den dortigen Komponentenlesekreis. Auf der Anwendungsseite beeinflussten mich neben meiner Zusammenarbeit mit Musikwissenschaftlern in Hamburg und in Stanford vor allem meine Mitarbeit in Projekten am CCARH. Dort entwickelte ich das Musikdatenkonvertierungssystem *scr2hmd* weiter [Kor96b] und konzipierte sowie realisierte die auf Anforderungen von Musikwissenschaftlern zugeschnittenen Web-basierten Dienste *Themefinder*[Kor98a] und *MuseData-Search*[Kor98b].

Entscheidenden Einfluss auf meine Sichtweise des objektorientierten Entwicklungsprozesses hat der am Arbeitsbereich Softwaretechnik entwickelte Werkzeug & Material-Ansatz WAM (vgl. [BZ90], [GKZ94], [Gry96] und [WAM98]) und das ihm zugrundeliegende Leitbild vom Arbeitsplatz für eigenverantwortliche Expertentätigkeit, zu dessen Formulierung ich beigetragen habe. Der WAM-Ansatz unterstützt den Entwicklungsprozess durch für Anwender und Entwickler gleichermaßen verständliche Analysedokumente, zueinander passende Konzepte und Konstruktionstechniken sowie durch eine evolutionäre Vorgehensweise, durch die auch der Ansatz selbst an die konkreten Anforderungen des jeweiligen Projekts angepasst werden kann.

Zentral für diese Arbeit ist die im WAM-Ansatz zum Tragen kommende Auffassung von Anwendungsorientierung, unter der ich das Folgende verstehe:

> „Ein Programmsystem heißt anwendungsorientiert, wenn Folgendes gilt:
>
> - Die Funktionalität des Systems orientiert sich an den Aufgaben aus dem Anwendungsbereich.
> - Die Handhabung des Systems ist benutzergerecht.
> - Die im System festgelegten Abläufe und Schritte lassen sich je nach Anwendungssituation problemlos an die tatsächlichen Anforderungen anpassen.“[WAM98, S. 4]

Ein wesentlicher Beitrag dieser Dissertation ist in diesem Zusammenhang die Ausdehnung des dritten Punktes von der Anpassung während der Entwicklung durch die Entwickler auf die Anpassung von Sitzung zu Sitzung durch die Anwender.

Im Zuge meiner Argumentation gehe ich wie Bäumer (vgl. [Bäu98] und [WAM98, S. 142ff]) davon aus, dass der Softwareentwicklungsprozess neben dem primären Kontext *Anwendungsbereich* auch von den Kontexten *Handhabung & Präsentation* (mit Leitbild und Entwurfsmetaphern) und *verwendete Technik* beeinflusst wird.

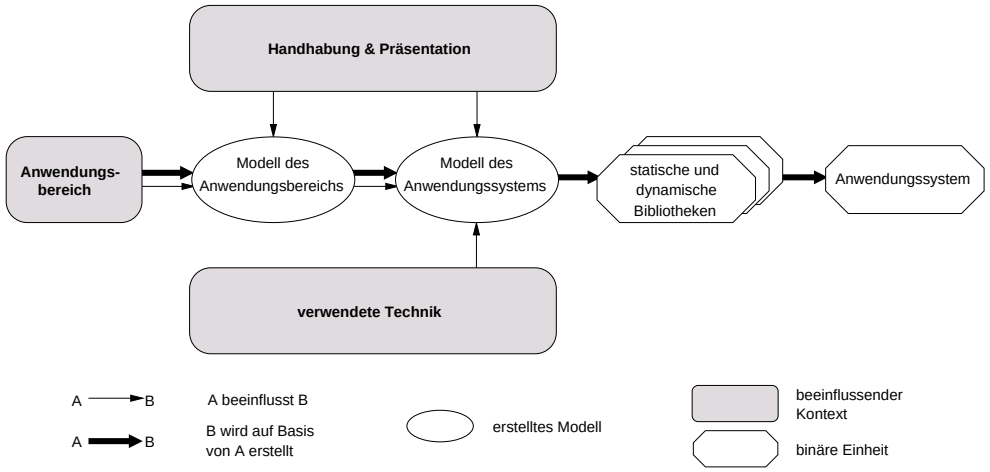


Abb. 1-1 Kontexte, Modelle und binäre Einheiten im objektorientierten Softwareentwicklungsprozess
Darstellung nach [Bäu98, S. 34] und [WAM98, S. 159]

Auf der Grundlage des Anwendungsbereichs wird zunächst – beeinflusst durch das gewählte Leitbild – ein fachliches Modell des Anwendungsbereichs erstellt. Dieses bildet die Ausgangsbasis für ein Modell des Anwendungssystems, das sowohl unter Berücksichtigung des Leitbilds als auch der verwendeten Technik ergänzt wird. Eine Partitur aus dem Anwendungsbereich der musikwissenschaftlichen Analyse wird z.B. als relevantes Material zunächst in seinen wesentlichen Aspekten ins fachliche Modell aufgenommen und dann im Modell des Anwendungssystems als Materialklasse `Partitur` realisiert. Da die rein fachlich motivierten Operationen wie `gibName` und `loescheMotivMarkierung` nicht für ein lauffähiges Anwendungssystem ausreichen, muss u.a. für die Darstellung auf einem Desktop `gibIcon` (Kontext *Handhabung & Präsentation*) und für die Anbindung an ein MDV-System `gibDatenformat` hinzugefügt werden (Kontext *verwendete Technik*). Aufgrund dieses ergänzten Modells können mit Hilfe von Compilern binäre Einheiten erstellt und schließlich zu einem ausführbaren Anwendungssystem zusammengestellt werden. Um die Verständlichkeit der Modelle und damit die (Weiter-)Entwicklung des Anwendungssystems zu erleichtern ist es dabei vorteilhaft, wenn die Modelle in ihrer Struktur der Struktur der beeinflussenden Kontexten ähnlich sind (vgl. [Bäu98, S. 58]).

Bei den zu untersuchenden Fragestellungen habe ich aufgeführt, dass ich dem Problem nachgehen werde, wie Rahmenwerke und Komponenten kombiniert werden können, um durch die Anwender sitzungsweise anpassbare Anwendungssysteme bruchlos zu modellieren. Dabei verwende ich den Rahmenwerkbegriff zunächst im Sinne Bäumers, der wiederum die von Gamma et al, Johnson, Foote und Russo geprägten Begriffe fortschreibt (vgl. [GHJV98], [JF88], [JR91] und [Joh92]).

Begriff 1.1 Rahmenwerk *(framework)* 1. Fassung

Ein Rahmenwerk stellt eine softwaretechnische Lösung für eine Reihe ähnlicher Probleme zur Verfügung. Statisch besteht es aus einer Menge von Klassenhierarchien, die, bezogen auf das zu lösende Problem, das Zusammenspiel (Kontrollfluss) einer Gruppe von Objekten zur Laufzeit eines Softwaresystems festlegen.[Bäu98, S. 93]

Obwohl Komponenten aufgrund der zunehmenden Verwendung von *CORBA*, *COM* und *JavaBeans* zur Zeit stark diskutiert werden, ist noch nicht eindeutig geklärt, welches Komponentenkonzept mit diesem Begriff in Verbindung gebracht wird. Bei meinen Vorschlägen für klar abgegrenzte Konzepte und dementsprechende Begriffe berücksichtige ich zwar alle repräsentativen Konzepte aus der Literatur, fokussiere dann aber in Anbetracht der Tatsache, dass Anwender selbst die Anpassungen eines Anwendungssystems vornehmen sollen, rasch auf binäre Komponenten im Sinne Szyperskis.

Begriff 1.2 Komponente *(component)* 1. Fassung

Komponenten sind in Binärform vorliegende Lösungen softwaretechnischer Probleme, die so zusammengefügt werden können, dass sie ein lauffähiges Softwaresystem bilden. Sie können unabhängig voneinander hergestellt, erworben und eingesetzt werden. Dazu besitzen Komponenten eine feste Schnittstelle und enthalten potentiell auch eigene Ressourcen, auf die sie bei der Ausführung zurückgreifen.(vgl. [Szy98, S. viii, S. 34 und S. 276])

Bei den untersuchten Arten der musikwissenschaftlichen Analyse beschränke ich mich ausschließlich auf diejenigen Analyseverfahren der klassischen Musikwissenschaft und der Music Theory, die unmittelbar von gedruckten Partituren ausgehen. Keine Berücksichtigung finden Ansätze, bei denen die Untersuchung von Akustikdaten im Vordergrund steht.

Aus der Wahl des Leitbilds des Arbeitsplatzes für eigenverantwortliche Expertentätigkeit folgt, dass ich die Rolle des Computers als Werkzeug auffasse, das die Entscheidung darüber, was ein Thema, Leitmotiv oder Set ist, dem Musikwissenschaftler überlässt, der sich die Partitur Strukturelement für Strukturelement erschließt. Dementsprechend berücksichtige ich keine Analyseverfahren, bei denen Rechner als Theorieprüfungsautomaten fungieren, die gesamte Werke oder große Werkabschnitte gemäß einer formalisierten Theorie eigenständig segmentieren, untersuchen und den Anwendern am Ende ein Gesamtergebnis liefern (vgl. [Alp68] und [MZ94]).

1.3 Methodik

Die von mir gewählte Methodik resultiert aus meiner am Arbeitsbereich Softwaretechnik der Universität Hamburg geprägten Sichtweise von Software als sozial eingebettetem System im Sinne Lehmans (vgl. [Leh80] und [FZ99, S. 644]). Auf die Vorgehensweise wirkt sich dies in zweierlei Hinsicht aus:

(1) Da Software nur vor dem Hintergrund des konkreten Anwendungsbereichs verständlich ist, muss bei der Analyse der fachlichen Problemstellung ein hohes Maß an fachlichem Wissen aufgebaut werden.

(2) Eine Fragestellung bezüglich sozial eingebetteter Software kann erst dann als gelöst gelten, wenn die Anwender die Software im Sinne einer konstruktiven Qualitätssicherung akzeptieren und im Rahmen ihrer Arbeit einsetzen (vgl. [FZ99, S. 663]).

Demzufolge stellt sich das dieser Arbeit zugrundeliegende methodische Vorgehen wie folgt dar:

1. In einem ersten Schritt habe ich die fachlichen und die softwaretechnischen Probleme herauspräpariert. Im Bereich der Musikwissenschaften habe ich mich dazu intensiv in die Musiktheorie eingearbeitet und mich in längeren Besuchen an musikwissenschaftlichen Instituten mit der dortigen Arbeitspraxis vertraut gemacht.

2. Ausgehend von den Problemstellungen habe ich daraufhin den konzeptionellen Forschungsstand sowohl in der Softwaretechnik als auch der Musikinformatik erarbeitet und geprüft, inwiefern er Ansätze bietet, um die betrachteten Probleme zu lösen.

3. Da eine Lösung auf der Grundlage des aktuellen Forschungsstandes in vielerlei Hinsicht unzureichend geblieben wäre, habe ich eigene softwaretechnische Konzepte entwickelt und auf ihre Tragfähigkeit hin überprüft.

4. Schließlich habe ich meine Lösung konstruktiv validiert, indem ich das Analysesystem *JRing* in Zusammenarbeit mit Musikwissenschaftlern bis hin zur Produktreife implementiert und meine Lösungen in Vorträgen und Artikeln der musikwissenschaftlichen Fachwelt vorgestellt habe (vgl. [Kor96b], [Kor97a], [Kor97b], [Kor00], [Kor01a] und [Kor01b]). Aufgrund des positiven Echos fanden zum Zeitpunkt der Einreichung dieser Arbeit Erwägungen zur Markteinführung *JRings* durch die Firma Apcon WPS statt.

Der weitere Aufbau der Arbeit ergibt sich aus den untersuchten Fragestellungen, der Einordnung und der eben geschilderten Methodik.

1.4 Aufbau der Arbeit

Zu Beginn erörtere ich in Kapitel 2 auf der Grundlage softwaretechnischer Literatur das Konzept der Anwendungsfamilie und identifiziere die Auslieferungseinheiten, aus denen konkrete Anwendungssysteme zusammengestellt werden können. Im Gegensatz zu anderen Autoren fokussiere ich hierbei nicht primär auf die Wiederverwendung bereits existierender, aber in einem anderen Zusammenhang verwendeter Komponenten, die durch Scripting-ähnliche Programmierung miteinander zu einer neuen Anwendung verbunden werden (vgl. [Bäu98], [CHW98] und [Szy98]), sondern auf eine Kombination von einem Kernsystem und dazu passenden Komponenten, wobei das Kernsystem bereits sämtliche Gemeinsamkeiten verkörpert und die Komponenten dazu dienen, Varianten des in seinem Gegenstand stets gleichen Anwendungssystems zu erzeugen. Insbesondere betrachte ich dabei die Verbindungsstellen zwischen Kernsystem und Komponenten und entwickle dabei das Konzept der Anpassungsstelle, mit dem sich voneinander unabhängige Arten der Anpassung beschreiben lassen.

Im Rahmen von Kapitel 3 diskutiere ich auf der Grundlage einer nach WAM durchgeführten Analyse die Eigenschaften, die Anwendungssysteme zur Unterstützung der mu-

sikwissenschaftlichen Analyse aufweisen müssen und zeige dabei, dass sie sich trotzt der in (1) ihrem theoretischen Hintergrund, (2) ihrer verwendeten Partiturarten und (3) ihren plattformabhängigen Anforderungen unterschiedlichen Eigenschaften als eine Anwendungsfamilie auffassen lassen. Zugleich arbeite ich die softwaretechnischen Probleme heraus, die sich daraus ergeben, dass die Funktionalität existierender MDV-Systeme maximal genutzt werden soll, ohne dass jedoch dabei eine solche Abhängigkeit entsteht, dass das Analysesystem auf Rechnern, auf denen ein solches System nicht zur Verfügung steht, unbrauchbar wird. Am Ende des Kapitels stelle ich das die Gemeinsamkeiten verkörpernde *JRing*-Kernsystem vor, das ich in den folgenden Diskussionen als Primärbeispiel verwende.

So zeige ich in Kapitel 4 zunächst basierend auf Vorschlägen aus der Literatur an diesem Beispiel, wie sich das Kernsystem einer Anwendungsfamilie mit Hilfe eines Black-Box-Anwendungsrahmenwerks modellieren lässt und arbeite die Nachteile der damit verbundenen monolithischen Auslieferung heraus. Um Anwendungsfamilien unter Umgehung dieser Nachteile getrennt als Kernsystem und Komponenten ausliefern zu können, untersuche ich deren Verhältnis zueinander nun aus softwaretechnischer Sicht und konstatiere, dass beide sowohl ein incoming interface als auch ein outgoing interface benötigen, um schlanken Komponenten Zugriff auf ausgewählte Dienste des Kernsystems zu gewähren. Außerdem zeige ich, dass Anwender die Komponenten nur dann sinnvoll zur Anpassung handhaben können, wenn diese ebenso wie die Gegenstände und Tätigkeiten des Anwendungsbereichs zum gewählten Leitbild passen und eine fachlich motivierte Funktionalität verkörpern. Für die in der Literatur diskutierten Ansätze, die Komponenten als Server (vgl. [MD95], [Rog97], [FZ99] und [OMG97a]), Einsteck-Erweiterungen (engl. *plug-in extensions, plug-ins*; vgl. [Net98a] und [Tur99]) oder Dokumente zu modellieren (vgl. [Nel95] und [Obe97]), zeige ich, dass diese die genannten Voraussetzungen nicht oder nur eingeschränkt erfüllen und mit ihnen eine verständliche, sichere Erzeugung von Mitgliedern von Anwendungsfamilien durch die Anwender nicht möglich ist.
Im Zuge der Argumentation arbeite ich zudem die damit notwendigen, aber in der Literatur nicht explizit diskutierten Unterschiede zwischen Abstraktions-Black-Boxes und Schutz-Black-Boxes sowie zwischen Konstruktions-, Implementations- und Laufzeitkomponenten heraus und bringe sie begrifflich auf den Punkt.

In Kapitel 5 entwickle ich die zum WAM-Leitbild des Arbeitsplatzes für eigenverantwortliche Expertentätigkeit passende Systemmetapher des „Einschub und Einschubrahmen“, mit der sowohl die Komponenten selbst als auch die Anpassungsstellen des Kernsystems für Anwender und Entwickler gleichermaßen anschaulich vergegenständlicht werden können. Darauf aufbauend zeige ich zunächst, wie als Black-Box-Anwendungsrahmenwerke modellierte Kernsysteme modifiziert werden müssen, um sie in metapherkonforme Einschubsysteme umzuwandeln, und demonstriere dann, wie sich ein solches System in Java auf der Basis von *JWAM* konstruktiv umsetzen lässt. Dazu entwickle ich sowohl einen Konfigurations-Desktop für die interaktive sitzungsweise Anpassung durch die Anwender als auch einen Einschubmanager, der Einschübe wunschgemäß in das laufende Anwendungssystem eingliedert.

Im Rahmen des 6. Kapitels wende ich mich vor dem Hintergrund dieser Ergebnisse den offenen Fragen der fachlichen und MDV-bezogenen Modellierung *JRings* zu und diskutiere die Realisierung der komplexen Materialien Partitur und Notizkatalog von der Warte der Handhabung und Präsentation her. Darauf aufbauend greife ich das in Kapitel 3 herausgearbeitete Problem bei deren softwaretechnischer Umsetzung unter Rückgriff auf ein MDV-Subsystem auf und entwickle die Konzepte des intern getrennten Materials sowie des schlanken komplexen Fachwerts. Mit ihrer Hilfe führt das Fehlen eines als Einschub modellierten MDV-Subsystems nicht zu einem Totalausfall des Anwendungssystems, sondern lediglich zu einer erträglichen Einschränkung der Funktionalität. Unbeschadet des Nutzens dieser Konzepte im Rahmen *JRings* sind beide generell dazu geeignet, Materialien mit nicht automatisch generierbaren Präsentationsdaten bzw. Fachwerte mit komplexen Operationen mit weniger Problemen als herkömmliche Techniken zu modellieren.

Abschließend werden die Ergebnisse dieser Arbeit in Kapitel 7 zusammengefasst und Ausgangspunkte für weitere Forschungsarbeiten aufgezeigt.

2 Anwendungsfamilien

Die Idee, ähnliche Anwendungssysteme nicht einzeln, sondern als Mitglieder einer Anwendungsfamilie zu betrachten, geht auf David Parnas' allgemein gefassten Vorschlag des Konzepts der Programmfamilie zurück:

> „Program families are defined (analogous to hardware families) as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analysing individual members."[Par76, S.1]

Parnas' Definition ist rein ökonomisch und enthält keinerlei Aussagen darüber, in welcher Hinsicht sich die Familienmitglieder ähneln und ob der Familienzusammenhang erst nach der individuellen Herstellung der Programme konstatiert oder von vorneherein bei der Entwicklung berücksichtigt wird. Obgleich Parnas im weiteren Verlauf des Artikels durch die Wahl seiner Beispiele den Schluss nahelegt, dass er sich primär auf eine von Anfang an geplante Familienstruktur bezieht, nimmt er keine Präzisierung des Begriffs vor, die andere Auffassungen ausschließt.

Unter dem Begriff der Programm- oder Anwendungsfamilie werden daher zur Zeit in der Literatur vier verschiedene Konzepte diskutiert, die sich darin unterscheiden, worauf sie die von Parnas erwähnten Gemeinsamkeiten beziehen:

(1) **Gemeinsamkeiten funktionaler Eigenschaften von Produkten verschiedener Hersteller** (vgl. [CW98]). Die Mitglieder werden vollkommen getrennt voneinander entwickelt und teilen dementsprechend keinen Teil ihrer Implementationen miteinander. Die Gemeinsamkeiten bestehen alleine darin, dass alle Anwendungen für den gleichen Anwendungsbereich konzipiert sind und ähnliche Mittel zur Verfügung stellen, um die Aufgaben der Anwender zu unterstützen. Beispiele finden sich unter Bildbearbeitungsprogrammen (*Corel Photopaint*, *Adobe Photoshop* und *GIMP*[1]) und Web-Browsern (*Netscape Navigator*, *Microsoft Internet Explorer*, *Opera*).

(2) **Gemeinsamkeiten verschiedener Entwicklungsstufen eines Produkts eines Herstellers** (vgl. [OS96] und [CG96]). Die verschiedenen Mitglieder lassen sich chronologisch anordnen und werden aufeinander aufbauend entwickelt. Wie bei Anwendungsarten besteht dabei auf jeden Fall ein gemeinsamer Anwendungsbereich. Neuere Anwendungen nutzen große Teile des Entwurfs und in den meisten Fällen auch der Implementation der unmittelbaren Vorgängeranwendung, sofern es sich nicht um eine komplette Neuimplementation wie z.B. den *Netscape Navigator 6* handelt.

(3) **Gemeinsamkeiten bei der Nutzung von Komponenten durch verschiedene, aufeinander abgestimmte Produkte** (vgl. [DNF96] und [Bäu98, S. 78]). Obwohl der Anwendungsbereich der Mitglieder ähnlich sein kann, ist die entscheidende Gemeinsamkeit die Nutzung identischer implementierter Einheiten. Im weitesten Sinne

[1] Das Akronym steht für Gnu Image Manipulation Program

kann es sich dabei um alle für ein bestimmtes Betriebssystem entwickelten Anwendungen handeln, die dieselben Betriebssystemaufrufe verwenden, um z.B. über Pipes, gemeinsame Speicherbereiche, Nachrichten oder Prozedurfernaufrufe miteinander zu kommunizieren. Im engeren Sinne sind die Anwendungsbereiche verwandt aber nicht identisch und beinhalten zum Teil die gleichen Arbeitsgegenstände, die allerdings von den einzelnen Mitgliedern in unterschiedlicher Weise genutzt werden.

Ein Beispiel aus dem Bereich der Internetanwendungen ist der *Netscape Communicator*, der u.a. den Web-Browser *Navigator*, den HTML-Editor *Composer*, den Mail- und News-Client *Messenger* und den Zeitkoordinator *Calender* umfasst. Neben den überall gleichen Komponenten zur Vorbereitung des Ausdrucks, nutzen der *Navigator* und der *Messenger* dieselbe sogenannte „HTML-Engine“ zur Interpretation und Darstellung von HTML-Dokumenten und sowohl der *Composer* als auch der *Messenger* greifen auf dieselbe Komponente zurück, um Hypertexte zu erstellen. Eine ähnliche gemeinsame Nutzung – hier allerdings bezogen auf Komponenten zum Drucken, Bearbeiten von Paletten sowie zum Import und Export von Graphikdaten – besteht bei den Programmen der *Corel Draw Graphics Suite* (*Draw* für Vektorzeichnungen, *Photopaint* für Rastergraphiken, *Capture* für Screenshots und *Texture* für die Bearbeitung von Mustern).

(4) **Gemeinsamkeiten verschiedener Ausführungen eines Grundprodukts eines Herstellers** (vgl. [CG96][2], [DNF96], [OS96], [Pre97], [CHW98], [CW98] und [GJKW98]) Die Anwendungsbereiche der Familienmitglieder sind entweder identisch oder trotz fachlicher Unterschiede in ihren Tätigkeiten so ähnlich, dass sie sich durch ein einziges Kernanwendungssystem unterstützen lassen. Die einzelnen Ausführungen der Mitglieder können sich dabei entweder in der Art oder im Umfang von Teilen der Funktionalität, in Details der bearbeiteten Gegenstände oder in der technischen Umsetzung von Teilsystemen bestehen. Beispiele für diese Art von Anwendungsfamilien sind z.B.

- Web-Browser mit verschiedenen HTML-Engines und/oder Komponenten zur Darstellung von nicht in HTML abgefassten Inhalten (Videos, Audio-Daten, etc.),
- Bildbearbeitungsprogramme mit unterschiedlichen Sätzen von Filtern zur Bildmanipulation (z.B. einen für die Bildverarbeitung mit dem Schwerpunkt auf Kontrastverstärkung und Kantenverdeutlichung im Gegensatz zu einem Satz von Filtern zur künstlerischen Verfremdung),
- Profi- und „Light“-Ausführungen derselben Anwendung.

Im Gegensatz zu den unter den Punkten (1) – (3) beschriebenen Arten von Gemeinsamkeiten erstrecken sich nur diejenigen unter Punkt (4) auf den gesamten Softwareentwicklungsprozess und werfen somit die in dieser Arbeit diskutierten Fragen über die bruchlose Modellierung über Analyse, Entwurf, Implementation und Auslieferung hin-

[2] Die doppelten Verweise auf einige Quellen bezüglich der Interpretation des Begriffs der Anwendungsfamilie sind darin begründet, dass die Autoren eine weit gefasste Definition des Begriffs verwenden (vgl. [CG96], [OS96] und [CW98]) oder explizit zwischen verschiedenen Arten von Anwendungsfamilien unterscheiden (vgl. [DNF96]).

weg auf. Im Folgenden bezeichne ich nur diejenigen Mengen von Anwendungssysteme als *Anwendungsfamilie*, die die unter (4) genannten Gemeinsamkeiten aufweisen. Die Eigenschaften und die Modellierung der unter (1) bis (3) beschriebenen Mengen von Anwendungssystemen werden nicht weiter betrachtet.

Bereits aus den bisher nur oberflächlich diskutierten Beispielen von Anwendungsfamilien geht hervor, dass es zumindest hinsichtlich der Relevanz der Komponenten für die Lauffähigkeit des Kernsystems unterschiedliche Arten der Anpassung gibt. Während Bildbearbeitungsprogramme auch ohne Filter immer noch zum Zeichnen taugen und „Light"-Ausführungen trotz funktionaler Einschränkungen brauchbare Programme sind, so sind Web-Browser ohne HTML-Engine vollkommen nutzlos, da hier eine Kernfunktionalität betroffen ist. Bevor ich das Verhältnis zwischen dem Kernsystem und Komponenten einerseits und den Komponenten untereinander andererseits eingehender untersuche, sollen die bisherigen Ergebnisse zunächst in einer Begriffsdefinition festgehalten werden:

Begriff 2.1 Anwendungsfamilie *(application family)*

Eine Anwendungsfamilie umfasst eine Menge von Anwendungssystemen eines Herstellers, die verschiedene Ausführungen ein und derselben chronologischen Version einer Anwendung darstellen. Eine Anwendungsfamilie besteht aus einem die Gemeinsamkeiten aller Mitglieder verkörpernden Kernsystem und dazu passenden Komponenten, mit denen das Kernsystem an unterschiedliche Anforderungen angepasst werden kann.

Komponenten können für ein Anwendungssystem essentielle Funktionalität enthalten, so dass das Kernsystem für sich genommen gegebenenfalls nicht einsetzbar ist.

Mit dieser Definition ist das Konzept der Anwendungsfamilie nach außen hin gegen verwandte Konzepte abgegrenzt. Im Inneren sind die Eigenschaften der Bestandteile Kernsystem und Komponenten aber noch nicht hinreichend definiert. Außer der Beobachtung, dass Komponenten auch Kernfunktionalität eines Anwendungssystems enthalten können, sind die Arten der Grenzen bzw. Verbindungsstellen zwischen dem Kernsystem und den Komponenten einerseits und den Komponenten untereinander andererseits bisher nicht erörtert worden.

Um die Fragen, deren Beantwortung zum Konzept der Anpassungsstelle und ihrer Varianten führen, vertiefe ich die bereits in groben Zügen umrissenen Beispiele der Familien von Bildbearbeitungssystemen bzw. Web-Browsern und führe zusätzlich das Beispiel eine Anwendungsfamilie von Simulationssystemen ein. Neben der im nächsten Kapitel beschriebenen komplexen Familie von Anwendungssystemen zur Unterstützung der musikwissenschaftlichen Analyse verwende ich diese drei ohne fachliche Einarbeitung verständlichen Beispiele im gesamten weiteren Verlauf dieser Arbeit, um die Eigenschaften bestehender und neuer Ansätze zur Modellierung von Anwendungsfamilien zu veranschaulichen.

1. Beispiel: Familie von Bildbearbeitungssystemen

Mit Bildbearbeitungsprogrammen können Rastergraphikbilder nachbearbeitet und manipuliert werden. Sie stellen den Anwendern mehrere Zeichenflächen zur Verfügung, in die Graphiken geladen und mit elementaren Graphikelementen wie Punkten, Linien, Bögen und Zeichen versehen werden können. Zu den Gemeinsamkeiten aller Familienmitglieder gehören ebenfalls Operationen zum Ausschneiden, Kopieren und Einfügen von Graphiken; zum Bearbeiten der Farbpalette; zur Änderung der Parameter wie Linienstärke und -art, gemäß derer Graphikelemente angebracht werden kann; sowie Operationen zum Speichern der bearbeiteten Graphiken.

Um den Anwendern die Arbeit der Integration der Ergebnisse verschiedener Anwendungen einer Anwendungsgruppe – wie Anwendungen zur Bildakquisition per Screenshot oder von Scannern – zu ersparen, können diese auch unmittelbar ins Bildbearbeitungssystem eingebettet werden, so dass die gesamte Anwendungsgruppe zu einem einzigen System wird.

Abgesehen von diesen Gemeinsamkeiten können sich Familienmitglieder in viererlei Hinsicht unterscheiden:

(1) Sie bieten die bereits erwähnten, verschiedenen Arten von Filtern an (vgl. S. 12). Ebenso austauschbar sind die

(2) Konverter zum Import und

(3) Export von Bilddaten in verschiedenen Graphikformaten. Import und Export stellen separate Formen der Anpassung dar, da das Lesen im Gegensatz zum Schreiben von insbesondere von komplexen Formaten grundverschiedene Algorithmen erfordern. Außerdem kann aus der Tatsache, dass ein Familienmitglied das Speichern z.B. in den primär zur Ausgabe konzipierten Formate Postscript und PDF unterstützt, nicht automatisch geschlossen werden, dass Anwender sie auch als Datenquelle nutzen wollen.

(4) Schließlich können auch die eingebauten Subsysteme z.B. zur Bildakquisition ausgewechselt werden.

Offensichtlich bestehen unter den Komponenten unterschiedlicher Anpassungsarten keine gegenseitigen Verbindungen, so dass sie sich problemlos voneinander abgrenzen lassen: Ob ein Import-Konverter verwendet werden kann oder nicht, hängt nicht davon ab, dass ein bestimmter Export-Konverter, Filter oder Akquisitionssubsystem vorhanden ist. Dieselbe Unabhängigkeit lässt sich auch zwischen Komponenten der selben Anpassungsart durchgängig konstatieren. Da sich diese Beobachtung sowohl für die bereits diskutierten Aspekte der Beispiele der Web-Browser als auch der „Light“- und Profi-Ausführungen machen lässt, scheint hier eine generelle Eigenschaft des Verhältnisses von Komponenten von Anwendungsfamilien vorzuliegen.

Bezüglich des Verhältnisses der Komponenten zum Kernsystem sind die Eigenschaften der Bildbearbeitungsfamilie weniger typisch: Für alle Anpassungsarten können hier prinzipiell beliebig viele Filter, Konverter, und Subsysteme zur Bildakquisition ausge-

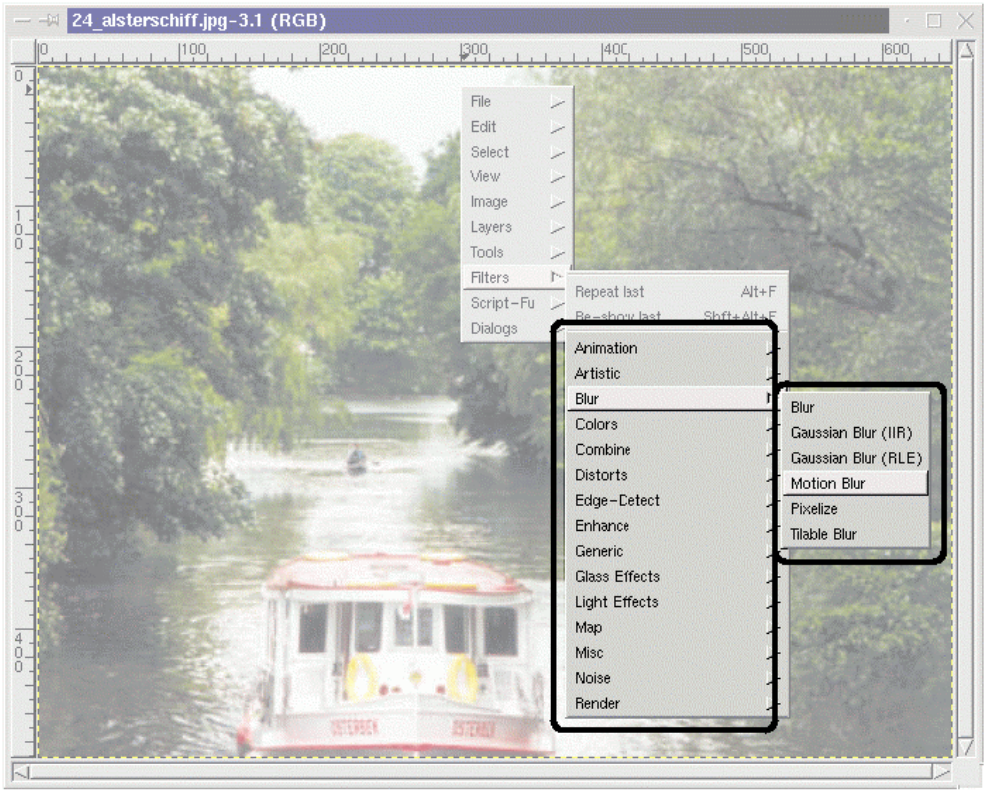


Abb. 2-1 Bildbearbeitungssystem mit aufgeklappten Filter-Menüs

Diejenigen Teile der sichtbaren Bedienschnittstelle, die sich in Abhängigkeit der gewählten Anpassung ändern können, sind hervorgehoben.

wählt werden, die zudem sämtlich optional sind. Dem steht die Beobachtung im Web-Browser-Beispiel entgegen, in dem genau eine HTML-Engine notwendigerweise vorhanden sein muss, um ein lauffähiges System zu erhalten. Das Verhältnis zwischen Kernsystem und Komponenten bedarf daher noch der genaueren Untersuchung. Hierzu verfeinere ich zunächst das Beispiel der Web-Browser.

2. Beispiel: Familie von Web-Browsern

Die Kernfunktionalität von Web-Browsern besteht darin, Verbindungen mit Web-Servern im Hypertext-Transfer-Protokoll (http) aufzubauen, die Ergebnisse darzustellen und die Adressen der besuchten Seiten zu Navigationszwecken zu speichern (zurück, Liste der besuchten Seiten, etc.). Um den Benutzungskomfort, können außerdem Eingaben im Adressfeld, die keine gültige http-Adresse sind, automatisch an Suchmaschinen weitergeleitet werden.

Wie auch bei Bildbearbeitungssystemen lassen sich andere Anwendungssysteme aus einer Internet-Anwendungsgruppe wie Mail- und News-Clients unmittelbar in den Web-Browser einbinden, so dass diese automatisch aufgerufen werden, wenn die Anwender Verweise des Typs `mail:` bzw. `news:` auswählen. Zusammen mit den bereits zuvor diskutierten Komponententypen ergeben sich mindestens vier Arten, auf die ein Web-Browser angepasst werden kann:

(1) Eine HTML-Engine ist notwendig, um die übertragenen Daten zu interpretieren und darzustellen. Sie ist aber in ihrer Arbeitsweise von den anderen Systemteilen unab-

hängig und kann ausgetauscht werden, um z.B. statt Dokumenten HTML 4.01 auch Dokumente in XHTML 1.0 verarbeiten zu können.

(2) Der Mail-Client und

(3) der News-Client sind sowohl austauschbar als auch optional, da sie Kernfunktionalität des Browsers nicht berührt werden. Es kann maximal ein Client jedes Typs vorhanden sein. Da beide Clients unterschiedliche Dienste erbringen, handelt es sich im Gegensatz zu den jeweils neue Graphiken liefernden Screenshot- und Scanner-Komponenten des Bildbearbeitungssystems auch um zwei verschiedene Anpassungsarten.

(4) Die Komponenten, mit deren Hilfe nicht in HTML-codierte Hypertextinhalte wie Video- oder Audiodaten dargestellt werden können, sind ebenfalls austauschbar und optional. Falls sie nicht für den Aufbau von Hypertextseiten zur Verfügung stehen, bleibt die für sie vorgesehene Stelle im Dokument leer.

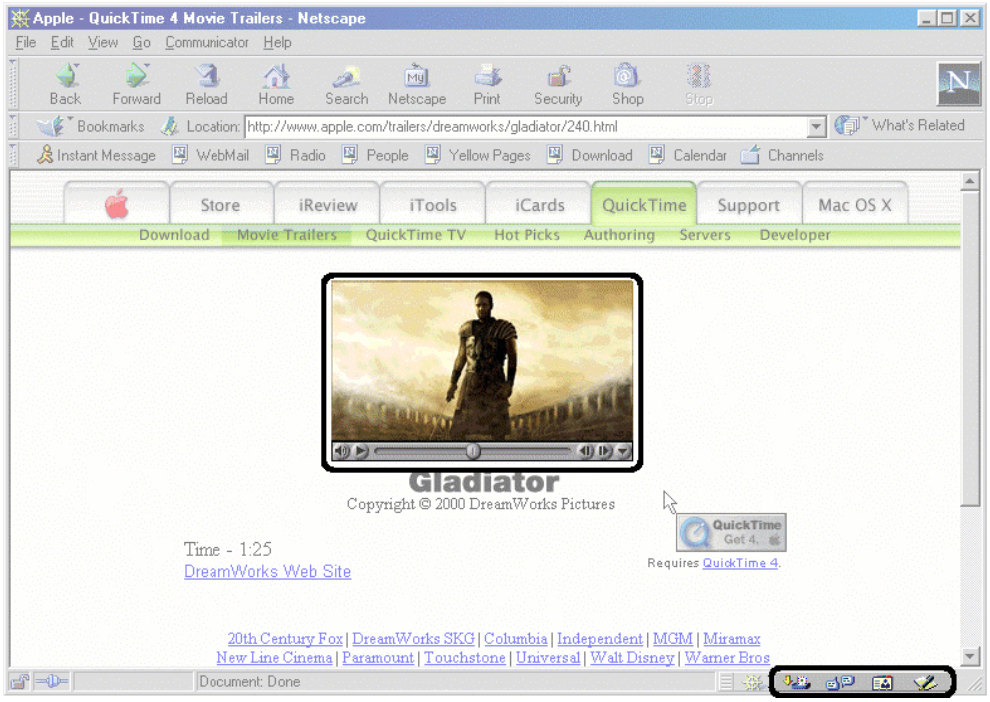


Abb. 2-2 Web-Browser mit Hypertextseite, die ein Video enthält.
Hervorhebungen wie in Abb. 2-1.

Wie auch bei der Familie von Bildbearbeitungssystemen können die einzelnen Komponenten unabhängig voneinander eingesetzt werden, so dass eine Komponente zum Anzeigen von Videos weder von der HTML-Engine, dem Mail- oder News-Client noch einer Komponente zum Anhören von Audio-Daten abhängt.

Im Gegensatz zum vorangegangen Beispiel lassen sich hier aber zwei grundverschiedene Arten der Anpassung beobachten:

(1) Zu keinem Zeitpunkt können mehr als eine HTML-Engine, ein Mail-Client oder ein News-Client im System vorhanden sein.

(2) Dagegen kann es Komponenten zum Anzeigen von Nicht-HTML-Daten – ebenso wie z.B. Import-Konverter im Bildbearbeitungssystem – beliebig häufig geben oder sie können sogar ganz fehlen.

Anhand des Web-Browser-Beispiels lässt sich daher feststellen, dass es verschiedene Arten von Verbindungsstellen des Kernsystems gibt, die entweder zur einfachen oder zur mehrfachen Anpassung geeignet sind: Einfach- und Mehrfachanpassungsstellen. Mehrfachanpassungsstellen sind dabei stets optional, während Einfachanpassungsstellen sowohl optional (Mail-Client, News-Client) als auch für die Lauffähigkeit des System notwendig sein können (HTML-Engine).

> **Begriff 2.2 Anpassungsstelle *(adaptation spot)*, 1. Fassung**
>
> Eine Stelle des Kernsystems einer Anwendungsfamilie, an der dieses an wechselnde Anforderungen angepasst werden kann, heißt Anpassungsstelle. Besitzt eine Kernsystem mehrere Anpassungsstellen, so verkörpern diese voneinander unabhängige Arten der Anpassung.
>
> Anpassungsstellen können so beschaffen sein, dass dort entweder genau eine oder beliebig viele Komponenten angebracht werden können (*Einfach- oder Mehrfachanpassungsstelle*).

Anhand der beiden Beispiele hat sich noch nicht herauskristallisiert, ob ein Zusammenhang zwischen der Art der Anpassungsstelle und der Funktionalität besteht, die in den passenden Komponenten verkörpert ist. Alle bisher untersuchten Komponenten lassen sich im weitesten Sinne als Dienstleister auffassen,[3] die vom Kernsystem genutzt werden können. Anhand des im Folgenden eingeführten dritten und letzten Beispiels einer Anwendungsfamilie von Simulationssystemen werde ich zeigen, dass sich ohne Widerspruch zu den bisherigen Beobachtungen Dienstleister zu Einfachanpassungsstellen und Systembausteine zu Mehrfachanpassungsstellen zuordnen lassen.

3. Beispiel: Familie von Simulationsanwendungen

Simulationsanwendungen erlauben es ihren Benutzern, zunächst ein zu simulierendes System aus vorgefertigten, parametrisierbaren Elementen zusammenzustellen, und anschließend zu beobachten, wie das System auf geplante oder zufällige Ereignisströme reagiert. Beispiele für Simulationsanwendungen sind *HADES*[4][Hen01], *Debora*[Hee93] und die von Wolfgang Pree in [Pre97, S. 27ff] diskutierte Anwendung. Kernsysteme von Simulationsanwendungen lassen sich in zweierlei Hinsicht anpassen:

(1) Die Elemente, aus denen das zu simulierende System aufgebaut werden kann, sind je nachdem, was für eine Art von System simuliert werden soll, austauschbar.

[3] Michael Otto und Norbert Schuler zeigen in [OS00], dass der Begriff „Dienstleister“ in der Informatik-Literatur ähnlich unscharf verwendet wird, wie der in Kapitel 4 diskutierte Begriff „Komponente“. Der in dieser Dissertation verwendete Dienstleisterbegriff ist der des „zentralen Dienstleisters“ aus [OS00, S. 64]. Zusätzlich zu dem allgemeinen Dienstleisterbegriff, unter dem sich auch Klassen mit ihren Operationen subsumieren lassen, ist bei einem zentralen Dienstleister sichergestellt, dass er zur Laufzeit nur einmal im System vorhanden ist. Spezielle Annahmen über Mehrbenutzerfähigkeit, Verteilung und Sitzungsmodelle (vgl. [OS00, S. 78ff]) werden nicht gemacht.

[4] Das Akronym steht für Hamburg Design System.

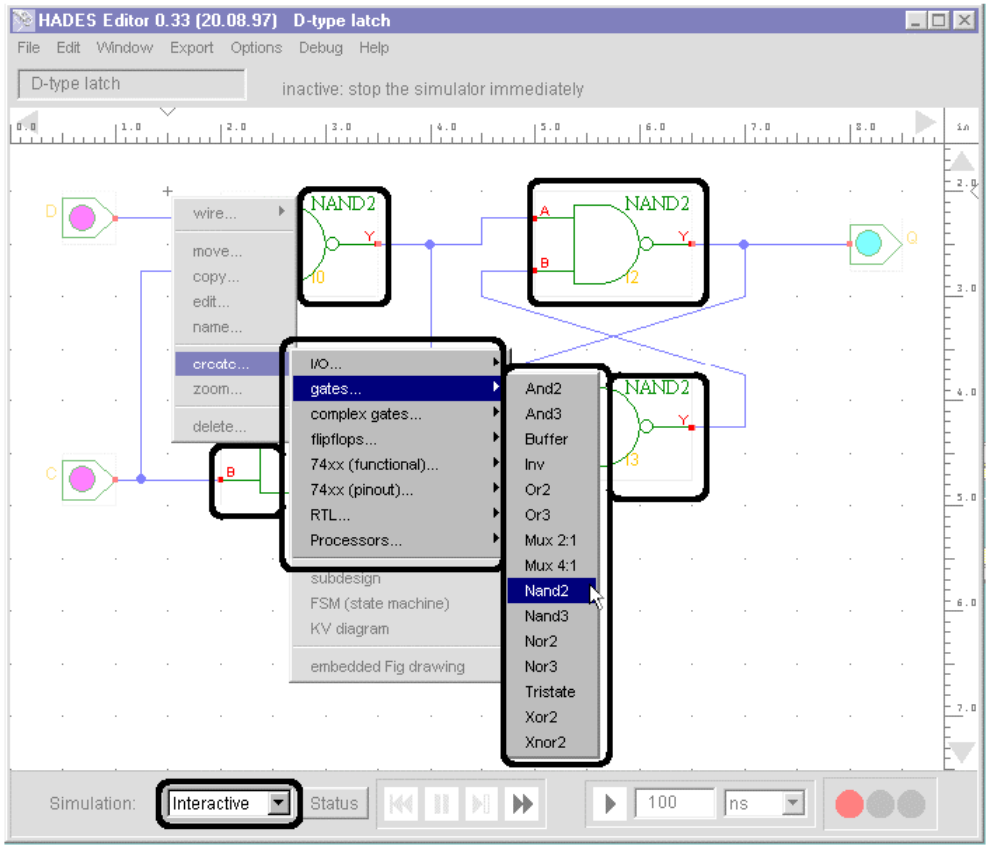


Abb. 2-3 Simulationsanwendung in der Konstruktionsphase

Über die aufgeklappten Menüs können die Elemente ausgewählt werden, die in dem simulierten System hinzugefügt werden sollen. In der Arbeitsfläche ist das bereits konstruierte D-Flipflop zu sehen. Hervorhebungen wie in Abb. 2-1.

(2) Der Taktgeber, der bestimmt, in welchen zeitlichen Abständen die Ereignisse abgearbeitet werden, kann sich z.B. entweder nach Realzeit oder Simulationszeit richten. Realzeit bzw. linear gestauchte Realzeit eignet sich am besten dann, wenn die Anwender unmittelbar in die Simulation eingreifen wollen. Bei vollkommen vordefinierten Ereignisströmen empfehlen sich Taktgeber in Simulationszeit, bei denen ereignislose Zeiträume übersprungen werden.

Erneut bestätigen sich die bereits an den anderen beiden Beispielen gemachten Beobachtungen, dass die Komponenten voneinander unabhängig sind und dass es zwei unterschiedliche Arten der Anpassung gibt: Für die Elemente gibt es eine optionale Mehrfachanpassungsstelle und für den Taktgeber eine in diesem Fall notwendige Einfachanpassungsstelle.

Wie auch in den vorangegangenen Beispielen lassen sich die Komponenten an der Einfachanpassungsstelle als Verkörperungen von Diensten auffassen, während dies für die Komponenten an der Mehrfachanpassungstelle (die einzelnen Elemente, aus denen das System aufgebaut werden kann) in diesem Fall nicht zutrifft. Sie stellen vielmehr Bausteine dar, aus denen Teile der Bedienschnittstelle der Anwendung sowie des Anwendungsgegenstands konstruiert werden:

- **Baustein für die Bedienschnittstelle.** Für jede Komponente an der Mehrfachanpassungsstelle baut das Anwendungssystem genau einen Eintrag in einem Untermenü

des Konstruktions-Pop-Up-Menüs ein – z.B. für die Komponente „NAND2“ den Menüeintrag „Nand2“ und für die Komponente „XNOR2“ an der gleichen Mehrfachanpassungsstelle den Menüeintrag „Xnor2“. Das heißt, die Struktur der Bedienschnittstelle richtet sich danach, welche Komponenten an der Mehrfachanpassungsstelle vorhanden sind.

Bei einer Einfachanpassungsstelle bestimmt sich die Struktur der Bedienschnittstelle hingegen nicht nach einer dort ggf. vorhandenen Komponente, sondern wird vom Kernsystem fest vorgegeben. Es gibt (maximal) *genau ein* mit der Einfachanpassungsstelle assoziiertes Bedienelement, das zunächst deaktiviert ist und lediglich dann aktiviert wird, wenn eine Komponente vorhanden ist (z.B. ein Mail- oder News-Client). Am strukturellen Aufbau der Bedienschnittstelle ändert sich aber nichts. In einigen Fällen (wie bei der HTML-Engine), existiert sogar überhaupt kein Bedienelement für die Komponente an der Einfachanpassungsstelle.

- **Baustein für den Anwendungsgegenstand.** Wählen die Anwender im Simulationssystem einen Menüeintrag aus, so erscheint ein Exemplar des dazugehörigen Elements in der Konstruktionsfläche. Es steht dann zur Verfügung, um daraus ein zu simulierendes System aufzubauen.

  Da Elemente einer Art beliebig häufig im zu simulierenden System vorkommen können, besteht aber keine Eins-zu-Eins-Beziehung zwischen genau einer Komponente an der Mehrfachanpassungstelle und genau einem konkreten Element in der Konstruktionsfläche. Stattdessen verkörpert die Komponente den *Bauplan eines Bausteins*, dessen sich eine im Kernsystem befindliche Baustein-Fabrik bedienen kann, um je nach Bedarf beliebig viele gleichartige Bausteine herzustellen.

  Komponenten an Einfachanpassungsstellen verkörpern dagegen stets im System nur einmal vorhandene Dienste, die in keinem der untersuchten Beispiele in der Form von Bausteinen in die Anwendungsgegenstände eingehen. Wenn überhaupt, dann bilden sie wie im Fall der HTML-Engine die Grundplatte, auf der Bausteine erst aufbauen.

Die Beobachtung, dass zu Mehrfachanpassungsstellen passende Komponenten als Bausteine fungieren, lässt sich auch anhand der beiden zuvor diskutierten Beispiele nachvollziehen:

- Im Web-Browser-Beispiel werden die Komponenten zur Darstellung von Nicht-HTML-Daten von der HTML-Engine als Bausteine eingesetzt, um daraus Teile des Anwendungsgegenstands – die Nicht-HTML-Bestandteile der Hypertextdokumente – zu konstruieren.

- Im Bildbearbeitungssystem werden in Abhängigkeit davon, welche Komponenten an der Mehrfachanpassungsstelle vorhanden sind, dementsprechende Menüs bzw. Auswahllisten an der Bedienschnittstelle aufgebaut, um zwischen den verschieden Import- und Export-Konvertern, den Filtern sowie den Bildakquisitionsquellen auszuwählen.

Das bedeutet, dass es die primäre Eigenschaft von Komponenten an Mehrfachanpassungsstellen ist, ein Baustein zu sein. Während sie auch Merkmale von Diensten haben können (z.B. Filter), so haben im Gegensatz dazu Komponenten an Einfachanpassungsstellen niemals Merkmale von Bausteinen, sondern ausschließlich von Diensten.

Es versteht sich von selbst, dass nicht jeder Bestandteil einer Anwendungsfamilie, der die Eigenschaften eines Bausteins oder eines Dienstleisters hat, auch zwangsläufig eine Komponente ist. Dies ist nur für diejenigen Bausteine bzw. Dienstleister sinnvoll, die auch einen Unterschied zu anderen Anwendungsfamilienmitgliedern verkörpert. Bausteine, die keine Komponenten sind, sondern zum Kernsystem gehören, sind z.B. die festen Einträge im Menü „Filters“ (siehe Abb. 2-1) des Bildbearbeitungssystems bzw. im Menü „create…“ des Simulationssystemeditors (siehe Abb. 2-3) sowie die stets vorhandenen Bausteine zur Darstellung von Bildern in den Formaten GIF und JPEG im Web-Browser (siehe Abb. 2-2). Zum Kernsystem gehörige, nicht austauschbare Dienstleister könnten z.B. Maßeinheiten von Pixeln in Zentimeter umrechnen (Bildbearbeitungssystem) oder auf Wunsch http-Verbindungen aufbauen (Web-Browser).

Bausteine und Dienstleister, die zum Kernsystem gehören, bezeichne ich im Folgenden als *interne Bausteine* und *interne Dienstleister*.

Begriff 2.3 Interne Bausteine und Dienstleister (*internal building blocks and services*)

Systembausteine oder Dienstleister, die in jedem zur Anwendungsfamilie gehörenden Anwendungssystem vorhanden sind, heißen interne Bausteine oder interne Dienstleister. Sie gehören zum Kernsystem und sind nicht in Komponenten enthalten.

Anhand der drei untersuchten Beispiele habe ich das Konzept der Anwendungsfamilie nun auch bezüglich seiner inneren Bestandteile und deren Verhältnis zueinander hinreichend genau beschreiben können. Bevor ich das hierfür zentrale Konzept der Anpassungsstelle mit in der Literatur diskutierten Konzepten zur Anpassung von Anwendungssystem vergleiche, halte ich die herausgearbeiteten Eigenschaften von Anpassungsstellen in einer Begriffsdefinition fest.

Begriff 2.4 Anpassungsstelle *(adaptation spot)*, 2. Fassung

Eine Stelle des Kernsystems einer Anwendungsfamilie, an der dieses an wechselnde Anforderungen angepasst werden kann, heißt Anpassungsstelle. Besitzt eine Kernsystem mehrere Anpassungsstellen, so verkörpern diese voneinander unabhängige Arten der Anpassung.

Anpassungsstellen können so beschaffen sein, dass dort entweder genau eine Dienstleisterkomponente (*Einfachanpassungsstelle*) oder beliebig viele als Systembausteine verwendete Komponenten angebracht werden können (*Mehrfachanpassungsstelle*).

Eine Anpassungsstelle, an der eine Komponente angebracht werden muss, um eine lauffähige Anwendung zu erhalten, heißt *Zwangsanpassungsstelle*.

Zwei verwandte Konzepte zur Beschreibung von Anpassungen von Anwendungssystemen sind Erweiterungsdimensionen (engl. *dimensions of extensibility*; vgl. [Wec97] und [Szy98, S. 90f]) und Hot Spots (vgl. [Pre97, S. 63ff]). Da beiden aber ein weniger klar abgegrenztes bzw. ein stärker eingeschränktes Anwendungsfamilienkonzept zugrunde liegt, werde ich sie im weiteren Verlauf dieser Arbeit zugunsten des Konzepts der Anpassungsstelle nicht weiter behandeln.

- *Erweiterungsdimensionen* liegt ein Konzept zugrunde, das sich nur teilweise mit dem der Anwendungsfamilie überlappt. Wie bei Anwendungsfamilien wird von einem anpassbaren Kernsystem ausgegangen. Von diesem Kernsystem wird angenommen, dass es n verschiedene Eigenschaften aufweist, die einen n-dimensionalen Vektorraum aufspannen, wobei die einzelnen Dimensionen als Eigenschaftsdimensionen bezeichnet werden. Diejenigen Eigenschaftsdimensionen, deren Eigenschaft nicht bereits im Kernsystem verbindlich für alle Familienmitglieder festgelegt sind, werden Erweiterungsdimensionen genannt. Wie auch bei Anpassungsstellen, gibt es die Möglichkeit der Einfachanpassung (*singleton configuration*) und der Mehrfachanpassung (*parallel extension*). Im Gegensatz zum Konzept der voneinander unabhängigen Anpassungsstellen und entgegen der Anlehnung an mathematische Terminologie wird aber von Erweiterungsdimensionen nicht verlangt, dass sie zueinander orthogonal sein müssen. Was als Eigenschaftsdimension des Kernsystems modelliert wird, richtet sich nämlich nicht nach dessen beim Entwurf klar umrissenen Eigenschaften, sondern nach den Eigenschaften von bereits existierenden, in Unkenntnis des Kernsystems entwickelten Komponenten. Ähneln oder überlappen sich also z.B. zwei verwendete Komponenten in ihrer Funktionalität, so etablieren sie zwei zueinander nicht orthogonale Erweiterungsdimensionen, die sie jeweils zum einen ganz und zum anderen teilweise abdecken. Das Konzept der Erweiterungsdimensionen zielt also trotz der auf ein Kernsystems bezogenen Terminologie nicht auf die geplante Modellierung von Anwendungsfamilien, sondern auf die maximale Wiederverwendung von bereits bestehenden Komponenten in einem zu erweiternden Programmen ab. Schwierigkeiten bereitet dabei vor allem, dass ein – dort auch problematisierter – Bedarf für eine aufwendige Koordination zwischen den sich funktional überlappenden Komponenten entsteht. Die beiden weiteren Nachteile von Erweiterungsdimensionen sind:

  1. Anpassungen sind ausschließlich auf Erweiterungen eines bereits für sich lauffähigen Systems bezogen. Komponenten wie HTML-Engines und Taktgeber, die essentielle Kernfunktionalität von Anwendungssystemen verkörpern, lassen sich mit Erweiterungsdimensionen nicht beschreiben.

  2. Im Gegensatz zum Konzept der Anpassungsstelle wird kein Zusammenhang zwischen der Art der Anpassung (einfach vs. mehrfach) und der Art der passenden Komponenten (Dienste vs. Bausteine) etabliert.

- *Hot Spots* sind spezielle Stellen eines Kernsystem einer Anwendungsfamilie, die eine echte Anpassung – und nicht nur Erweiterung – an einer vorgegebenen Verbindungsstelle erlauben, um ein konkretes Familienmitglied zu erzeugen. Wie auch bei Anpassungsstellen besteht ein Zusammenhang zwischen Einfachanpassungsstellen und Dienstkomponenten. Im Gegensatz zu Anpassungsstellen sind aber Mehr-

fachanpassungsstellen und Bausteine nicht vorgesehen, wodurch z.B. das gesamte Bildbearbeitungsbeispiel nicht beschreibbar wäre. Zusätzlich wird die Komplexität der Dienste an den Einfachanpassungsstellen dadurch begrenzt, dass sie sich über eine einzige Operation ausdrücken lassen müssen (vgl. [Pre97, S. 71]). Komplexe Dienste wie HTML-Engines oder Mail- und News-Clients lassen sich mit Hot Spots daher nicht beschreiben.

Während die Konzepte der Anwendungsfamilie und der Anpassungsstelle keinerlei Festlegung darüber enthalten, wann und durch wen die Anpassung des Kernsystems durch die Komponenten erfolgt, werde ich im Folgenden die Zusammenstellung zu einem Familienmitglied durch die Entwickler während der Entwicklung oder durch Scripting zur Konfigurationszeit nicht weiter betrachten. Statt dessen untersuche ich gemäß der Zielsetzung dieser Arbeit ausschließlich Modellierungsansätze, mit denen eine Anpassung von Sitzung zu Sitzung durch die Anwender selbst erreicht werden kann. Dabei werde ich zeigen, dass eine für die Anwender verständliche Vergegenständlichung der Komponenten und der Anpassungsstellen eine Handhabung durch die Anwender wesentlich erleichtert.

In Kapitel 3 diskutiere ich zunächst die Anforderungen an drei komplexe Anwendungssysteme zur Unterstützung der musikwissenschaftlichen Analyse und weise nach, dass sie sich trotz Unterschieden im musiktheoretischen Hintergrund, der Beschaffenheit der verwendeten Arbeitsgegenstände und des Grades der Nutzung bestehender MDV-Systeme als eine Anwendungsfamilie mit drei Anpassungsstellen auffassen lassen, deren Kernsystem ich zum Abschluss des Kapitels beschreibe. Basierend auf diesem Hauptbeispiel und den bereits in diesem Kapitel eingeführten drei Beispielen untersuche ich in Kapitel 4 bestehende Ansätze zur Modellierung von Anwendungsfamilien mit Rahmenwerk und Komponenten-Technologien und arbeite heraus, dass mit ihnen die geforderte flexible Anpassung durch die Anwender problematisch ist.

In Kapitel 5 beschreibe ich den Einschub-Ansatz, mit dem die zuvor geschilderten Probleme überwunden werden können. Kern des Ansatzes ist die Einschub-Metapher, die Komponenten als Einschübe und Anpassungsstellen als Einschubrahmen vergegenständlicht und somit für alle am Softwareentwicklungsprozess Beteiligten begreiflich macht. Den Ansatz validiere ich konstruktiv am Beispiel *JRings*, das als Einschubsystem realisiert ist. Schließlich wende ich mich in Kapitel 6 dem Problem der flexiblen Kapselung von MDV-Subsystemen zu und zeige, wie es sich im Rahmen des Einschub-Ansatzes vorteilhaft lösen lässt.

3 Ein Kernsystem für die musikwissenschaftliche Analyse

Ob eine Anwendungsfamilie mit Kernsystem, Anpassungsstellen und Komponenten vorliegt, lässt sich insbesondere dann gut beantworten, wenn wie bei den im letzten Kapitel diskutierten Beispielen die betreffenden Anwendungsbereiche einfach strukturiert und analytisch gut durchdrungen sind, so dass sich Untersuchungen weitgehend auf existierende Anwendungssysteme stützen können. Bereits zu Beginn der Einleitung habe ich auf die besondere Situation in den Geisteswissenschaften und insbesondere der Musikwissenschaft verwiesen: Die Anwendungsbereiche verschiedener musikwissenschaftlicher Analysearten, die in dieser Arbeit aufgrund ihrer besonderen Flexibilitätsanforderungen als Hauptanwendungsbeispiel dienen, sind weder einfach strukturiert noch analytisch gut durchdrungen und es liegen keine bestehenden Anwendungssysteme vor.

Um daher untersuchen zu können, in welchem Ausmaß die für Anwendungsfamilien charakteristischen Gemeinsamkeiten und Unterschiede zwischen verschiedenen musikwissenschaftlichen Analysesystemen bestehen, müssen diese Analysesysteme zunächst von Grund auf spezifiziert werden. Die Grundlage müssen dabei im Sinne der Anwendungsorientierung empirische Analysen des Entwicklungskontexts *Anwendungsbereich* (siehe S. 6) für verschiedene Analysearten bilden, aus denen die Gemeinsamkeiten und Unterschiede deutlich hervorgehen. Um hierbei die Unterschiede besonders klar hervortreten zu lassen, untersuche ich die sich musiktheoretisch erheblich unterscheidenden Bereiche der thematischen, der leitmotivischen und der Set-Theory-Analyse.

Zahlreiche der in der Literatur vorgeschlagenen Methoden sind für diese empirische Analyse allerdings problematisch, da sie bereits von existierenden softwaretechnischen Modellen oder Anwendungssystemen ausgehen bzw. nicht auf die Analyse von Anwendungsfamilien ausgelegt sind:

- Die *klassischen Methoden zur objektorientierten Analyse* (OOA; vgl. [CJJO92], [Boo94] und [Der96]) gehen zwar im Sinne der Anwendungsorientierung unmittelbar von einem bestimmten Anwendungsbereich aus, fokussieren aber nur auf ein einziges zu entwickelndes Anwendungssystem anstatt auf eine Anwendungsfamilie.

- Der *FAST-Ansatz und die dazu passende SCV-Analyse*[1] (vgl. [CHW98], [CW98] und [GJKW98]) sind dementgegen speziell auf Anwendungsfamilien ausgerichtet. Sie fußen aber auf der Annahme, dass Softwareentwicklung primär auf die Weiterentwicklung bestehender Software beschränkt ist (vgl. [CW98, S. 24]) und gehen dementsprechend von einem bestehenden softwaretechnischen Modell und nicht unmittelbar von einem Anwendungsbereich aus.

[1] Die Akronyme stehen für Family-Oriented Abstraction, Specification and Translation bzw. Scope, Commonality and Variability.

- Die *Hot-Spot-Analyse* (vgl. [Pre97]) kommt den in dieser Arbeit benötigten Anforderungen näher als die beiden anderen Ansätze, da sie sowohl von vorneherein auf die Analyse einer Anwendungsfamilie abzielt als auch unmittelbar von einem konkreten Anwendungsbereich ausgeht. Sie ist jedoch aus Sicht der Anwendungsorientierung problematisch, da bei ihr kein explizites, für die Anwender verständliches fachliches Modell des Anwendungsbereichs (siehe Seite 6) erstellt wird, auf dessen Grundlage die Analyseergebnisse kritisiert und Fehlentwicklungen frühestmöglich vermieden werden können (vgl. [Pre97, S. 65] und [WAM98, S. 132f]).[2]

- Dieses Problem der Hot-Spot-Analyse behebt der *WAM-Ansatz* (siehe Seite 5) indem er auf der Grundlage von vollkommen in Anwendersprache abgefassten Analysedokumenten[3] ein explizites fachliches Modell des Anwendungsbereichs vorschreibt, auf dessen Basis Missverständnisse zwischen Entwicklern und Anwendern erkannt und ausgeräumt werden können. Da mit dem WAM-Ansatz ebenso wie mit den anderen Ansätzen die für Anwendungsfamilien charakteristischen Unterschiedlichkeiten identifiziert werden können (vgl. [Kra00, S. 101ff] und [WAM98, S.607f]), erfolgte die diesem Kapitel zugrundeliegende empirische Analyse nach dem WAM-Ansatz.

Aus der anfangs geschilderten Situation in der Musikwissenschaft und der Wahl des WAM-Ansatzes ergibt sich der weitere Aufbau dieses Kapitels. In den Abschnitten 3.1 bis 3.3 diskutiere ich jeweils separat für die Entwicklungskontexte *Anwendungsbereich*, *Handhabung & Präsentation* und *verwendete Technik* (siehe S. 6) die Anforderungen, die an Anwendungssysteme zur Unterstützung der musikwissenschaftlichen Analyse nach thematischen, leitmotivischen und set-theoretischen Gesichtspunkten gestellt werden und halte dabei in Kontextspezifikationen fest

(a) welche speziellen Herausforderungen sich für die softwaretechnische Modellierung ergeben,

(b) inwiefern sich Gemeinsamkeiten und Unterschiedlichkeiten feststellen lassen und

(c) ob die Unterschiedlichkeiten voneinander unabhängig sind und sich die spezifizierten Anwendungssysteme somit als eine Anwendungsfamilie auffassen lassen.

In Abschnitt 3.4 zeige ich darauf aufbauend, dass sich die verschiedenen umrissene Anwendungssysteme tatsächlich als eine Anwendungsfamilie auffassen lassen und skizziere das Kernsystem mit seinen Anpassungsstellen und den dazu passenden Komponenten, bevor ich das Kapitel in Abschnitt 3.5 mit einer Zusammenfassung der Ergebnisse und der sich daraus ergebenden Anforderungen an die softwaretechnische Modellierung abschließe.

---

[2] Lediglich für den die Unterschiedlichkeiten ausmachenden Teil einer Anwendungsfamilie verwendet die Hot-Spot-Analyse auch für die Anwender verständliche Hot-Spot-Karten. Für die Analyse der Gemeinsamkeiten, die den ganz überragenden Teil einer Anwendungsfamilie ausmachen, baut die Hot-Spot-Analyse dementgegen auf CRC-Karten, die die Analyseergebnisse unmittelbar in Beschreibungen von Klassen mitsamt deren Verpflichtungen und Verflechtungen festhalten (vgl. [BC89]).

[3] Für die Ist-Analyse kommen bei WAM die Dokumenttypen *Glossar* (für Begriffe und Gegenstände) und *Szenario* (für Tätigkeiten) zum Einsatz. Die Ergebnisse von Soll-Analysen werden in Dokumenten des Typs *ermittelte Anforderungen* festgehalten (vgl. [WAM98, S. 601ff]). Da ich die musikwissenschaftlich bedingten Anpassungen der Dokumenttypen sowie der WAM-Vorgehensweise bereits in [Kor96a] dargelegt habe, befasse ich mich in dieser Arbeit ausschließlich mit den in den Dokumenten festgehaltenen Ergebnissen.

3.1 Anwendungsbereich

Die musikwissenschaftliche Analyse dient der Aufdeckung und Vermittlung struktureller und inhaltlicher musikalischer Zusammenhänge. Der Analysegegenstand kann sich dabei auf ein einzelnes Stück beschränken oder auf mehrere Werke eines oder mehrerer Komponisten, einer oder mehrerer →Gattungen[4], einer oder mehrerer →Stilrichtungen sowie einer oder mehrerer →Epochen erstrecken.

Da es sich sowohl bei den untersuchten Musikstücken als auch bei den darauf angewandten analytischen Theorien um von Menschen geschaffene Gebilde handelt, sind Analysen zwangsläufig immer subjektiv. Objektivität kann nur dem Anschein nach durch Bezug auf eine vorherrschende Interpretationsart einer bestimmten Theorie erreicht werden, doch selbst ein so natürlich erscheinender Vorgang wie die Reduktion eines Stückes auf eine Folge von →Tonhöhen und -dauern ist bereits eine qualitative Entscheidung. Musikanalyse und Musiktheorie sind daher auf das engste miteinander verbunden. Während die Musiktheorie dabei von abstrakten Ideen ausgeht und diese an konkreten Musikstücken demonstriert oder anhand derer belegt, so geht umgekehrt die Analyse von konkreten Musikstücken aus und versucht, diese mit Hilfe abstrakter Theorien zu erklären. Der implizit oder explizit erste Schritt bei der Durchführung einer Analyse ist daher die Einnahme eines übergeordneten Standpunktes, von dem aus analysiert werden soll. Von der hierbei gewählten Theorie hängt es ab, auf welche Weise die Analyse durchgeführt wird und welche Ergebnis möglich sind.

Da die Antwort auf die Frage, ob sich Anwendungssysteme zur Unterstützung der thematischen, der leitmotivischen und der Set-Theory-Analyse als eine Anwendungsfamilie auffassen lassen, im Sinne der Anwendungsorientierung primär von den im Rahmen einer Ist-Analyse des *Anwendungsbereichs* ermittelten Gegenständen und Tätigkeiten abhängt, und diese Gegenstände und Tätigkeiten wiederum durch die gewählte musikwissenschaftliche Theorie bedingt werden, untersuche ich in den Abschnitten 3.1.1 bis 3.1.3 die drei Anwendungsbereiche jeweils in drei aufeinander aufbauenden Schritten:

- In einem *ersten Schritt* beschreibe ich den jeweiligen Untersuchungsgegenstand sowie dessen musiktheoretischen und -historischen Kontext.

- In einem *zweiten Schritt* schildere ich die Analyseziele und die Elemente, die ein ideales Analyseergebnis aufweisen sollte.

- In einem *dritten Schritt* zeige ich schließlich, welche Gegenstände und Tätigkeiten notwendig sind, um dieses Ziel zu erreichen und in welcher Weise dabei Gemeinsamkeiten mit und Unterschiede zu anderen Analysearten bestehen.

Anwendungssysteme sollen nicht lediglich den gegenwärtigen Zustand im Anwendungsbereich widerspiegeln, sondern auch Verbesserungswünsche der Anwender berücksichtigen. Daher ergänze ich in Abschnitt 3.1.4 die Ist-Analysen um eine Soll-Analyse, bevor ich die Einflüsse aus dem *Anwendungsbereich* mit ihren Gemeinsam-

[4] Musikwissenschaftliche Termini, von denen anzunehmen ist, dass sie einem großen Teil der Leser bereits bekannt sind, erläutere ich nicht unmittelbar im Text, sondern in einem im Anhang befindlichen Glossar musikwissenschaftlicher Begriffe.

keiten und Unterschieden in Abschnitt 3.1.5 in einer Kontextspezifikation zusammenfasse.

3.1.1 Thematische Analyse

Ein *Thema* ist eine aufgrund ihrer melodisch und meistens auch harmonisch und rhythmisch relativ festen Gestalt wiedererkennbare musikalische Einheit, die in einem größeren Zusammenhang prägend wirkt. Es kann aus mehreren Motiven bestehen, die ihrerseits zwar ebenfalls wiedererkennbar, aber nicht für sich genommen prägend sind.(Egg67, S. 950)[5] Nur wenn Motive außerhalb des Verbunds eines Themas auftreten, wächst ihnen die prägende Wirkung eines Themas zu, so dass sie dann ebenfalls Hauptgegenstand einer thematischen Analyse werden. Der Einfachheit halber verwende ich im Folgenden ausschließlich den Begriff „Thema“, wenn ich mich auf musikalisch prägende Einheiten beziehe.

Abgesehen von der Notwendigkeit, die Themen in irgendeiner Form zu wiederholen, hängt die Art und Weise, in der aus Themen musikalische Stücke aufgebaut werden, ausschließlich von der verwendeten Kompositionslehre ab.

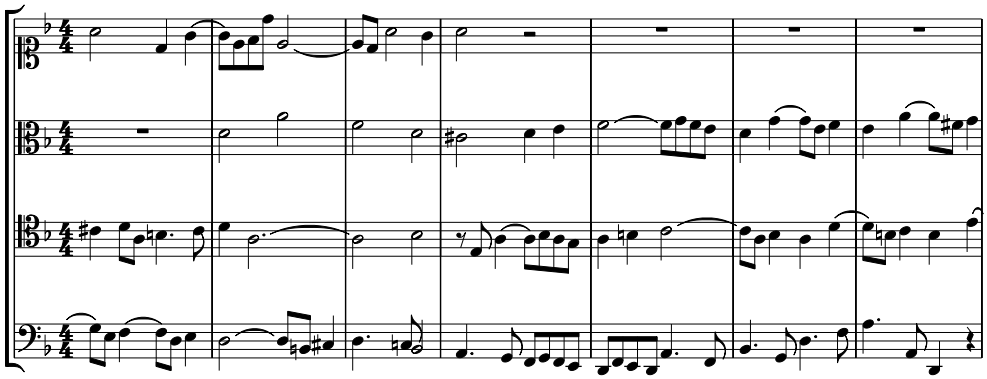


Abb. 3.1-1 Thema aus *Die Kunst der Fuge* von Johann Sebastian Bach

Es gibt eine Fülle von Möglichkeiten, um die Wiederkehr eines Themas zu gestalten.(Gra82) Neben der unveränderten Wiederholung sind dabei u.a. wesentlich:

- **Veränderung der Melodik.** Erweiterung oder Verringerung des Umfangs vom tiefsten zum höchsten Ton; Umkehrung.

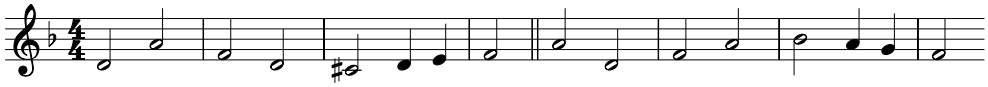


Abb. 3.1-2 Umkehrung des Themas.

Zu jedem →Halbtonschritt, den das ursprüngliche Thema aufwärts geht, geht die Umkehrung des Themas abwärts. Die Tondauern bleiben unverändert.

[5] Als Ergänzung zu den empirisch ermittelten Hintergründen der einzelnen Analysearten gebe ich in runden Klammern zusätzliche Verweise auf musikwissenschaftliche Literatur an, die zur Vertiefung des jeweiligen Gegenstands geeignet ist.

- **Veränderung der Rhythmik.** Gleichmäßige Vergrößerung oder Verkleinerung aller Notendauern um einen beliebigen Faktor.

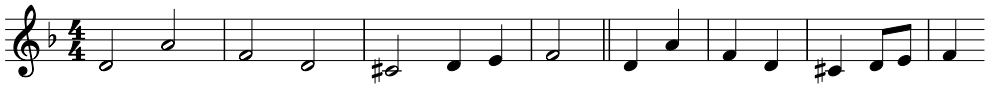


Abb. 3.1-3 Diminution (Verkleinerung) des Themas .

Alle Tondauern des ursprünglichen Themas sind um die Hälfte verringert. Aus halben Noten werden Viertelnoten und aus Viertelnoten Achtelnoten. Die Tonhöhen ändern sich nicht.

- **Kombination melodischer und rhythmischer Veränderungen.** Verlängerung durch Anhängen zusätzlicher Noten oder Verkürzung durch Auslassung eines Teils des Themas; Modifikation eines Teils der Tonfolge (z.B. Anfang und Ende gleichbleibend, Mittelteil neu); Steigerung oder Verringerung des melodischen Bewegungsimpulses; Steigerung oder Verringerung des rhythmischen Bewegungsimpulses; Kombination aus Umkehrung und Vergrößerung oder Verkleinerung.

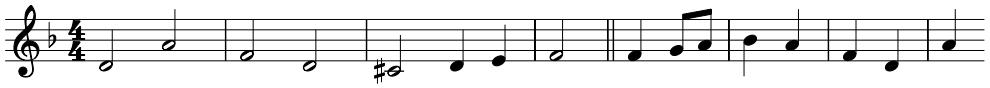


Abb. 3.1-4 Diminuierte, rückwärts gespielte Umkehrung des Themas.

- **Veränderung der Notation.** Obwohl abweichend und zum Teil „versteckt“ notierte Themen keine Änderungen des Hörerlebnisses nach sich ziehen, spielen sie eine wichtige Rolle bei der musikwissenschaftlichen Interpretation und Identifikation von Themen.(Sel98, S. 8ff) Ein Thema ist „versteckt“, wenn es nicht in der Form notiert ist, die der akustische Eindruck vermuten ließe. Einige Möglichkeiten dafür sind Vermischung mit anderem (ggf. thematischem) Material; Vermischung der melodischen, harmonischen und rhythmischen Komponenten; Verteilung von aufeinanderfolgenden Abschnitten auf verschiedene Stimmen; und Verteilung einzelner aufeinanderfolgender Noten auf verschiedene Stimmen.

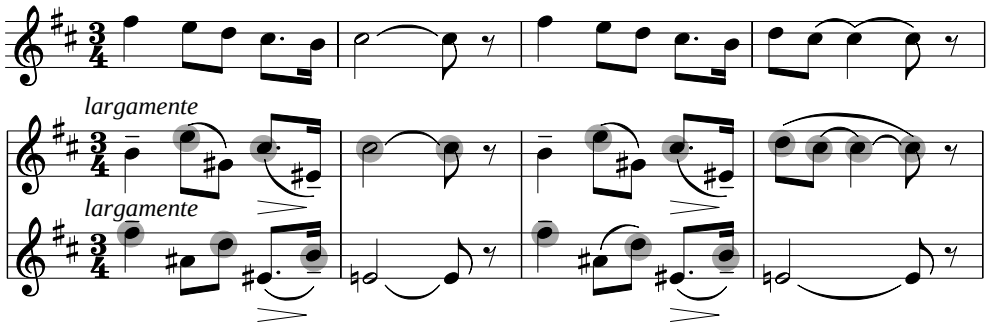


Abb. 3.1-5 Thema aus der 6. Symphonie von Peter Tschaikowsky

In der oberen Fassung ist das Thema ebenso notiert, wie es akustisch wahrgenommen wird. In der unteren Fassung vom Beginn des vierten Satzes ist der Klangeindruck derselbe, bezüglich der Notation ist es jedoch „versteckt“ und alterniert zwischen den 1. Violinen (oben) und den 2. Violinen (unten).

Die Kompositionslehren, die teilweise in großer Strenge vorschreiben, in welcher Weise Themen und deren Bestandteile zu behandeln sind, lassen sich danach unterscheiden, ob sie sich auf Reihungs- oder Entwicklungsformen beziehen:

- Bei **Reihungsformen** ändern sich die Themen nicht wesentlich, sondern werden in gleichbleibender Form aneinandergereiht. Einige Vertreter sind Liedformen (A, AB, ABA sowie A A'A B B'B A A'A) und komplexere Formen wie z.B. die Rondoform (ABACABA).

- Bei **Entwicklungsformen** kehren Themen außer in identischer Wiederholung in zahlreichen Formen melodischer und/oder rhythmischer Veränderungen wieder, wobei zusätzlich meist noch →Transponierungen auftreten. Wie genau einzelne Themen und vor allem mehrere Themen im Verhältnis zueinander entwickelt werden, bestimmt sich ausschließlich aus der Gattung des Musikstücks und der dafür üblichen Formenlehren und kann nicht generell angegeben werden. Entscheidend ist jedoch in jedem Fall, dass die Themen trotz aller Veränderungen erkennbar bleiben. Entwicklungsformen sind u.a. die zahlreichen →Fugenformen (Grundschema: ||: Durchführung Zwischenspiel :|| Durchführung), und die →Sonatenhauptsatzform (Grundschema: Exposition Durchführung Reprise Coda)

Musikhistorischer Hintergrund

Der Beginn der Beschäftigung mit komplexer Musiktheorie und Formenlehre(Gra82) fällt mit dem Übergang von musikalischer Einstimmigkeit zur Mehrstimmigkeit zusammen. Die Regeln des →Kontrapunkts des 14. Jahrhunderts schreiben vor, dass zwei Stimmen nur in bestimmter Weise zueinander gesetzt werden dürfen, aber sagen noch nichts über die Behandlung von Themen. Eine rein imitative (d.h. veränderungslose) Behandlung von Themen entwickelt sich im 15. Jahrhundert, wobei die Regeln dafür in der Form der →Fuge am strengsten sind. Regeln für die Gestaltung der →Harmonik und →Tonarten gehen auf Gioseffo Zarlino zurück, der in *Le istitutione harmoniche*(1558) erstmals wissenschaftlich untersucht, warum die bestehenden kontrapunktischen Regeln nicht ausreichen, um →Dissonanzen zu verhindern. Im Barock entfalten sich auf dieser Grundlage kurze Zeit später in Verbindung mit einer nicht bloß imitativen Behandlung von Themen frühe Ausprägungen von weiteren Entwicklungsformen wie der Suite, dem Konzert und der Sonate. In der Klassik werden diese Formen weiterentwickelt und aus einer Folge von Sonatensätzen entsteht die Symphonie, mit der die Komplexität der thematischen Behandlung einen Höhepunkt erreicht. Weniger strenge Regeln an die Art der thematischen Entwicklung stellen Reihungsformen wie die Variation und die Phantasie.

Analyseziele

Der thematischen Analyse geht es um die Aufdeckung und Vermittlung struktureller und inhaltlicher musikalischer Zusammenhänge. Bei der Einzelwerkanalyse wird nur ein Stück betrachtet, bei der vergleichenden Analyse können es Werke eines oder mehrerer Komponisten, einer oder mehrerer Gattungen, einer oder mehrerer Stilrichtungen sowie einer oder mehrerer Epochen sein.

Die **Einzelwerkanalyse** untersucht, welche Themen wo und in welcher Gestalt auftauchen, inwieweit die thematische Verarbeitung die formalen Kriterien der gewählten Theorie erfüllt und in welcher Weise der Komponist die ihm durch die Regeln gegebenen Freiräume nutzt. Die Interpretation dieser Übereinstimmungen, Abweichungen und

künstlerischen Ausgestaltungen hat sich im Laufe der Geschichte der musikwissenschaftlichen Analyse stark gewandelt. Alle Herangehensweisen sind dabei auch heute noch legitim, obwohl Vertreter anderer Methoden sie teilweise stark kritisieren.

- Die *strenge Interpretation* sieht jede Abweichung von den reinen Kompositionsregeln als Unzulänglichkeit oder sogar Fehler des Komponisten an. Bei „fehlerfreier" Satztechnik steht die Beurteilung der künstlerischen Ausgestaltung der Freiräume im Vordergrund.

  Die Ursprünge der strengen Interpretation gehen auf die bürgerlichen „Meisterwerk-Ästhetik" des 19. Jahrhunderts zurück. Damals begannen erstmals auch Nicht-Musiker, sich der eingehenden Analyse musikalischer Werke zu widmen. Wie auch bei der Musikerausbildung dienten die damaligen Kompositionslehren, deren Umsetzung in analytische Fragestellungen keiner Legitimation bedurfte, als Grundlage. Auch die konkreten (Meister-)Werke Beethovens und später auch Bachs dienten als Maßstäbe, an denen die nachfolgenden Kompositionen sich messen lassen mussten. Wichen sie ab, so wurden sie nicht ob ihrer Unterschiedlichkeit eingehender untersucht, sondern abqualifiziert.

- Die *inhaltsästhetische* Interpretation unterscheidet sich von der strengen Interpretation dadurch, dass sie Abweichungen von der reinen Kompositionslehre nicht unbedingt als Fehler ansieht, sondern prüft, ob der Komponist mit diesem formalen Bruch z.B. auf einen wie auch immer gearteten inhaltlichen Bruch hinweisen will. Am offensichtlichsten können solche Zusammenhänge bei Vokalwerken (z.B. Kantaten, Oratorien, Opern und Liedern) hergestellt werden, die einen expliziten Inhalt haben. Bei Instrumentalwerken muss der Interpretationsrahmen weiter gefasst werden.

Grundlage der **vergleichenden Analyse** sind mehrere Einzelwerkanalysen. Ausgehend von einem Vergleich der darin identifizierten Themen können sowohl Ähnlichkeiten innerhalb von bestimmten Gruppen von Werken als auch Unterschiede zwischen Werkgruppen etabliert werden. Für die Menge der prinzipiell unbegrenzten Vergleichsmöglichkeiten können u.a. die folgenden als Beispiele dienen:(Car67, Cop98)

- Beim *komponistenspezifischen Vergleich* stehen werkübergreifende Ähnlichkeiten im Vordergrund, die für einen bestimmten Komponisten typisch sind und somit als dessen Merkmale gelten können. Ein Komponist kann z.B. ein Thema in nahezu identischer Form in mehreren Werken einsetzen oder aber ein Thema über mehrere Werke hinweg in weiterentwickelter Form wiederaufgreifen. Auch ohne Wiederverwendung eines bestimmten Themas können Themen zahlreiche strukturelle Parallelen aufweisen, die oft die Ursache des speziellen Klangs eines Komponisten ausmachen.

- Dieses Vorgehen lässt sich auf *epochenspezifische Vergleiche* und *länderspezifische Vergleiche* ausdehnen, wobei es dementsprechend z.B. darum geht, teilweise bis zur sechsten oder siebten Note identische Themen als zeitabhängige Modeerscheinung zu deuten oder bestimmte Melodien und Rhythmen als landestypisch zu identifizieren.

Ein besonderer Analysegegenstand ist die Untersuchung *unveränderter Wiederverwendungen durch andere Komponisten*. Offensichtliche Formen sind hierbei die Variation – z.B. Brahms' *Variationen über ein Thema von J. Haydn* (op.56a) – oder die Parodie – z.B. Debussys Bezug auf das Vorspiel von Wagners *Tristan und Isolde* in *Golliwogg's Cake Walk*. Andere Formen der Wiederverwendung können entweder als bewusste Hommage gelten oder aber ein – bewusstes oder unbewusstes – Plagiat sein, dass Rückschlüsse über die eventuell unterbewusste Auseinandersetzung mit Werken anderer Komponisten zulässt.(BM48, S. xii)

Um das Ziel der Vermittlung der strukturellen und inhaltlichen Zusammenhänge eines thematisch strukturierten Werks bzw. mehrerer Werke zu erreichen, umfasst ein ideales Analyseergebnis die drei folgenden Elemente:

1. Die Grundlage bildet eine *annotierte Partitur*, in der alle Themen in allen Vorkommen aller Varianten markiert sind. Die melodischen, harmonischen und rhythmischen Bestandteile sind dabei gesondert hervorgehoben. Jedes Thema ist dabei mit einem möglichst aussagekräftigen Bezeichner versehen, anhand dessen es im

2. *thematischen Katalog* aufgefunden werden kann. Dieser Katalog enthält für jede Markierung eine separate Notiz, aus der alle für die jeweilige Analyseart relevanten Informationen über das Thema hervorgehen:

    - Ein komplettes Exzerpt der markierten Stimme. Das Thema erscheint darin in exakt der gleichen Weise wie in der annotierten Partitur mitsamt den Markierungen der einzelnen Themenbestandteile sowie eventuell vorhandenem Gesangstext.[6] Die Bezeichnung der Stimmen, Tonart und Taktart müssen ablesbar sein.

    - Eine geeignete Bezeichnung, anhand derer das Thema im Katalog eindeutig identifizierbar ist.

    - Eine Variantennummer und die Art der Variierung (siehe S. 26), sofern es sich nicht um die Urform handelt.[7]

    - Eine Nummer, die angibt, um das wievielte Vorkommen der Variante es sich handelt.

    - Der Ort des Auftretens mit zumindest Werkname und Taktnummer. Je nach Art des Werks ggf. auch mit Abschnitt und Unterabschnitt – z.B. Akt und Szene bei Opern. Die Taktnumerierung der Partitur wird beibehalten.

    - Der Grund des Auftretens aus formaler und/oder inhaltsästhetischer Perspektive.

[6] In der sich in Veröffentlichungen niederschlagenden Praxis werden Themen nahezu ausschließlich in der komprimierten Form des à Klavierauszugs dargestellt (vgl. [Wol96], [Win], [BM48], [BM66], [Par75]). Information über die Stimmenführung und Instrumentation gehen dadurch für auf den Katalog aufbauende Analysen verloren. Die Konsequenzen für die Aussagekraft der Analyse und die sich daraus ergebenden Verbesserungswünsche an eine rechnergestützte Analyseumgebung diskutiere ich im Rahmen der Soll-Analyse in Abschnitt 3.1.4.

[7] Auch hier beschränken sich existierende Kataloge aus ökonomischen Gründen meist lediglich auf die Urform.

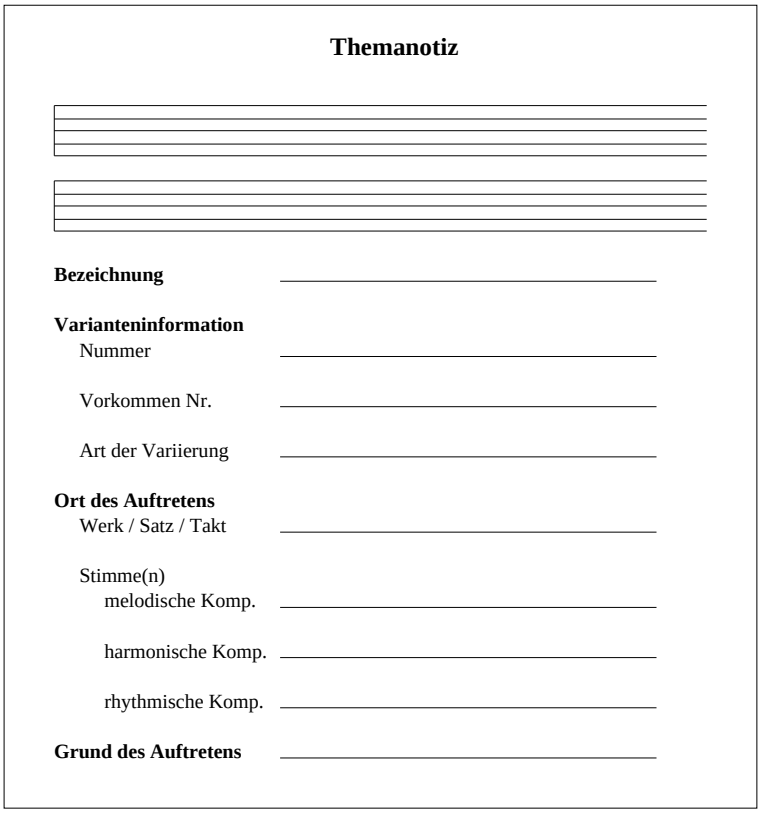


Abb. 3.1-6 Notizzettel für die manuell durchgeführte thematische Analyse.

Das Format und die Aufteilung der Einträge können von Analyse zu Analyse voneinander abweichen. Dieser Notizzettel mit seinen zwei Systemen von je 8 cm Länge und zwei Zeilen für Bezeichnungen stellt eine gute Ausgangsbasis für speziell angepasste Notizzettel dar. So ist es beispielsweise für komplexe Orchesterwerke vorteilhaft, weitere Systeme hinzuzufügen.

Sowohl die annotierte Partitur als auch der thematische Katalog sind zu voluminös, um sie als Analyseergebnis zu veröffentlichen. Sie bilden daher eine oft unveröffentlichte, interne Arbeitsgrundlage, auf deren Basis der Musikwissenschaftler

3. einen *erklärenden Text* verfasst, in dem er zunächst die Ziele und die Methode seiner Analyse darlegt und dann die in der annotierten Partitur und dem thematischen Katalog manifestierten Ergebnisse so aufbereitet, dass seine Ansichten über den musikalischen Zusammenhang des Werks deutlich werden.

Konkretes Vorgehen

Bevor das konkrete Vorgehen im Rahmen einer Ist-Analyse ermittelt werden kann, muss zunächst abgegrenzt werden, welche Teile des betrachteten *Anwendungsbereichs* zwecks Erstellung eines Anwendungssystems überhaupt analysiert werden sollen und welche nicht. Neben einer rein ökonomischen Abgrenzung spielt dabei auch die Realisierbarkeit einer adäquaten Rechnerunterstützung eine wesentliche Rolle (vgl. [FZ99, S. 775]). In Zusammenarbeit mit Musikwissenschaftlern hat sich dabei ergeben, dass die Erstellung des erklärenden Texts – ebenso wie das Schreiben einer Novelle oder die Komposition eines Musikstücks – ein so hohes Maß an Kreativität erfordert, dass eine Unterstützung durch anwendungsspezifische Software als unmöglich

angesehen wird. Anwendungsinvariante Standardsoftware wie Textverarbeitungen und Notensatzprogramme können lediglich dabei helfen, die Ergebnisse des kreativen Akts festzuhalten. Inhaltlich unterstützen können sie die Anwender jedoch nicht.[8] In der folgenden Darstellung der Tätigkeiten bei der thematischen Analyse und im Rest dieser Arbeit betrachte ich daher nur diejenigen Tätigkeiten und Gegenstände, die dazu notwendig sind, um eine Partitur gemäß der gewählten Analyseart zu annotieren und die gefunden Strukturelemente zu katalogisieren.

Neben dieser quantitativen Einschränkung des Analysegegenstandes ergibt sich zusätzlich eine weitere qualitative Einschränkung hinsichtlich der Beschreibung des Anwendungsbereichs: Sämtliche zu Beginn dieses Kapitels diskutierten Ist-Analyse-Ansätze dienen primär dazu, Tätigkeiten eines solchen Komplexitätsgrades zu beschreiben, wie sie an einem typischen Sachbearbeiterplatz anfallen (vgl. [WAM98, S. 5]). Die Motivation und Entscheidungsprozesse der Anwender sind dabei zwar bereits zu komplex, um sie vollständig niederschreiben (und damit automatisieren) zu können, jedoch ist das notwendige Fachwissen noch so überschaubar, dass es im Rahmen der Ist-Analyse-Dokumente in einigem Detail darstellbar ist. Im Fall der musikwissenschaftlichen Analyse ist dementgegen ein Maß an Fachwissen erforderlich, das erheblich über dasjenige eines Sachbearbeiters hinausgeht. Wie ich bereits in [Kor96a] dargelegt habe und auch im bisherigen Teil dieses Abschnitts zeigen konnte, ist es so komplex, dass es den Rahmen der Ist-Analyse-Dokumente sprengte, wenn versucht würde, es möglichst vollkommen niederzuschreiben. Aus diesem Grund beschränken sich sämtliche Tätigkeitsbeschreibungen dieser Arbeit auf die sichtbaren Tätigkeiten sowie die dazu benötigten Arbeitsmittel.[9]

Auf der Ebene der Tätigkeiten ergibt sich nach diesen umfänglichen und qualitativen Einschränkungen die folgende Beschreibung für die thematische Analyse:

Tätigkeiten bei der thematischen Einzelwerkanalyse

Der Analysator ***sucht*** anhand selbst verfertigter oder existierender thematischer Kataloge Themen in der Partitur. Ein gefundenes Thema wird ***markiert*** und ausführlich ***notiert***. Der Analysevorgang wird so lange fortgeführt, bis alle erkannten Vorkommen aller Varianten aller Themen sowohl in der Partitur als auch auf Themanotizzetteln vermerkt sind, die zusammengenommen einen thematischen Katalog bilden. Darauf aufbauend wird dann der erklärende Text verfasst, der die Zusammenhänge aufzeigt.

Für die ***Suche*** hört sich der Analysator wenn möglich eine Tonaufzeichnung der zu untersuchenden Partiturstelle an und liest dabei in der Partitur mit.

Daraufhin geht er in der Partitur zum Beginn der zu untersuchenden Stelle zurück und nimmt die existierenden und/oder die selbst verfertigten thematischen Kataloge zur Hand. Er geht vom Anfang her beginnend durch die Partitur und sucht Themen. Erkennen kann er ein Thema trotz dessen möglicherweise starken Veränderung und/oder „versteckten" Notation entweder anhand seiner Erinnerung an die Tonwiedergabe bzw. durch erneutes Hören der betreffenden Stelle oder anhand der Gestalt des Themas, die er aus vorangegangenen Analysesitzungen her kennt.

Anhand von Vergleichen mit Einträgen der thematischen Kataloge kann er feststellen, ob das Thema bereits dokumentiert ist und im positiven Fall die Bezeichnung aus dem Katalog entnehmen. Falls es sich um ein bisher undokumentiertes Thema handelt, so nimmt er selbst eine der strukturellen oder inhaltsästhetischen Bedeutung angemessene Benennung vor.

[8] Zu den Grenzen des Rechnereinsatzes siehe z.B. [DG99, S. 976f]

[9] Mit der Frage, wie solche komplexen Tätigkeiten auf geeignete Weise durch Software unterstützt werden können, beschäftige ich mich im Rahmen des Kontexts *Handhabung & Präsentation* in Abschnitt 3.2.

Die ***Markierung*** eines erkannten Themas umfasst die gesonderte Kennzeichnung der melodischen, der harmonischen und der rhythmischen Komponenten in der Partitur (sofern diese vorhanden sind).

Mit der ***Anfertigung einer thematischen Notiz*** wird der eigene thematische Katalog ergänzt. Der Analysator nimmt dazu einen leeren Themanotizzettel zur Hand und füllt ihn in beliebiger Reihenfolge aus.

Tätigkeiten bei der vergleichenden thematischen Analyse

Für die vergleichende Analyse lässt sich aufgrund der unbegrenzten Anzahl der Vergleichsmöglichkeiten keine prinzipielle Vorgehensweise beschrieben. Als Vorbereitung empfiehlt sich aber eine Indizierung der verwendeten thematischen Kataloge entsprechend den Analysezielen, um das zu vergleichende Material rascher zu finden.

In den folgenden Abschnitten 3.1.2 und 3.1.3 werde ich diese Tätigkeiten im Rahmen der thematischen Analyse mit denjenigen im Rahmen der leitmotivischen und der Set-Theory-Analyse vergleichen und herausarbeiten, inwiefern die dabei identifizierten Gemeinsamkeiten und Unterschiede geeignet sind, um für alle drei Anwendungsbereiche anstelle von drei separaten Anwendungssystemen eine einzige, anpassbare Anwendungsfamilie zu entwickeln.

3.1.2 Leitmotivische Analyse

Ebenso wie ein *Thema* ist auch ein *Leitmotiv* eine aufgrund ihrer melodisch und oft auch harmonisch und rhythmisch relativ festen Gestalt wiedererkennbare musikalische Einheit, die in einem größeren Zusammenhang prägend wirkt. Im Gegensatz zum Thema bezieht es sich aber dauerhaft auf ein bestimmtes Element der Handlung eines dramatischen Vokalwerks wie z.B. eine Person, einen Gegenstand, ein Gefühl oder eine Idee. Der Einsatz und die musikalische Entwicklung eines Leitmotivs orientieren sich ausschließlich am Fortgang der dramatischen Ereignisse. Je nachdem wie sich inhaltliche Bedeutungszuweisung und musikalische Realisation zeitlich zueinander verhalten, fungiert ein Leitmotiv als eine Ahnung (neues Leitmotiv ohne sofortige dramatische Ausdeutung), Vergegenwärtigung (ahnend eingeführtes oder neues Leitmotiv mit gleichzeitiger dramatischer Ausdeutung) oder Erinnerung (bekanntes Leitmotiv mit sofortiger dramatischer Ausdeutung). Erinnerungen können das auf der Bühne Gesagte dabei bestätigen, aber vor allem auch nicht Gesagtes oder in Wirklichkeit anders gemeintes psychologisch ausdeuten. So wie die dramatischen Handlungselemente nicht beziehungslos nebeneinanderstehen, so sind auch die dazugehörigen Leitmotive miteinander verflochten.

Ebenso wie Themen können Leitmotive variiert werden und neue Leitmotive durch Variation aus bereits eingeführten Leitmotiven hervorgehen.(Dal86a) Im Gegensatz zu Themen ergeben sich diese Variationen aber aus der Handlung. Ein Wechsel der Instrumentation eines auf einen Gegenstand bezogenen Leitmotivs kann z.B. mit dessen neuer Verwendung einhergehen. Übernimmt eine Person die Harmonisierung einer anderen, so kann dies auf persönlichen Einklang hindeuten. Wird ein auf eine Idee bezogenes Leitmotiv dissonant harmonisiert, so kann dies von einer Pervertierung der ursprünglichen Zielvorstellung herrühren. Taucht ein neues Leitmotiv auf, das die musikalischen Umkehrung eines bereist eingeführten Leitmotivs darstellt, dann repräsentiert es ein Konzept, das dem Konzept des ersten Leitmotivs entgegensteht.

Selbst große nach thematischen Gesichtspunkten komponierte Werke besitzen nur eine Handvoll von Themen. Im Gegensatz dazu hängt die Anzahl der Leitmotive in einem Werk ausschließlich von dessen dramatischer Komplexität ab. Da das Mittel der Variation sowohl zur Ausdeutung bestehender Leitmotive als auch zur Schaffung neuer Leitmotive dient, darf es angesichts der potentiell großen Anzahl von Leitmotiven nur so eingesetzt werden, dass es der sofortigen Wiedererkennbarkeit nicht entgegensteht. Möglich sind u.a.

- **Veränderung eines einzelnen Leitmotivs.** Zusätzlich zu den auch für Themen mögliche Veränderungsarten (siehe Seite 26) spielen vor allem der Wechsel in ein anderes Instrument, das Hinzufügen oder die Änderung der Harmonisierung, sowie die Einführung von Dissonanzen eine besondere Rolle.

- **Kombination mehrerer Leitmotive.** Dem dramatischen Inhalt folgend können sich Leitmotive aneinanderreihen, einander umklammern und auf verschiedene Arten miteinander verschmelzen. Die Kombination kann sich dabei nur auf einen, mehrere, oder alle Bestandteil beziehen (melodisch, harmonisch, rhythmisch).

Wird ein musikalisches Drama ausschließlich durch ein dichtes Gewebe von Leitmotiven von innen heraus zusammengehalten und nicht nur als Beiwerk unter Beibehaltung eines anderen Kompositionsverfahrens eingesetzt, so spricht man von *Leitmotivtechnik*. Die verschiedenen auf Leitmotive abzielende Analysearten lassen sich nur aus dem musikhistorischen Kontext heraus begreiflich machen.

Musikhistorischer Hintergrund

Musiktheoretisch ist die Oper in ihren Anfängen des 17. Jahrhunderts eher einfach. Ein existierendes Drama wird von einem Librettisten so umgearbeitet, dass es sich in Rezitative und in Strophen aufgeteilte Arien gliedern lässt. In den Rezitativen findet die eigentliche Handlung statt, während in den Arien die Gefühle der Akteure beleuchtet werden. Während die Rezitative unvertont bleiben und wie ein Theaterstück gegeben werden, komponiert ein meist unabhängig vom Librettisten arbeitender Komponist Lieder für die Arien. Da diese Lieder durch die Rezitative voneinander getrennt sind, bestehen meistens auch keine musikalischen Querbezüge. In seltenen Fällen werden Melodien als Erinnerungsmotiv wieder aufgegriffen, um einen inhaltlichen Zusammenhang herzustellen.

In der Aufführungspraxis werden das Drama und die Rezitative zunehmend als lästiges Beiwerk behandelt, das möglichst schnell abzuhandeln ist, um möglichst bald wieder die Virtuosität der Sänger zum Zuge kommen zu lassen. Die Sänger spielen während der Arien nicht weiter, sondern begeben sich zum Bühnenrand und singen von dort aus direkt ins Publikum. Berühmten Sängern werden Arien sogar „auf den Leib“ komponiert.

Neben theoretischen Appellen zur Wiederherstellung des Gleichgewichts von Drama und Musik wie z.B. durch Christoph Willibald Gluck ändert sich am spärlichen Theoriegebäude wenig, außer dass Komponisten das Rezitativ zu Beginn des 19. Jahrhunderts zunehmend auch melodiös ausgestalten und orchestral untermalen.

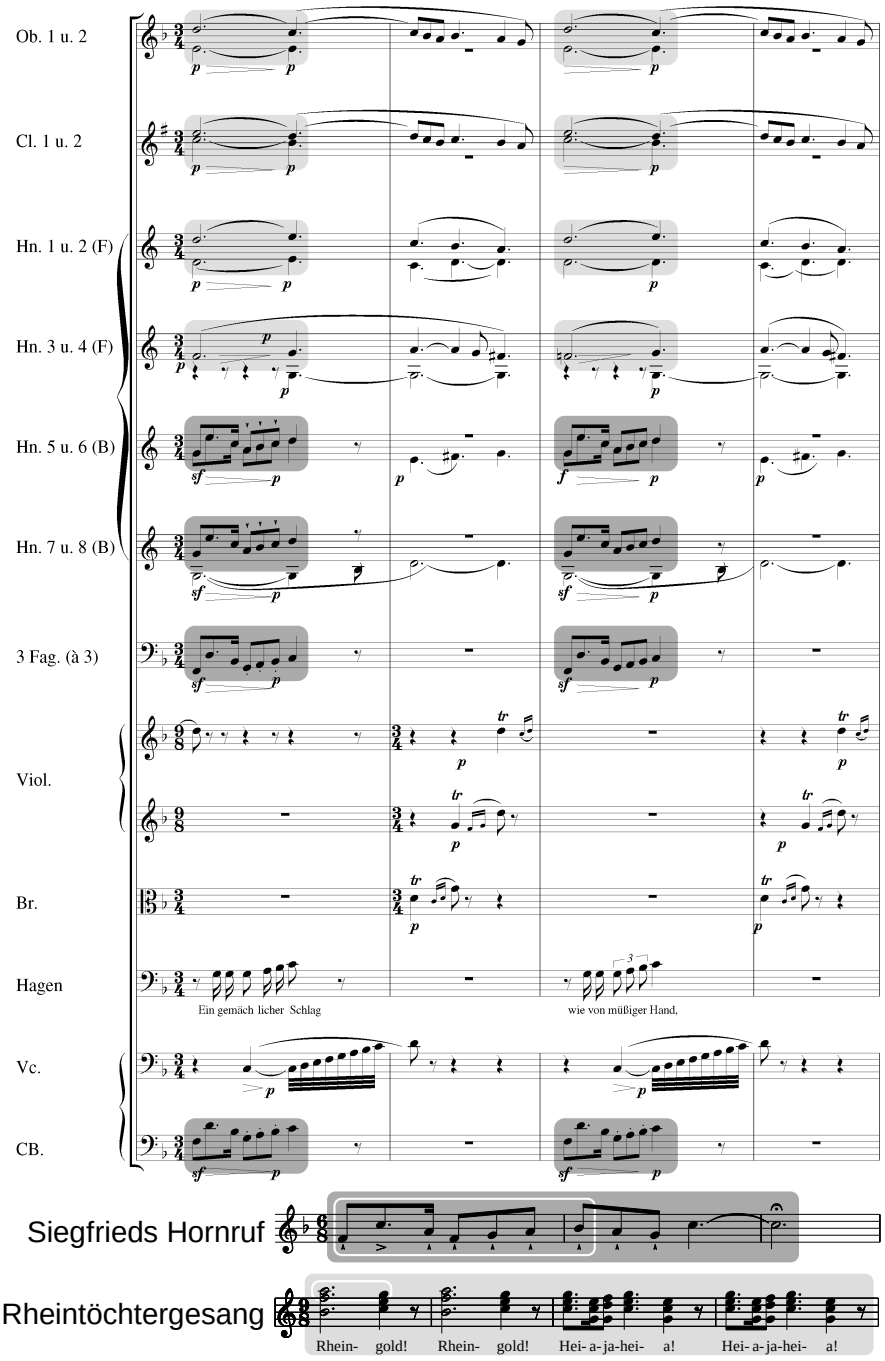


Abb. 3.1-7 Die Verbindung der Leitmotive für Siegfrieds Hornruf (oben) und den Gesang der Rheintöchtergesang (unten) deutet in Szene 1 des 1. Aktes von *Götterdämmerung* an, dass es sich bei dem im Boot Vorbeifahrenden um Siegfried handelt, noch bevor dies mit Worten ausgedrückt wurde.

Beide Leitmotive tauchen nicht in ihrer Originalform auf, sondern sind auf ihre charakteristischen Minima verkürzt (durch helle Umrandung hervorgehoben).

Erst Richard Wagner widmet sich in der Mitte des 19. Jahrhunderts massiv der Operntheorie und entwirft in zahlreichen theoretischen Schriften wie *Oper und Drama*(1852, Wag83) eine neue Synthese von Drama und Musik, in der er die Musik dem Drama unterordnet. Musik solle nicht nur der bloßen Gefälligkeit halber in dramatisch nicht rechtfertigbaren Arien auftauchen, sondern als permanente gefühlsmäßige Ausdeutung der sich gleichmäßig gefühls- und verstandesorientiert entfaltenden Handlung

dienen. Die einfache Form der Oper weicht sowohl im Großen (Arie-Rezitativ-Schema) als auch im Kleinen (Liedform der Arien) einer durchgängigen als Gefühlswegweiser fungierenden symphonischen Orchestermelodie, deren Struktur sich aber nicht wie bei der Symphonie an festen Entwicklungsformen, sondern am Drama orientiert. Das Mittel der Gefühlsvermittlung sind die Leitmotive, die Kompositionstechnik ist die Leitmotivtechnik. Für die Aufführungspraxis fordert Wagner dementsprechend echtes und durchgängiges Schauspiel der Sänger, ein Ende des Deklamierens vom Bühnenrand aus und zahlreiche andere Maßnahmen, die die Konzentration des Publikums auf das *ganze* Geschehen unterstützen sollen. In seinen nachfolgenden Opern *Tristan und Isolde*(1857 Komposition begonnen, 1865 uraufgeführt), *Die Meistersinger von Nürnberg*(1862, 1868), der aus *Das Rheingold*, *Die Walküre*, *Siegfried* und *Götterdämmerung* bestehenden Operntetralogie *Der Ring des Nibelungen*(1853, 1876) und *Parsifal*(1877, 1882) sowie in dem nach seinen Plänen errichteten Bayreuther Festspielhaus setzt er seine Ideen auch tatsächlich um und beeinflusst dadurch nahezu alle zeitgenössischen und nachfolgenden Komponisten, obwohl ihm nicht alle vollständig folgen.(Bau88)

Analyseziele

Wie auch bei der thematischen Analyse geht es der leitmotivischen Analyse um die Aufdeckung und Vermittlung inhaltlicher und struktureller musikalischer Zusammenhänge. Bei der Einzelwerkanalyse wird nur ein Stück betrachtet, bei der vergleichenden Analyse können es Werke eines oder mehrerer Komponisten sein.

Die **Einzelwerkanalyse** untersucht, welche Leitmotive wo und in welcher Gestalt auftauchen, inwieweit deren Verwendung die Kriterien der gewählten Theorie erfüllt und in welcher Weise der Komponist die Leitmotive miteinander verflicht. Die Frage, welche Theorie dabei zu wählen sei, war unter Musikwissenschaftlern lange Zeit umstritten. Der in Abschnitt 3.1.1 beschriebene Konflikt zwischen der Legitimation anhand traditioneller versus inhaltsästhetischer Interpretation fand im Streit um den angemessenen Umgang mit Wagners Opern einen ersten Kulminationspunkt.(Fin97, S. 582)

- Die *traditionelle Interpretation* bezieht sich auf Wagners Erwähnung des symphonischen Stils und anderer traditioneller Entwicklungsformen und legt daher an leitmotivische Verarbeitung die Maßstäbe der thematischen Verarbeitung an. Alles, was nicht den Kriterien der strengen Interpretation der thematischen Einzelwerkanalyse genügt (siehe S. 29), wird als Unzulänglichkeit oder Fehler des Komponisten ausgelegt. Je nach gewünschtem Urteil gibt es zwei Anwendungsvarianten.

  1. Die hauptsächlich von Wagners zahlreichen Kritikern angewendete *streng traditionelle Interpretation* wendet die Regeln unmodifiziert an. Da leitmotivisch komponierte Opern nur in dramatisch motivierten Ausnahmefällen auf diese Regeln zurückgreifen, können sie größtenteils als formloses Diletantentum abqualifiziert werden.

  2. Einige von Wagners Anhängern versuchen mit einer *dehnbaren traditionellen Interpretation* die Formregeln so weit zu flexibilisieren, dass leitmotivisch komponierte Werke darin Platz finden. So wurde beispielsweise *Der Ring des Nibe-*

lungen als viersätzige Symphonie gedeutet (*Das Rheingold*: Allegro, *Die Walküre*: Adagio appasionato, *Siegfried*: Scherzo mit Trio, *Götterdämmerung*: Finale mit Rückbeziehung) oder auch nur als ein einzelner Sonatenhauptsatz (*Das Rheingold*: Exposition, *Die Walküre* und *Siegfried*: Durchführung, *Götterdämmerung*: freie Reprise). Höhepunkt dieser Interpretationsvariante ist Alfred Lorenz' Schrift *Das Geheimnis der Form bei Richard Wagner. I. Band: Der musikalische Aufbau des Bühnenfestspiels ‚Der Ring des Nibelungen'*(Lor24), in der er das Werk auf einfache Reihungsformen (siehe S. 28) zurückführt, deren Perioden aber in der Länge zwischen annehmbaren 14 und grotesken 840 Takten schwanken.(Bau88)

- Die *inhaltsästhetische Interpretation* versucht, die musikalische Form gemäß Wagners theoretischen Schriften – und nicht nur selektiven Ausschnitten daraus – vom Inhalt her zu erschließen. Die Bedeutung, das Vorkommen, die Veränderung und die Verbindung von Leitmotiven im Kontext des Dramas stehen daher im Vordergrund. Formale Melodik, Harmonik und Rhythmik werden zwar zur Erklärung herangezogen, nicht aber im Sinne einer Richtschnur zum Urteilen über „Richtig" und „Falsch" verwendet.

Die **vergleichende leitmotivische Analyse** verfolgt ähnliche Ziele wie die vergleichende thematische Analyse, d.h. es geht ihr darum, Ähnlichkeiten und Unterschiede innerhalb einer Gruppe von Werken aufzudecken. Allein die epochenspezifischen Vergleiche sind nur eingeschränkt möglich, da die Leitmotivtechnik im Gegensatz zu thematischen Techniken nur auf eine knapp 150jährige Tradition zurückblicken kann.

Trotz des abweichenden theoretischen Hintergrunds, besteht das Ergebnis einer idealen leitmotivischen Analyse ebenso wie bei das ideale Ergebnis einer thematischen Analyse aus einer *annotierte Partitur*, einem *Katalog*, und einem *erklärenden Text*, der die Zusammenhänge aufzeigt (siehe S. 30). Bezüglich dieser Arbeitsmittel bestehen nur zwei Arten von Unterschieden:

(1) **Unterschiede in der Benennung.** Abgesehen davon, dass die untersuchten Strukturelemente natürlich Leitmotive anstatt Themen sind, ist der einzige Unterschied die Benennung der Kataloge. Bei der thematischen Analyse heißen sie „thematische Kataloge", bei der leitmotivischen Analyse „Leitmotivtabellen".

(2) **Unterschiede in der Gewichtung.** Unter „Grund des Auftretens" wird im Gegensatz zur thematischen Analyse bei der leitmotivischen Analyse primär Inhaltsästhetisches und erst nachrangig Strukturelles auf dem Notizzettel eingetragen. Besondere Bedeutung kommt außerdem der Bezeichnung der Leitmotive zu. Die Leitmotivnamen dienen nicht alleine der Identifikation in der Leitmotivtabelle, sondern sind selbst Gegenstand von musikwissenschaftlichen Diskursen (vgl. [Coo68, S. 24]). Je nach Quellenlage kann hierbei auf vom Komponisten selbst eingeführte Namen oder auf Sekundärquellen zurückgegriffen werden. Zusätzlich zu diesen Namen kann der Analysator auch selbstkreierte Namen verwenden, falls die Literatur keinen Namen für das Thema kennt oder er mit diesen Bezeichnungen nicht konform geht.

Konkretes Vorgehen

Entsprechend dieser strukturellen Ähnlichkeit der Idealergebnisse einer leitmotivischen und einer thematischen Analyse stellte sich im Rahmen einer Ist-Analyse der konkreten Analysetätigkeiten heraus, dass auch hier außer den bereits beobachteten Unterschieden in der Benennung und der Gewichtung keinerlei Abweichungen im Vorgehen verglichen mit dem Vorgehen bei der thematischen Analyse festzustellen sind (siehe S. 32). Dieses verblüffende Maß an Übereinstimmung der Tätigkeitsbeschreibung lässt sich vollständig durch die in Abschnitt 3.1.1 diskutierten quantitativen und qualitativen Einschränkungen erklären, denen Ist-Analysen im Bereich der Musikwissenschaft unterworfen sind (siehe S. 31):

- Die erheblichen musiktheoretischen Unterschiede zwischen den beiden Analysearten kommen erst im Rahmen der Rechtfertigung des Analysestandpunkts und der Interpretation der thematischen bzw. leitmotivischen Struktur im erklärenden Text voll zum Tragen. Diese Unterschiede kommen in der Ist-Analyse der Tätigkeiten jedoch nicht zum Tragen, da die Erstellung des erklärenden Texts aufgrund seiner äußerst kreativen Natur außerhalb des durch Rechner unterstützbaren Bereichs liegt und somit gar nicht erst Gegenstand einer Ist-Analyse ist.

- Während der Analyse der Partitur manifestieren sich die musiktheoretischen Unterschiede lediglich im Hintergrundwissen des Musikwissenschaftlers, vor dessen Hintergrund er darüber entscheidet, welche Elemente er als zusammenhängende Themen bzw. Leitmotive oder deren Variationen ansieht. Da die Ist-Analyse jedoch lediglich auf die sichtbaren Arbeitsmittel und Tätigkeiten fokussiert, schlagen sich auch diese Unterschiede nicht in der konkreten Beschreibung der Tätigkeiten nieder.

Aufgrund der Übereinstimmungen bezüglich der Arbeitsmittel und Vorgehensweise bei der leitmotivischen und der thematischen Analyse erscheint es denkbar, beide Tätigkeiten durch Analysesysteme zu realisieren, die Mitglied ein und derselben Anwendungsfamilie sind, deren Kernsystem gemäß den beobachteten Unterschieden in der Benennung und der Gewichtung anpassbar ist. Für solch ein softwaretechnisches Urteil können die hier festgestellten Gemeinsamkeiten im *Anwendungsbereich* natürlich nur ein Ausgangspunkt sein, von dem aus in den Kontexten *Handhabung & Präsentation* und *verwendete Technik* weitere Untersuchungen angestellt werden müssen (siehe Abschnitt 3.2 und 3.3).

Bevor ich mich aber diesen Kontexten zuwende, betrachte ich zunächst mit der Set-Theory-Analyse noch einen dritten, stärker abweichenden *Anwendungsbereich*, um eine breitere fachliche Basis für die zukünftige Anwendungsfamilie zu schaffen.

3.1.3 Set-Theory-Analyse

Ein *Set* ist eine aus bis zu 12 sich nicht wiederholender →Pitch Classes bestehende zusammenhängende Tongruppe, die im größeren Zusammenhang eines atonalen Musikstücks prägend wirkt. Gemäß Allen Fortes *Set Theory*(For77) ist es für die strukturelle Bedeutung eines Sets unerheblich, in welcher Reihenfolge die Pitch Classes angeordnet sind, oder ob das Set transponiert oder umgekehrt wird. Dadurch reduziert Forte den Analysegegenstand der Set Theory von einer unüberschaubar großen

Anzahl kombinatorisch möglicher Sets auf nur 224 Äquivalenzklassen von Sets, die er *Set Classes* nennt. Da aufgrund der vielfältigen Veränderungsmöglichkeiten nicht sämtliche Mitglieder einer Set Class unbedingt auf den ersten Blick erkennbar sind, kann in einem rein mathematischen Verfahren zu jedem Set dessen *Prime Form* ermittelt werden, die es mit allen anderen Mitgliedern der Set Class teilt. Um zur Identifikation von Set Classes nicht jedesmal die komplette Prime Form aufzählen zu müssen – immerhin bis zu 12 Pitch Classes – kann zu jeder der 224 Prime Forms eine Kurzbezeichnung aus einer Tabelle nachgeschlagen werden. Die Set Class mit der Prime Form 02479 hat z.B. die Bezeichnung „5-35“, wobei „5“ für die Länge und „35“ für die laufende Nummer innerhalb der Gruppe derjenigen Set Classes steht, die fünf Pitch Classes lang sind.

Ein Set kann in einer beliebigen Gestalt in einem Werk auftauchen, solange es zusammenhängend ist. Dabei spielt es keine Rolle, ob die Pitch Classes eindimensional horizontal, eindimensional vertikal oder zweidimensional (also horizontal und vertikal verteilt) angeordnet sind.

Die Set Theory selbst macht keine Vorgaben, in welcher Form ein Werk aus Sets zu komponieren ist. Bezüglich der Auswahl der Sets wird aber gefordert, dass es im Werk möglichst genau eine ausgezeichnete Set Class mit der Bezeichnung *Nexus*[10] *Set* geben soll, mit der alle anderen Set Classes des Stücks in abstrakter Form verbunden sind. Die zu diesem Zweck rein mathematisch definierten Relationen K und Kh (eher weiter bzw. eher enger gefasste Form der Verbundenheit) beruhen ihrerseits auf einer weiteren Relation, die besagt, dass eine Set Class X als „abstrakt enthalten“ in Y gilt, genau dann wenn mindestens ein Mitglied von X identisch mit der Prime Form von Y ist.

Wie auch bei der thematische und leitmotivische Analyse bildet sich ein klareres Bild von der Set-Theory-Analyse, wenn sie wie im Folgenden in einen musikhistorischen Kontext gestellt wird.

Musikhistorischer Hintergrund

Nachdem Wagner in *Tristan und Isolde* aus dramatischen Gründen als erster Komponist im großen Umfang die Grenzen der strengen Tonalität überschreitet, beginnt eine Entwicklung zu tonaler Mehrdeutigkeit, die schließlich zu Beginn des 20. Jahrhunderts in völliger Bezugslosigkeit zu tonalen Zentren ihren Abschluss findet.(Gra82) An Stelle der traditionellen Themen treten eine Vielzahl verschiedener Strukturelemente, die je nach Theoretiker bzw. Komponist formal mehr oder wenig ähnlich wie Themen behandelt werden können (siehe S. 26). Die von Schönberg anfänglich verwendete Technik der Reihen aus 12 Pitch Classes wird im sich anschließenden allgemeinen Serialismus noch weiter gefasst, so dass Reihen nicht nur aus Pitch Classes, sondern aus allen musikalischen Parametern wie z.B. Tondauern, Lautstärken, Oktavlagen oder Klangfarben gebildet werden können. Um den Behauptungen entgegenzutreten, die Vielzahl der neuen Kompositionstheorien bedeuteten den Verlust einer allgemein verbindlichen Tonsprache, entwickelt Allen Forte aufbauend auf Milton Babbits Arbeiten der 1950er Jahre die Set Theory, die mit Sets ein Strukturelement propagiert, dass auf alle atonalen Kompositionen anwendbar ist.

[10] lat.: Zusammenhang, Verbindung, Verflechtung [Dud90, S. 533]

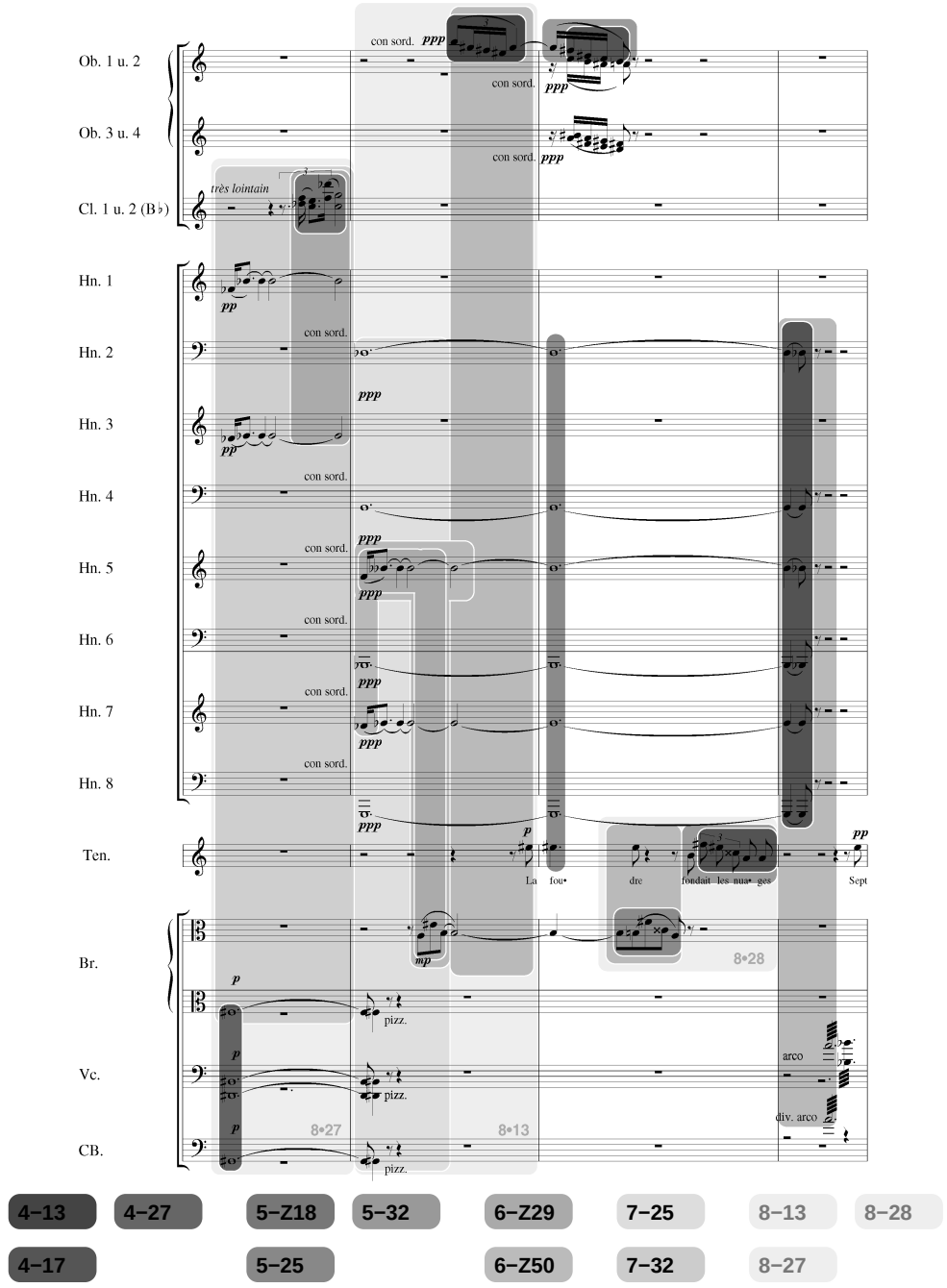


Abb. 3.1-8 Mögliche Set-Segmentierung einer Passage aus Igor Strawinskys *Le Roi des Étoiles* (Takt 42 bis 45) (vgl. [For77, S. 115]).

Die Sets sind gemäß der *streng strukturellen Forte'schen Analyse* bezeichnet (siehe S. 41). Die Nummer vor dem Bindestrich gibt die Kardinalität des Sets an, die Nummer danach die Reihenfolge innerhalb der Set-Gruppe gleicher Kardinalität. Sets können einander überlagern, so dass Sets großer Kardinalität teilweise Sets kleiner Kardinalität enthalten.

Analyseziele

Der Set-Theory-Analyse geht es – wie den beiden anderen vorgestellten Analysearten – um die Aufdeckung und Vermittlung struktureller und inhaltlicher musikalischer Zusammenhänge innerhalb eines Musikstücks. Insbesondere wird untersucht, welche Set Classes wo und in welcher Gestalt auftauchen, inwieweit deren Verwendung den theoretischen Kriterien der Set Theory genügt und in welcher Weise

der Komponist die ihm durch die Theorie gegebenen erheblichen Freiräume nutzt. Der Grad der Anwendung der Set Theory ist dabei heftig umstritten, so dass das Analyseziel und -ergebnis stark davon abhängt, inwieweit den Forte'schen Vorschlägen gefolgt wird.(Cas94)

- Bei der *streng strukturellen Forte'schen Analyse* geht es darum, das Nexus Set einer Komposition zu finden. Dazu werden alle Sets identifiziert, deren Set Classes ermittelt und in Übereinstimmung mit Fortes Forderung das Nexus Set gesucht. Ein Analyseergebnisses, das zu dem Schluss kommt, dass ein Werk mehr als ein Nexus Set enthält, lässt bei der strengen Analyse nur zwei Schlussfolgerungen zu: Entweder ist die Komposition einfach „weniger gut" oder der Analysator hat die Set Classes „falsch" identifiziert, so dass er das in Wirklichkeit vorhandene Nexus Set nicht finden konnte. „Weniger gute" Kompositionen können qualitativ noch abgestuft werden, indem die (möglichst kleine) Anzahl der weiteren Nexus Sets untersucht wird, die nötig sind, um all diejenigen Set Classes zu vereinen, die nicht im Haupt-Nexus-Set enthalten sind. Je dichter die Kardinalität des Nexus Sets an der Zahl 6 ist, desto aussagekräftiger und ergo besser ist die Komposition. Bei guten Kompositionen ist insbesondere die Art des musikalischen Hervortretens und die Artikulation der Set Classes von Interesse.

- *Alternative Analysen* verwenden zwar das grundlegende Instrumentarium der Set Theory und größtenteils auch die Forte'schen 224 Set Classes, messen dem Nexus Set aber keine oder nur geringere Bedeutung zu. Zur Begründung wird angeführt, dass für die Sets teilweise entgegen den Segmentierungsrichtlinien Fortes abstruse Formen gewählt werden müssen, damit diese dann auf die „richtigen" Set Classes zurückführbar sind, die mit „dem" Nexus Set in Verbindung stehen. Die konkreten alternativen Analyseziele ergeben sich daraus, welche der ungefähr 40 vorgeschlagenen Theorien gewählt wird. Inhaltsästhetische Gesichtspunkte können dabei eine Rolle spielen.

- *Kompositionstreue Analysen* setzen für Kompositionen, die nicht nach der Set Theory erstellt wurden, Sets und Set Classes nur als technisches Hilfsmittel ein, und greifen weder auf Fortes Einteilung der Set Classes noch auf dessen Ansichten zum Nexus-Set zurück. Statt dessen versuchen sie, Sets nur in dem Sinne zu interpretieren, wie es der Komponist des untersuchten Werks getan hat. Inhaltsästhetische Aspekte können – falls dies sinnvoll erscheint – mit einbezogen werden.

Ungeachtet der musiktheoretischen Unterschiede besteht ein ideales Ergebnis jeder der genannten Analysevarianten ebenso wie ein Ergebnis einer thematischen oder leitmotivischen Analyse aus einer *annotierten Partitur*, einem Katalog und einem auf diesen beiden Elementen fußenden *erklärenden Text* (siehe S. 30 und S. 37). Zusätzlich zu den bereits in Abschnitt 3.1.2 diskutierten Unterschiedlichkeitskategorien „Benennung" und „Gewichtung" (siehe S. 37) ergeben sich aufgrund der in diesem Abschnitt geschilderten musiktheoretischen Besonderheiten der Set Theory die folgenden zusätzlichen Besonderheiten zu den bereits untersuchten Analysearten:

- Markierte Sets werden nicht weiter in melodische, harmonische und rhythmische Komponenten unterteilt, da dies für die Bestimmung der Set Class nicht notwendig ist.

- Die Bezeichnung für ein markiertes Set kann nicht frei gewählt werden, sondern ergibt sich eindeutig dadurch, dass zunächst dessen Prime Form berechnet und anschließend der Name der dazugehörigen Set Class in der Set-Class-Tabelle nachgeschlagen wird.

- Das Set-Exzerpt muss weder Gesangstext noch Tonart enthalten, da es bei dem auf Berechnung zurückgreifenden Verfahren weder auf Text noch auf Tonart ankommt.

Die überwiegende Anzahl der Einträge auf einem Notizzettel (Exzerpt, Bezeichnung, Ort und Grund des Auftretens) sowie die Struktur des Notizzettels selbst bleibt von diesen Unterschieden jedoch ganz oder größtenteils unberührt.

Konkretes Vorgehen

Eine Ist-Analyse der konkreten Tätigkeiten während der Set-Theory-Analyse ergab, dass diese sich – wie auch bei den zuvor untersuchten Analysearten – primär aus einer am augenblicklichen Interesse des Analysators ausgerichteten Abfolge des Suchens, Markierens und Notierens besteht. Trotz dieser Gemeinsamkeit mit den anderen Analysearten beschränken sich die Unterschiede nicht wie bei thematischer und leitmotivischer Analyse alleine auf Fragen der Gewichtung und der Terminologie, sondern erstrecken sich auch auf die Art und Weise, in der die Tätigkeiten selbst durchgeführt werden.

- Existierende Kataloge, die die Ergebnisse der Analysen anderer Analysatoren verkörpern, werden bei der Suche nicht eingesetzt, da diese keine bei der Identifikation von Sets behilfliche Informationen enthalten können, da diese ja bereits sämtlich in der Tabelle der 224 Set Classes vorhanden sind. Anstatt der Kataloge kommt dementsprechend nur die erwähnte, stets gleiche Tabelle mit der Forte'schen Numerierung der Set Classes zum Einsatz.

- Die Suche nach einem Nexus Set bei der Set Theory führt zu einem zusätzlichen Abbruchkriterium bei der Suche: Eine Analyse ist vollendet, wenn ein den Analysator zufriedenstellendes Nexus Set gefunden ist.

Da beide Unterschiede weder in die Kategorie (1) „Unterschiede in der Benennung" noch (2) „Unterschiede in der Gewichtung" fallen (siehe S. 37), ergibt sich somit für die Set-Theory-Analyse eine dritte Kategorie von Unterschieden:

(3) **Ergänzungen.** Einige Analysearten gehen bezüglich der eingesetzten Arbeitsmittel über eine allen Analysearten gemeinsame Grundvorgehensweise hinaus oder sie erweitern die Tätigkeiten selbst um zusätzliche Spezifika.

Im Lichte der Ist-Analyse der Set-Theory-Analyse bedeutet dies für die verwendeten Arbeitsmittel, dass existierende Kataloge nicht zu den Grundarbeitsmitteln gehören,

sondern analyseartspezifische Ergänzungen der thematischen und der leitmotivischen Analyse darstellen, ebenso wie die Tabelle der 224 Set Classes eine spezifische Ergänzung der Analyse nach der Set Theory ist. Die Tatsache, dass bei der Set-Theory-Analyse ein zusätzliches Abbruchkriterium für die Suche besteht, stellt die einzige Ergänzung hinsichtlich einer Tätigkeit dar.

Da mit der Set-Theory-Analyse nun Ist-Analysen für alle drei betrachteten Analysearten vorliegen, können sowohl die Arbeitsmittel als auch die Tätigkeiten während einer der drei Analysearten jetzt auf eine Grundvorgehensweise zurückgeführt werden, die allen Analysearten gemein ist:

Analyseartübergreifende Tätigkeiten bei der Einzelwerkanalyse

Der Analysator ***sucht*** Strukturelemente in der Partitur. Ein gefundenes Strukturelement wird ***markiert*** und ausführlich ***notiert***. Der Analysevorgang wird so lange fortgeführt, bis alle erkannten Vorkommen aller Varianten aller Strukturelemente sowohl in der Partitur als auch auf Notizzetteln in Form einer eigenen Notizsammlung vermerkt sind. Darauf aufbauend wird dann der erklärende Text verfasst, der die Zusammenhänge aufzeigt.

Für die ***Suche*** hört sich der Analysator wenn möglich eine Tonaufzeichnung der zu untersuchenden Partiturstelle an und liest dabei in der Partitur mit.

Daraufhin geht er in der Partitur zum Beginn der zu untersuchenden Stelle zurück und nimmt die selbst verfertigten Notizsammlungen zur Hand. Er geht vom Anfang her beginnend durch die Partitur und sucht Strukturelemente. Erkennen kann er ein Strukturelement trotz dessen möglicherweise starken Veränderung und/oder „versteckten“ Notation entweder anhand seiner Erinnerung an die Tonwiedergabe bzw. durch erneutes Hören der betreffenden Stelle oder anhand der Gestalt des Strukturelements, die er aus vorangegangenen Analysesitzungen her kennt.

Die ***Markierung*** eines erkannten Strukturelements kann sich über mehrere Stimmen erstrecken und umfasst die Auswahl eines Bezeichners.

Mit der ***Anfertigung einer Notiz*** wird die eigene Notizsammlung ergänzt. Der Analysator nimmt dazu einen leeren Notizzettel zur Hand und füllt ihn in beliebiger Reihenfolge aus. Ein Notizzettel enthält:

- ein komplettes Exzerpt der ***markierten*** Stimmen in dem sich alle relevanten Strukturelementbestandteile wiederfinden müssen. Das Strukturelement erscheint darin in exakt der gleichen Weise wie in der Partitur. Die Markierung der einzelnen Komponenten wird übernommen. Die Bezeichnung der Stimmen und Taktart müssen ablesbar sein;
- eine Nummer, die angibt, um das wievielte Vorkommen es sich handelt;
- die Art der Variierung;
- den Ort des Auftretens, der so genau wie möglich angegeben wird. Die Taktnumerierung der Partitur wird beibehalten;
- den Grund des Auftretens.

Die große Anzahl von Gemeinsamkeiten bezüglich der im Rahmen von Ist-Analysen festgestellten Arbeitsmittel und Tätigkeiten ist eine Grundvoraussetzung dafür, dass sich Anwendungssysteme zur Unterstützung der thematischen, der leitmotivischen und der Set-Theory-Analyse als eine Anwendungsfamilie auffassen lassen. Auf ihrer Basis werde ich im Rest dieses Kapitels die weiteren Einflüsse aus den Kontexten *Handhabung & Präsentation* und *verwendete Technik* untersuchen und in Abschnitt 3.4 schließlich ein Kernsystems einer Anwendungsfamilie zur musikwissenschaftlichen Analyse skizzieren.

Wie ich aber zu Beginn der Untersuchung des Kontexts *Anwendungsbereich* dargelegt habe, sollen Anwendungssysteme jedoch nicht lediglich den aktuellen Zustand am Arbeitsplatz der Anwender wiedergeben, sondern wenn möglich auch Verbesserungswünsche berücksichtigen, um Unzulänglichkeiten der bisherigen Arbeitsweise im zukünftigen Anwendungssystem zu beheben. Diese auf in einer Soll-Analyse zusammengefassten ermittelten Anforderungen stelle ich zunächst im nächsten Abschnitt 3.1.4 vor, bevor ich die Ergebnisse des Kontexts *Anwendungsbereich* in Abschnitt 3.1.5 zusammenfasse.

3.1.4 Ermittelte Anforderungen

Wie geschildert, streben es alle vorgestellten Arten der musikwissenschaftlichen Analyse idealerweise an, die strukturellen und inhaltlichen Zusammenhänge von Musikstücken vollständig aufzudecken. Wie ich aber bereits angemerkt habe (vgl. Fußnoten 6 und 7 auf Seite 30), wird dieses Ideal in der Praxis für große Werke aufgrund der manuellen Arbeitsweise nie erreicht, weil der Umfang des untersuchten Materials zu Einschränkungen aus ökonomischen Gründen zwingt. Diese Einschränkungen können unterschiedlich schwerwiegende Konsequenzen für die Qualität haben:

1. Im günstigsten Fall ist die Einschränkung rein quantitativ. Das heißt, dass das Analyseergebnis qualitativ nicht geändert werden müsste, wenn eine vollständige Analyse vorläge (vgl. [Hur96, S. 3]).

2. Weniger günstig ist es, wenn die Ergebnisqualität sich mit der Untersuchung weiteren Materials änderte. Da dies nicht herausgefunden werden kann, ohne den beträchtlichen ökonomischen Aufwand zu betreiben, die Analyse auszudehnen, ist eine über pure Intuition hinausgehende Abgrenzung zum günstigen Fall nicht möglich. Besonders problematisch ist hier die Situation, wenn der Verdacht besteht, dass sich ein Analysator den Umstand der notwendigen Beschränkung bewusst oder unbewusst derart zunutze macht, dass er aus dem Analysematerial nur seine Ansichten bestätigende und keine widerlegenden Beispiele auswählt (vgl. [Fin97, S. 579] und [Sel98, S. 21f]).

3. Im ungünstigsten Fall ist das Material so umfangreich, dass das gewünschte Analyseziel – z.B. die Aufdeckung werkumfassender Strukturen – aufgrund der notwendigen Einschränkungen schlechterdings gar nicht zu erreichen ist (vgl. [Coo68]).

Wünschenswert ist daher eine maximale Unterstützung der Analysetätigkeit durch geeignete Anwendungssysteme, so dass ideale Einzelwerkanalysen auch für große Stücke und ideale vergleichende Analysen auch für eine aussagekräftige Anzahl von Stücken mit vertretbarem Aufwand erstellbar sind. Die Konsequenzen, die sich aus den bisherigen Einschränkungen aufgrund der manuellen Arbeitsweise ergeben, könnten damit weitgehend vermieden und bewusst eklektisch ausgewählte Argumente leichter als solche identifiziert werden.

Dieser allgemeine Wunsch schlägt sich in mehreren in einer Soll-Analyse ermittelten konkreten Wünschen nieder. Sie sind jedoch bezüglich der konkreten Form der Umset-

zung unspezifiziert und geben somit nur Ausgangspunkte für die Auswahlprozesse im Rahmen der Kontexte *Handhabung & Präsentation* und *verwendete Technik* vor, in denen sie auf ihre Machbarkeit hin geprüft werden müssen. Die folgenden Anforderungen wurden ermittelt:

1. Für die **Suche** nach Strukturelementen oder anderen musikalischen Bestandteilen soll es Rechnerunterstützung geben.

2. Es ist wünschenswert, das zeitraubende und fehleranfällige manuelle **Exzerpieren** im Rahmen der Notizerstellung weitestgehend an den Rechner zu delegieren.

3. Eine **Rechnerunterstützung** speziell **für die mathematischen Anteile der Set-Theory-Analyse** (Ermittlung der Set Class eines Sets sowie die Berechnung der Prime Forms und der K- und Kh-Relationen) ist erstrebenswert.

4. Die **Verwaltung der angefertigten Notizen** soll so durch einen Rechner unterstützt werden, dass die manuelle Suche und die – vor allem bei vergleichenden Analysen relevante – Indexerstellung nach untersuchungsrelevanten Kriterien entfallen kann.

5. Da die im Rahmen einer Analyse untersuchten Aspekte ausschließlich von den Zielen und Annahmen des Musikwissenschaftlers abhängen, soll keine erzwungene Festlegung auf bestimmte Analysearten erfolgen, sondern eine **möglichst große individuelle Anpassbarkeit der Analysearten** angestrebt werden. Dies können Veränderungen des Notizaufbaus, alternative Berechnungsmethode zur Set Theory oder Kombinationen mehrerer Anpassungen bis hin zu Analysen hinsichtlich komplett anderer Strukturelemente sein.

6. Aufgrund der Tatsache, dass viele Stücke Formmerkmale verschiedener Kompositionsmethoden aufweisen (z.B. thematisch und leitmotivisch, leitmotivisch und settheoretisch) bzw. bezüglich solcher Formmerkmalvielfalt untersuchenswert erscheinen, soll ein **Wechsel der Analyseart** wie bisher möglich sein.

7. Da innerhalb einzelner musikwissenschaftlicher Institute normalerweise eine Vielzahl von Systemplattformen zum Einsatz kommt, soll das zukünftige Anwendungssystem **möglichst plattformungebunden** sein, damit problemlos zwischen Arbeitsplatzrechnern gewechselt werden kann. Dies sollte sich auch auf das technische Format der Analyseergebnisse erstrecken, damit diese möglichst einfach mit Kollegen ausgetauscht und mit Standardsoftware in zur Veröffentlichung vorgesehene *erklärende Texte* integriert werden können.

3.1.5 Kontextspezifikation

Mit den ermittelten Anforderungen habe ich die Einflüsse aus dem Entwicklungskontext *Anwendungsbereich* nunmehr vollständig spezifiziert und damit sowohl die Grundlage für die darauf aufbauende Modellierung von individuellen Anwendungssystemen als auch einer Anwendungsfamilie von Anwendungssystemen mit einem gemeinsamen Kernsystem gelegt:

(a) Die speziellen fachlichen Anforderungen der einzelnen Analysearten gehen aus den in den Abschnitten 3.1.1 bis 3.1.3 dargelegten musiktheoretischen und -historischen Betrachtungen hervor.

(b) Die Gemeinsamkeiten und Unterschiede der Analysearten habe ich in den Abschnitten 3.1.2 bis 3.1.4 herausgearbeitet:

- *Gemeinsamkeiten* ergaben sich dabei hinsichtlich einer aus Suchen, Markieren und Notieren bestehenden einheitlichen Grundvorgehensweise unter Verwendung der Arbeitsmittel Partitur, Notizzettel und selbsterstelltem Katalog (siehe Abschnitt 3.1.3) sowie bezüglich des Wunsches nach maximaler und zugleich hinsichtlich des Notizzettelaufbaus und der Plattform flexibler, sitzungsweise anpassbarer Rechnerunterstützung (siehe Abschnitt 3.1.4).

- *Unterschiede* bestehen bei allen Analysearten hinsichtlich der Benennung der Arbeitsmittel und der Gewichtung der Einträge auf den Notizzetteln. Bei der thematischen und der leitmotivischen Analyse kommen außerdem existierende Kataloge als weitere Arbeitsmittel hinzu und die Markierungen werden in melodische, harmonische und rhythmische Bestandteile untergliedert, wobei Varianten eine Rolle spielen (siehe Abschnitt 3.1.1 und 3.1.2). Lediglich bei der set-theoretischen Analyse kommen Set-Class-Tabellen zum Einsatz und es ergeben sich weitere Unterstützungsmöglichkeiten in Form von mathematischen Berechnungen von set-theoretischen Abstraktionen (siehe Abschnitt 3.1.3 und 3.1.4).

(c) Für den Kontext *Anwendungsbereich* bestehen damit ausreichende Gemeinsamkeiten, um darauf aufbauend modellierte Analysesysteme als eine Anwendungsfamilie aufzufassen. Die Frage, ob die im *Anwendungsbereich* beobachteten Unterschiede auch von denen der Kontexte *Handhabung & Präsentation* und *verwendete Technik* unabhängig sind und sich somit die für eine Anwendungsfamilie notwendigen, voneinander unabhängigen Anpassungsstellen modellieren lassen, ist allerdings erst im Rahmen der nachfolgenden Untersuchung dieser beiden Kontexte beantwortbar.

3.2 Handhabung und Präsentation

Aus der Analyse im Rahmen des Kontexts *Anwendungsbereich* geht die für den Entwicklungsprozess entscheidende Antwort auf die Frage hervor, *was* im zukünftigen Anwendungssystem modelliert werden soll. Die Frage, *wie* die ermittelten Tätigkeiten und Arbeitsmittel modelliert werden sollen, ist Gegenstand der Untersuchungen im Kontext *Handhabung & Präsentation*.

Dabei eröffnet sich ein gewaltiger Gestaltungsspielraum, innerhalb dessen für jede Tätigkeit und jedes Arbeitsmittel entschieden werden muss, wie es in einen bestimmten Teil eines Anwendungssystems umgesetzt und gegenüber dessen anderen Teilen abgegrenzt werden kann. Für die musikwissenschaftliche Analyse manifestiert sich dieses Ausmaß an Gestaltungsmöglichkeiten in bestehenden MDV-Systemen, die von offenen Sammlungen auf Kommandozeilenebene arbeitender Kleinstprogramme (z.B. Solomons

Music Analysis System[Sol84], *Humdrum* und *CMAP*) bis zu ablaufsteuernden Großautomaten reichen, bei denen die Anwender vollständig auf die Rolle von Dateneingebern und -ablesern zurückgedrängt werden (z.B. Bo Alphonces *Invariance Matrix*[Alp74] und *RUBATO*[MZ94]). Jedem dieser MDV-Systeme liegt dabei entweder eine explizit formulierte oder lediglich implizit vorhandene Sichtweise der jeweiligen Entwickler darüber zugrunde, wie der Anwendungsgegenstand zu sehen und Software zu gestalten ist. Oft ist diese Sichtweise maßgeblich von dem bevorzugten Realisierungsmitteln abhängig, so dass die Arbeitsmittel z.B. in Daten und Funktionen oder aber Objekte zerteilt werden und die Handhabung und Präsentation derjenigen einer Datenbank oder einer Tabellenkalkulation ähnelt.

Um die implizit vorhandenen Sichtweisen im Softwareentwicklungsprozess explizit zu machen und die hinderliche Verquickung von Sichtweise und technischer Realisierung aufzuheben, werden seit einiger Zeit die ursprünglich aus der Psychologie stammenden Leitbilder auch in einer softwaretechnischen Interpretation diskutiert (vgl. [MO92] und [Maa94]).

Begriff 3.1 Leitbild *(leitmotif*[11]*)*

Ein Leitbild in der Softwareentwicklung gibt im Entwicklungsprozess und für den Einsatz einen gemeinsamen Orientierungsrahmen für die beteiligten Gruppen. Es unterstützt den Entwurf, die Verwendung und die Bewertung von Software und basiert auf Wertvorstellungen und Zielsetzungen. Ein Leitbild kann konstruktiv und analytisch verwendet werden.[WAM98, S. 73]

Anstatt daher unmittelbar zur Umsetzung der im *Anwendungsbereich* identifizierten Tätigkeiten und Arbeitsmittel in Teile eines Anwendungssystems überzugehen, empfiehlt es sich zunächst, ein zur musikwissenschaftlichen Analyse passendes Leitbild auszuwählen und dieses dann als einheitliche Richtschnur für die konkrete Gestaltung des Analysesystems zu verwenden. Dazu untersuche ich die drei am häufigsten in der Literatur diskutierten Leitbilder *Fabrik*, *Objektwelten* und *Arbeitsplatz für eigenverantwortliche Expertentätigkeit* und stelle fest, welche Anpassungen für deren Anwendung musikwissenschaftliche Analysesysteme notwendig sind.

- Beim *Leitbild Fabrik* (vgl. [MO92] und [Maa94]) wird der Anwender als Bediener einer komplexen Maschine gesehen, die ihn fest durch den Arbeitsprozess führt und an bestimmten Stellen Eingaben fordert, um am Ende ein Ergebnis zu liefern. Das Ideal ist dabei die größtmögliche Automatisierung, so dass potentielle Fehleingaben des Bedieners vermieden werden können und die Chancen auf ein korrektes Resultat maximal sind. Der Bediener selbst muss nur punktuelles Fachwissen über die Eingaben besitzen. Es ist nicht nötig, dass er den gesamten Arbeitsprozess überblicken kann. Ein von einem Fachmann angelernter Hilfsarbeiter ist in vielen Fällen vollkommen ausreichend.

[11] Der englische Begriff *leitmotif* hat seinen Ursprung in dem in Abschnitt 3.1.2 diskutierten deutschen Begriff *Leitmotiv*. Obwohl nicht von Wagner geprägt, geht er auf eine Stelle in seinem Werk *Oper und Drama* zurück, in der er beschreibt, wie die wiederkehrenden musikalischen Motive die Hörer gleich einem Gefühlswegweiser durch die dramatische Struktur der Opern leiten sollen (vgl. [Wag83, S. 320]). Daraufhin prägte Hans von Wolzogen den Begriff *Leitmotiv*, der im deutschsprachigen Raum allerdings nur im musikalischen Sinne Verwendung findet. Die ins Englische entlehnte Form erfuhr dementgegen bald eine semantische Erweiterung und wurde von „Musik, die das Gefühl leitet“ auf „Idee, die das Denken leitet“ ausgedehnt.

Bezogen auf die Tätigkeiten im Rahmen der musikwissenschaftlichen Analyse lässt sich feststellen, dass nur sehr wenige davon so beschaffen sind, dass man im vornherein ein „korrektes“ Ergebnis nennen könnte. Für Tätigkeiten wie beispielsweise die Bestimmung der Set Class sind solche „kleinen Automaten“ zwar tatsächlich geeignet, doch die Reihenfolge, in der die Tätigkeiten der eigentlichen Analyse – Lesen, Hören, Suchen, Markieren, Notieren und Vergleichen – ausgeführt werden, lässt sich nicht starr festlegen, sondern muss in Abhängigkeit von der Natur des untersuchten Stücks individuell an die Arbeitsweise des jeweiligen Musikwissenschaftlers anpassbar sein. Im Gegensatz zu einem Hilfsarbeiter kann ein Musikwissenschaftler den gesamten Arbeitsablauf durchaus überblicken und weiß selbst am besten, welche Tätigkeit als nächstes sinnvoll wäre, um seinem selbst gesteckten Ziel näher zu kommen, dessen Qualität sich nicht einfach nur auf „korrekt“ und „nicht korrekt“ reduzieren lässt.

Das Leitbild *Fabrik* mit einem „großen Automaten“, der den Arbeitsablauf vorgibt, ist daher für die musikwissenschaftliche Analyse schlecht geeignet.

- *Das Leitbild Objektwelten* (vgl. [WWW90], [Jac92] und als Kritik [Mey97,S. 230f]) stellt eine unmittelbare Übertragung der Ideen der Objektorientierung auf konkrete Anwendungen dar. Das System ist hier nicht eine große monolithische Maschine, sondern besteht aus einer Vielzahl autonomer Objekte, die miteinander in Verbindung stehen, um Aufträge entgegenzunehmen und eventuell selbst zu bearbeiten oder aufzuteilen und an andere Objekte zu delegieren. Der Anwender steht dabei außerhalb des Systems und erteilt an geeignete Objekte Aufträge.

  Für die musikwissenschaftliche Analyse scheint dieses Leitbild zu technisch, um für Analysatoren fachlich verständlich zu sein. Zwar werden Anwender nicht wie bei der Fabrik auf Maschinenbediener reduziert, die sich einem Arbeitstakt unterordnen müssen, aber sie werden mit einem Artefakt konfrontiert, dessen Arbeitsweise ihnen vollkommen fremd ist und das zwar vielleicht das tut, was sie möchten, doch das dazu erst erforscht werden muss, um verständlich nutzbar zu sein. Ein dem Leitbild der *Objektwelten* verpflichtetes Anwendungssystem ist zum Beispiel das *Humdrum*-Toolkit sowie die von Michael Taylor dafür entwickelte graphische Benutzungsschnittstelle (vgl. [Tay96]).

  Aufgrund fehlender Anwendungsorientierung ist das Leitbild *Objektwelten* ebenfalls wenig für die Gestaltung eines Systems zur Unterstützung der musikwissenschaftlichen Analyse geeignet. Das Fachwissen der Anwender wird im System so umgestaltet, dass sie es selbst nicht erkennen und dessen Bedienung sie daher neu erlernen müssen.

- Beim *Leitbild Arbeitsplatz für eigenverantwortliche Expertentätigkeit* (vgl. [WAM98, S. 80f]) kommt weder die Ablaufsteuerung der *Fabrik* noch die Unverständlichkeit der *Objektwelten* zum Tragen. Statt dessen wird der Anwender als Fachmann auf seinem Gebiet angesehen, der selbständig in der Lage ist, die geeigneten Arbeitsmittel auszuwählen, um damit Tätigkeiten auszuführen, deren Ziel er selbst situationsabhängig festlegt, und die im Regelfall nicht zu einem Ergebnis führen, das mit den Kategorien „korrekt“ und „nicht korrekt“ zutreffend beschrieben werden kann.

Zu diesem Zweck sitzt er nicht an einer Maschine, sondern an einem von ihm individuell gestaltbaren Arbeitsplatz, an dem ihm für ihn verständliche Arbeitsmittel zur Verfügung stehen.

Da Musikwissenschaftler unzweifelhaft Experten auf ihrem Gebiet sind und ihnen eine Gestaltung ihres Arbeitsplatzes mit verständlichen Arbeitsmitteln mehr entgegenkommt, als abstrakte *Objektwelten*, empfiehlt es sich, als Ausgangsleitbild für die Gestaltung von musikwissenschaftlichen Analysesystemen den *Arbeitsplatz für eigenverantwortliche Expertentätigkeit* zu wählen.

Im Gegensatz zu der „Standardverwendung", die das Leitbild in der WAM-Praxis im Bereich der Finanzdienstleistungen erfährt (vgl. [WAM98, S. 80]), muss hier allerdings auf zwei Besonderheiten Rücksicht genommen werden:

- **Musikwissenschaftler besitzen oft nur geringe Erfahrung im Umgang mit Rechnern.** Wird ein neues Anwendungssystem in einer Bank oder Versicherung entwickelt, so blicken die Sachbearbeiter bereits auf ihre Erfahrungen mit zahlreichen Vorgängersystemen zurück und können gewisse technische Abstraktionen relativ leicht nachvollziehen, so dass Tätigkeiten, die z.B. zum Starten des Systems selbst führen, nicht aufwendig als Teil des Systems mitgestaltet werden müssen, sondern von den Anwendern in Eigenregie selbst verrichtet werden können. Bei Musikwissenschaftlern kann solche Vorerfahrung nicht vorausgesetzt werden. In der musikwissenschaftlichen Praxis werden zumeist nur – wenn überhaupt – Standardsoftware-Programme wie Textverarbeitungen und Webbrowser zur Literaturrecherche eingesetzt, so dass auf technische Erfahrung im obigen Sinne nicht aufgebaut werden kann.

- **Die Analysetätigkeit stellt nur einen Ausschnitt musikwissenschaftlicher Tätigkeiten dar.** Im Finanzdienstleisterbereich ist es heute bereits möglich, den ganz überwiegenden Teil der zu erledigenden Tätigkeiten vollständig durch geeignete Softwaresysteme zu unterstützen (vgl. [Bäu98, S. 119]). Im Bereich musikwissenschaftlichen Forschens ist dies nicht der Fall. Abgesehen von der eingangs erwähnten Verwendung von Standardsoftware kommen Rechner nur sporadisch zum Einsatz. Ein kompletter musikwissenschaftlicher Arbeitsplatz kann daher nicht geschaffen werden.

Für den Entwicklungsprozess eines musikwissenschaftlichen Analysesystems liegt es daher nahe, ein neues Leitbild *Werkzeugunterstützung für eigenverantwortliche Expertentätigkeit* zu verwenden, welches das Leitbild vom *Arbeitsplatz für eigenverantwortliche Expertentätigkeit* in den beiden genannten Punkten spezialisiert. Auf die sich daraus ergebenden Besonderheiten bei der Gestaltung eines musikwissenschaftlichen Analysesystems werde ich in den Kapiteln 4 und 5 zurückkommen, wenn es darum geht, die Komponenten und das Kernsystem einer Anwendungsfamilie so zu modellieren, dass sie auch von technisch nicht versierten Anwendern gehandhabt werden können. Dazu halte ich sie in einem Ergebnis fest:

Ergebnis 3.1

Bei der Gestaltung von Anwendungssystemen für Musikwissenschaftler kommt das Leitbild *Werkzeugunterstützung für eigenverantwortliche Expertentätigkeit* zum Tragen. Dabei müssen in einem über die WAM-Praxis hinausgehenden Maß auch Konfigurations- und andere technische Aufgaben im Vorfeld der eigentlichen Arbeit auf solche Weise in die anwendungsorientierte Gestaltung mit einbezogen werden, dass sie vollkommen fachlich verständlich sind und somit von den Anwendern sicher ausgeführt werden können. Die Ergebnisse einer rechnergestützten Analyse müssen in einer solchen Form aufbereitbar sein, dass sie auch außerhalb des Analysesystems nutzbar sind.

Obwohl die Wahl des Leitbilds *Werkzeugunterstützung für eigenverantwortliche Expertentätigkeit* den Gestaltungsrahmen bereits erheblich einschränkt, so verbleibt immer noch eine Vielzahl von Optionen, wie die im *Anwendungsbereich* ermittelten Analysetätigkeiten, Arbeitsmittel und Anforderungen konkret in Bestandteile eines Analysesystems umgesetzt werden können. Um eine klare Vorstellung von einem zukünftigen Anwendungssystem zu vermitteln, empfiehlt es sich daher, ein gewähltes Leitbild auf griffige Weise so zu konkretisieren, dass sowohl Anwender als auch Entwickler sich unmittelbar vorstellen können, wie die Tätigkeiten und Arbeitsmittel realisiert werden. Zu diesem Zweck wird in der Literatur das Konzept der *Entwurfsmetapher* vorgeschlagen:

Begriff 3.2 Entwurfsmetapher *(design metaphor)*

Eine Entwurfsmetapher ist eine bildhafte, gegenständliche Vorstellung, die ein Leitbild fachlich und konstruktiv „ausgestaltet", d.h. konkretisiert. Eine Entwurfsmetapher strukturiert die Wahrnehmung und trägt zur Begriffsbildung bei. Sie leitet die Vorstellung und Kommunikation über das, was fachlich analysiert, modelliert und technisch realisiert werden soll. Eine Entwurfsmetapher dient der Gestaltung von Softwaresystemen, indem sie Handhabung und Funktionalität für die Beteiligten verständlicher macht.[WAM98, S. 79]

Die bisher zum WAM-Leitbild vorgeschlagenen Entwurfsmetaphern sind *Werkzeug*, *Automat*, *Material*, *Arbeitsumgebung* und *Behälter*[12] (vgl. [WAM98, S. 83ff]). Da sich die drei letztgenannten Metaphern unmittelbar in der Ist-Analyse wiederfinden, ist ihre Modellierung in einem Analysesystem relativ unproblematisch:

- Die ***Arbeitsumgebung*** ist der Ort, an dem sich die Arbeitsmittel befinden und an dem die Arbeit stattfindet. Der Experte kann sich an diesem Arbeitsplatz die Arbeitsmittel so zurechtlegen, wie er es für seine Zwecke am sinnvollsten erachtet. Wenn die Arbeit mehr als eines Experten betrachtet wird, muss zwischen mehreren individuellen, von äußeren Zugriffen abgeschirmten Arbeitsplätzen im obigen Sinne

[12] Die im Kontext von WAM diskutierten „fachlichen Dienstleister" (vgl. [OS00] und [Boh00]) stellen keine eigene Metapher dar, sondern sind lediglich ein dem Kontext *verwendete Technik* zuzuordnendes softwaretechnisches Konstrukt ohne eigene Bedienschnittstelle, um Teile der Funktionalität von Softwarewerkzeuge und -automaten zu realisieren. Ich diskutiere sie im Zusammenhang mit graphisch komplexen Materialien in Abschnitt 6.1.

und einer allgemein zugänglichen Arbeitsumgebung unterschieden werden, in der z.B. seltener benötigte Arbeitsmittel gelagert sind (vgl. [WAM98, S. 86f])

Da die in Abschnitt 3.1 beschriebenen Analysetätigkeiten ausschließlich von einzelnen Musikwissenschaftlern verrichtet werden, die ihre Ergebnisse nicht während der Analyse sondern entweder davor oder danach austauschen, verwende ich in dieser Arbeit die Begriffe „Arbeitsumgebung“ und „Arbeitsplatz“ synonym.

- ***Materialien*** sind der eigentliche Gegenstand des Arbeitsprozesses und können mit Hilfe von *Werkzeugen* und *Automaten* bearbeitet und sondiert werden.

  Ausgangspunkte für die Modellierung von Softwarematerialien sind die während der Ist-Analyse identifizierten Gegenstände. Je nach fachlicher Relevanz kann das dem Material zugrundeliegende Konzept wichtiger sein als die konkrete physische Form. So treten bei einem Formular beispielsweise die Layout-Details der gewählten Schriftart, der Absatzausrichtung und der Verteilung auf mehrere Seiten hinter die Semantik und Benennung der auszufüllenden Felder zurück (vgl. [WAM98, S. 85f]).[13]

  Partituren, Notizzettel, Kataloge und Set-Class-Tabellen lassen sich als Materialien der musikwissenschaftlichen Analyse auffassen.

- ***Behälter*** sind spezielle Materialien, die sich dadurch auszeichnen, dass sie weitere Materialien aufnehmen und somit auf höherer Ebene gliedern und ordnen können. Ein mit Inhaltsverzeichnis versehener Ordner mit Akten zu einem bestimmten Vorgang kann beispielsweise anders gehandhabt werden als auf dem Schreibtisch verstreute Akten zum selben Thema: Im Ordner fallen fehlende oder entnommene Akten sofort auf – und deuten damit sofort z.B. auf einen noch nicht abgeschlossenen Vorgang hin – während diese Schlussfolgerung anhand der verstreuten Einzelakten erst erarbeitet werden müsste. Behälter können so geartet sein, dass sie wie z.B. ein Papierkorb jede Art von Material aufnehmen können, oder aber so, dass sie wie der erwähnte Ordner nur dazu geeignet sind, spezielle Materialien wie z.B. Akten aufzunehmen (vgl. [WAM98, S. 89f]).

  Da die während der Analyse erstellten Kataloge ausschließlich aus Notizzetteln bestehen, werde ich sie im Folgenden als Behälter modellieren.

Obwohl auch die beiden verbleibenden Metaphern *Werkzeug* und *Automat* auf allgemein bekannten Arbeitsmitteln aufbauen, hat es sich in der Praxis gezeigt, dass es nicht trivial ist, sie auf der Grundlage der Ist- und Soll-Analyse des *Anwendungsbereichs* zu modellieren. Z.B. ist im Fall der musikwissenschaftlichen Analyse auf den ersten Blick fraglich, welche Werkzeuge außer einem Bleistift und einem Radiergummi neben den bloßen Händen des Anwenders eigentlich zum Einsatz kommen. Aus diesem Grund stelle ich beide Metaphern zunächst in ihrer allgemeinen Form vor, bevor ich mit ihnen auf der Basis der Analyseergebnissen Softwarewerkzeuge und -automaten entwerfe.

[13] Die Frage, wie komplexe Materialien wie Karten, Faksimiles und Partituren modelliert werden können, bei denen das Layout ebenso wichtig ist wie der konzeptionelle Inhalt, diskutiere ich in Kapitel 6.

- Mit ***Werkzeugen*** können *Materialien* bearbeitet oder sondiert werden. Dabei hängt es von der jeweiligen Arbeitssituation ab, ob ein Gegenstand als Werkzeug oder Material aufgefasst wird: Beim Schreiben nimmt ein Bleistift die Rolle eines Werkzeugs ein, beim Anspitzen die eines Materials. Bei der Bearbeitung muss ebenso wie natürlich auch bei der Sondierung der Blick auf das Material frei sein, damit ersichtlich ist, welche Handlungen in Anbetracht dessen aktuellen Zustands möglich und sinnvoll sind (vgl. [WAM98, S. 168]).

  Softwarewerkzeuge orientieren sich nicht in erster Linie an den in einem Arbeitsbereich physisch vorhandenen Werkzeugen – wie beispielsweise dem allgegenwärtigen Bleistift –, sondern an den Tätigkeiten, die zur Erledigung einer Aufgabe ausgeführt werden, wie etwa dem Suchen, Markieren oder dem Sortieren (vgl. [WAM98, S. 83ff]). Ob und wie bei der konkreten Gestaltung der Softwarewerkzeuge mehrere Tätigkeiten in Kombiwerkzeugen vereint werden oder nicht, ist letztendlich eine Frage der Intuition, die allerdings durch *Konzeptionsmuster*[14] genannte Gestaltungsrichtlinien gelenkt werden sollte.

- Ebenso wie *Werkzeuge* dienen auch ***Automaten*** dazu, *Materialien* zu bearbeiten. Im Gegensatz zu Werkzeugen erledigen sie jedoch gemäß Einstellung eine bestimmte Routineaufgabe ohne weitere Eingriffe von außen. Ob der Automat zur Abarbeitung nur einen kurzen Augenblick – wie z.B. ein Taschenrechner – oder einen längeren Zeitraum benötigt – wie z.B. ein Kopierer, der einen 100 Seiten umfassenden Blätterstapel zwanzig Mal kopiert, sortiert und heftet – spielt dabei keine Rolle. Entscheidend ist, dass im Gegensatz zur Bearbeitung mit einem Werkzeug das Material nicht permanent im Auge behalten werden muss und dass die Qualität des Ergebnisses jederzeit nachgeprüft werden kann. Anders ausgedrückt steht beim Automateneinsatz das Ergebnis schon vorher fest und der Weg zu seiner Erzielung soll lediglich aufgrund des aufwendigen Weges dorthin automatisch ausgeführt werden, während beim Werkzeugeinsatz nur auf ein Ergebnis *hingearbeitet* wird, dessen Ausprägung noch nicht bis ins letzte Detail feststeht (vgl. [WAM98, S. 88f]).

Es steht außer Frage, dass sich alle Tätigkeiten im Rahmen der musikwissenschaftlichen Analyse wie beispielsweise „Exzerpieren“ problemlos in atomare Operationen wie „Note einfügen“, „Pause einfügen“, „rhythmischen Wert verändern“, etc. unterteilen ließen und somit als Softwarewerkzeuge umsetzbar wären (vgl. [WAM98, S. 195]). Angesichts des Wunsches nach maximaler Rechnerunterstützung (siehe Abschnitt 3.1.4) muss aber geprüft werden, welche Abfolgen solcher atomarer Operation als formalisierbar gelten und somit in Form von Automaten handhabbar gemacht werden sollten. Diese Untersuchung hinsichtlich des Grades der Formalisierbarkeit führe ich im Folgenden anhand des aktuellen Standes der musikwissenschaftlichen Diskussion durch (Abschnitt 3.2.1).

Da Werkzeuge stets den Blick auf die Materialien freigeben, muss anschließend noch untersucht werden, welche Besonderheiten sich hinsichtlich der graphischen Repräsentation der Materialien Partitur und Notiz ergeben. Vor dem Hintergrund der Modellie-

[14] Konzeptionsmuster existieren für alle Entwurfsmetaphern. Auf Grund ihrer größeren Nähe zur technischen Umsetzung sind sie allerdings primär dem Kontext *verwendete Technik* zuzuordnen und werden hier nicht weiter diskutiert.

rung der verschiedenen Analysesysteme als eine Anwendungsfamilie, zeige ich dabei auch, welche präsentationsbezogenen Gemeinsamkeiten und Unterschiede bestehen und dass die Unterschiede von denen im Kontext *Anwendungsbereich* unabhängig sind, so dass eine Modellierung voneinander unabhängiger Anpassungsstellen möglich ist (Abschnitt 3.2.2).

3.2.1 Besonderheiten bei der Modellierung von musikwissenschaftlichen Tätigkeiten

Während in WAMs Ursprungsdomäne des Sachbearbeiterarbeitsplatzes relativ rasch entschieden werden kann, ob eine Tätigkeit vollkommen automatisierbar ist oder nicht, so stellt die Frage, inwiefern sich die einer musikwissenschaftlichen Tätigkeit zugrundeliegenden musikalischen Bewertungen objektivieren und damit formalisieren lassen, einen zentralen Diskussionspunkt in der Musikwissenschaft dar. Um daher über eine Modellierung als Softwarewerkzeug oder als Softwareautomat entscheiden zu können, untersuche ich im Folgenden für jede der im Kontext *Anwendungsbereich* identifizierten Tätigkeiten, inwiefern für sie im Lichte des aktuellen Standes der musikwissenschaftlichen Forschung gilt, dass sie (1) formalisierbar ist oder (2) sie auf nicht formalisierbarem menschlichem Expertenwissen und musischem Empfinden beruht.

Am leichtesten lassen diejenigen Tätigkeiten einordnen, die Musikwissenschaftler in ihren Soll-Anforderungen in Abschnitt 3.1.4 mit Adjektiven wie „zeitraubend“ und „mathematisch“ versehen haben:

- Das **Exzerpieren** eines bereits vom Analysator in der Partitur markierten Strukturelements kann jeder Laie ohne jegliches Expertenwissen durchführen, da der markierte Bereich lediglich unverändert reproduziert und die zwischen den Markierungsbestandteilen nicht markierten Partiturteile entfernt werden müssen (siehe S. 30). Diese Tätigkeit kann daher auch von einem Softwareautomaten übernommen werden.

- Die **Bestimmung der Set Class eines Sets** erfolgt rein mathematisch und erfordert daher ebenfalls kein musikwissenschaftliches Expertenwissen. Die anhand des →Notenschlüssels und der →Versetzungszeichen einfach bestimmbare Tonhöhe der Mitglieder des Sets muss lediglich gemäß des in Abschnitt 3.1.3 vollständig beschriebenen Verfahrens in Pitch Classes umgewandelt und sortiert werden, um dann die Prime Form zu bestimmen, deren Set Class dann einfach in der Set-Class-Tabelle nachgeschlagen werden kann (siehe S. 38) Ein Softwareautomat kann dieses rein formale Verfahren ebensogut ausführen wie ein mathematisch begabter Laie.

  Das Gleiche gilt auch für die Berechnung der K- und Kh-Relation (siehe S. 39).

Da zahlreiche der weiteren Soll-Anforderungen sich abstrakt auf die Flexibilität des Analysesystems beziehen (Anpassbarkeit und Wechsel der Analyseart sowie Plattformunabhängigkeit; siehe S. 45) verbleiben als zu untersuchende unterstützbare Tätigkeiten lediglich das *Suchen und Markieren von Strukturelementen* sowie die *Verwaltung*

von Notizen in einem Katalog[15]. Da diese Tätigkeiten in unterschiedlich hohem Maße auf Musikexpertenwissen und musikalisches Empfinden voraussetzen, lassen sie sich nicht als reine Automaten umsetzen:

- Eine Suche in der Partitur oder der Vergleich zweier Strukturelemente im Katalog ist bei **vollkommen identischer Notation** am unkompliziertesten. Aufgrund der enorm großen Zahl von Abwandlungsmöglichkeiten jedes einzelnen der ein Strukturelement ausmachenden Klangelemente (siehe S. 26) kann davon ausgegangen werden, dass vollkommene Identität nicht auf einem Zufall, sondern eindeutig auf der Intention des Komponisten beruht. Solche Suchen und Vergleiche, die auf einem rein graphischen Vergleich beruhen und keinerlei Verständnis der notierten Musik voraussetzen, können durch einen musikwissenschaftlichen Laien oder auch einen Softwareautomaten ausgeführt werden.

- Eine **Ähnlichkeit gemäß isomorpher musikalischer Umformungen** liegt dann vor, wenn sich die beiden zu vergleichenden Strukturelemente *ausschließlich* unter Verwendung der vier Elementarumformungen der Transponierung, der Umkehrung, des Krebses, sowie der gleichmäßigen Vergrößerung oder der gleichmäßigen Verkleinerung rein graphisch ineinander überführen lassen (siehe Seite 26). Im Falle der Set Theory gelten zusätzlich all jene Umformungen als isomorph, die nichts an der Zuordnung des Sets zu einer bestimmten Set Class ändern. Obwohl die Anzahl der zu prüfenden Fälle größer ist als bei der vollkommenen Identität, ändert sich an der Delegierbarkeit an einen Laien nichts, da es weiterhin nicht notwendig ist, dass der musikalische Inhalt verstanden wird. Da sich die Elementarumformungen nur auf eine begrenzte Weise kombinieren lassen, können gefundene Übereinstimmungen immer noch mit Sicherheit auf Intention des Komponisten zurückgeführt werden.

  Durch Umkehrungen und Transponierungen können sich allerdings die Versetzungszeichen ändern, so dass eine sicherer, rein notationsbezogener Vergleich nicht in allen Fällen möglich ist. Im Gegensatz zum Vergleich bei vollkommener Identität muss der Analysator die Ergebnisse deshalb zunächst überprüfen, bevor er sie verwenden kann. Daher kommt kein reiner Softwareautomat zur Umsetzung in Frage, sondern lediglich ein in ein Werkzeug eingebetteter Automat, der die Ergebnisse zunächst als Vorschläge präsentiert, über die der Analysator dann mit den Mitteln des Werkzeugs befinden kann.

- Beide der bisher diskutierten Formen der Ähnlichkeiten machen nur einen Bruchteil der in den Abschnitten 3.1.1 und 3.1.2 beschriebenen Variationsmöglichkeiten eines Strukturelements aus. Solche **komplexen Ähnlichkeiten** konnten bisher weder von Musik- noch Kognitionswissenschaftlern auf Formalismen reduziert werden, sondern müssen von einem menschlichen Analysator entweder aus der Tonwiedergabe eines Stücks herausgehört oder aus einer Partitur bzw. einem Exzerpt herausgelesen

[15] Die Verwaltung von Notizen nach alphanumerischen Gesichtspunkten lässt sich mit aus dem Bereich der Datenbanken bekannten Techniken problemlos automatisieren (vgl. [Dit99]). Untersuchenswert ist hingegen die Identifikation einer Menge von Katalogeinträgen, die zusätzlich oder ausschließlich bestimmten musikwissenschaftlichen Kriterien genügt. Da diese Identifikation rein auf einem Vergleich der Exzerpte beruht, diskutiere ich die Katalogverwaltung im Weiteren nicht separat, sondern lediglich als ein Spezialfall desjenigen Vergleichs, der auch der Suche in der Partitur zugrundeliegt.

werden. Eine Automatisierung dieser Vergleiche gilt zum jetzigen Zeitpunkt aus dreierlei Gründen als unmöglich (vgl. [Sel98, S. 14]):

1. *Die Komplexität der Kompositionsregeln.* Wie ich in Abschnitt 3.1.1 dargelegt habe, haben sich die Entwicklungsformen mit ihrem komplexen Regelwerk im Laufe der Kompositionsgeschichte aus einfachen Reihungsformen herausgebildet. Obwohl zu keinem Zeitpunkt angenommen wurde, die Anwendung dieser Regeln alleine reiche aus, um Meisterwerke zu komponieren, so wurde doch früher angenommen, dass bei Befolgung dieser Regeln zumindest ein annehmbares Musikstück entsteht. Jüngere Forschungen, bei denen diese Regeln in Kompositionsprogramme eingebaut wurden, zeigen aber, dass die Ergebnisse teilweise vollkommen unbefriedigend sind (vgl. [Ebc86], [Sch89] und [Ebc92])[16]. Da die Kompositionsregeln sich nicht formalisieren lassen, besteht umgekehrt aufgrund des anfangs erwähnten Zusammenhangs zwischen Musiktheorie und Musikanalyse auch keine Möglichkeit, die Regeln zum Zweck der automatischen Suche einzusetzen.

2. *Die Größe der Werke.* Unabhängig von der Komplexität der Kompositionsregeln unterbindet die kombinatorische Explosion der möglichen Zusammensetzungen das systematische Ausprobieren selbst einfacher Identifikationsregeln. Selbst für die mathematiknahe Set Theory, bei der sich sämtliche Variationsmöglichkeiten explizit aufzählen lassen, führen solche Versuche nicht zum Erfolg (vgl. [Cas94, S. 17]).

3. *Die Komplexität der Notenschrift.* Wie ich am Beispiel der „versteckten" Notation aufgezeigt habe, können klanglich äquivalente Strukturelemente in einer großen Vielfalt von Möglichkeiten in der Partitur niedergeschrieben werden (siehe S. 27). Da musikalische Notation weitaus komplexer ist, als die Notation mathematischer Formeln (vgl. [Byr94]), und es darüber hinaus noch zahlreiche Zeichen wie →Verzierungen oder andere →Abbreviaturen gibt, die der menschlichen Interpretation bedürfen, sind automatische Suchen und Vergleiche nicht machbar.

Für nach inhaltsästhetischen Gesichtspunkten komponierte Vokalwerke kommt noch hinzu:

4. *Die Komplexität der menschlichen Sprache.* Die Interpretation der Gesangstexte und Regieanweisungen zwecks Ausdeutung des musikalischen Aufbaus erfordert ein enormes kulturelles Hintergrundwissen und birgt zu zahlreiche Vieldeutigkeiten, um sie automatisch ausdeuten zu können (vgl. [DG99, S. 976f]).

Diese vier Gründe verhindern zwar nicht jeglichen Einsatz von Softwareautomaten, doch wie schon im Fall der Ähnlichkeit gemäß isomorpher musikalischer Umformungen können automatisch gefundene Übereinstimmungen im Rahmen eines

[16] Bill Schottstaedt gießt in seinem vielbeachteten Artikel „Automatic Counterpoint"[Sch89] die explizit vorgegebenen kontrapunktischen Kompositionsregeln in automatische Kompositionsalgorithmen. Kemal Ebcioglu untersucht in [Ebc86] und [Ebc92] das enge Feld der Harmonisierung von Bach-Chorälen und findet alleine dort über 300 zuvor unbekannte Regeln.

Werkzeugs nur als Vorschläge präsentiert werden, die der Analysator genau prüfen muss, bevor er sie anerkennt. Während bei den zuvor diskutierten Vergleichsmethoden zumindest sicher war, dass alle Strukturelemente gefunden werden können und lediglich einige Fehltreffer aussortiert werden mussten, so muss im Fall komplexer Ähnlichkeit stets davon ausgegangen werden, dass nur ein Bruchteil der tatsächlich vorhandenen Strukturelemente automatisch auffindbar ist.

Da Musikstücke größtenteils Strukturelemente enthalten, die auf komplexe Weise zueinander ähnlich sind, müssen die Tätigkeiten der Suche und der Markierung in der Partitur sowie des Vergleichs von Katalogeinträgen daher als Softwarewerkzeuge modelliert werden. Automatische Suchhilfen, die Treffervorschläge anbieten, dürfen lediglich optionale, in Werkzeuge eingebettete Angebote darstellen, aber nicht die einzige Möglichkeit sein, um Strukturelemente zu finden, zu markieren und zu notieren. Nur für das Exzerpieren und set-theoretische Berechnungen sind Softwareautomaten uneingeschränkt geeignet.

Falls eingebetteten Suchhilfen angeboten werden, dann empfiehlt es sich gemäß neueren musikwissenschaftlichen Erkenntnissen, die Suchmöglichkeiten so flexibel wie möglich zu gestalten, um die Chancen auf musikalisch relevante Treffer zu erhöhen:

1. Da Strukturelementen äußerst selten in identischer Form wiederkehren, sollten die Suchalgorithmen nicht wie bei klassischer Zeichenkettensuche ausschließlich exakte Treffer liefern (vgl. [BM77]), sondern ein zuvor festlegbares Maß an Abweichungen zulassen (vgl. [CIR98], [HO92] und [DH80]).

2. Da Themen und Leitmotive melodische, harmonische und rhythmische Bestandteile aufweisen können, sollten diese Bestandteile für Suchanfragen voneinander isolierbar und in ggf. abstrahierter Form wieder kombinierbar sein, um die veränderliche Gestalt des Strukturelements besser fassen zu können (vgl. [Sel98, S. 54]). Anstatt beispielsweise das Thema „B-A-C-H“ mit den Notendauern „Viertel-Achtel-Achtel-Halb“ nur in exakt dieser Kombination der Bestandteile zu spezifizieren, kann es hilfreich sein, lediglich nach „Fallend-Steigend-Fallend“ im Rhythmus „Kurz-Kürzer-Gleich-Länger“ zu suchen.

Die Handhabung der Materialien durch Werkzeuge zieht die Konsequenz nach sich, dass die Materialien graphisch dargestellt werden müssen, damit die Analysatoren sie sondieren und modifizieren können (siehe S. 52). Die sich dabei ergebenden Besonderheiten und partiturartbedingten Unterschiede diskutiere ich im nachfolgenden Abschnitt 3.2.2, bevor ich die Einflüsse aus dem Kontext *Handhabung & Präsentation* in Abschnitt 3.2.3 in einer Kontextspezifikation zusammenfasse.

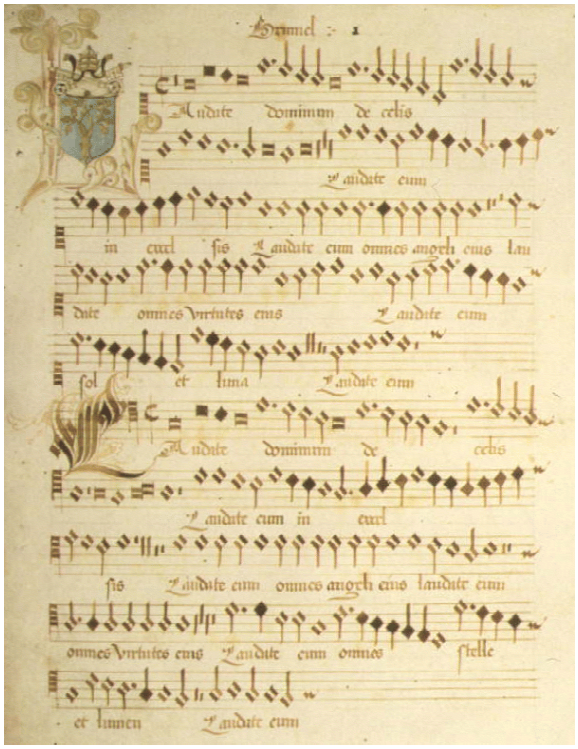
3.2.2 Besonderheiten bei der Modellierung von musikwissenschaftlichen Arbeitsmitteln

In den Anwendungsbereichen, in denen WAM hauptsächlich eingesetzt wird, spielen Materialien mit aufwendiger graphischer Darstellung keine Rolle. In den meisten Fällen reichen textorientierte Formulare und Tabellen aus, um die Arbeitsmittel zu

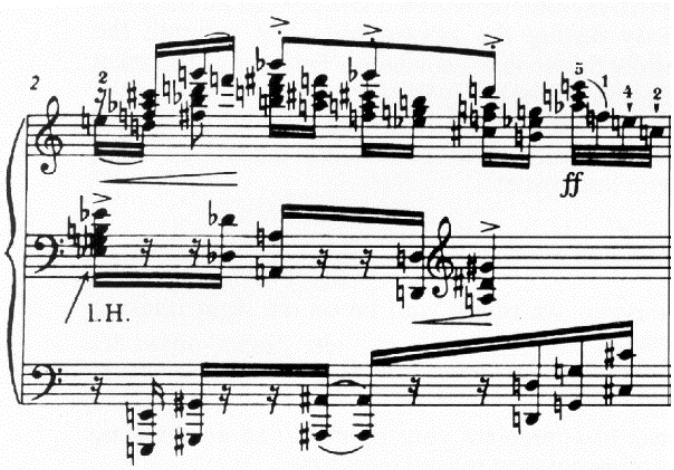
modellieren (vgl. [WAM98, S. 169]). Einige im Rahmen von WAM-Forschungsprojekten erstellte Systeme besitzen zwar Materialien mit graphischen Bestandteilen, doch die dort benötigten Diagramme sind für jeden Anwendungsfall von der gleichen Art und wechseln nicht in Abhängigkeit vom jeweiligen Arbeitsgegenstand (vgl. [Wul95] und [Nie98]).

Bei der musikwissenschaftlichen Analyse kommen dementgegen graphisch hochkomplexe Partituren zum Einsatz, deren Art der Darstellung in Abhängigkeit vom jeweils analysierten Werk stark voneinander abweichen kann. Von den hunderten existierenden und vorgeschlagenen Notationsarten (vgl. [Ree97]) sind die folgenden drei für die Praxis der im Kontext *Anwendungsbereich* beschriebenen Analysearten am relevantesten:

1. **Mensuralnotenschrift** wurde ab der Mitte des 13. Jahrhunderts verwendet und fixiert in einem Fünfliniensystem lediglich Tonhöhe und Tondauer. Trotz dieser im Gegensatz zur „modernen“ Notenschrift (s.u.) geringen Ausdruckskraft stellt sie einen wesentlichen Fortschritt gegenüber früheren Notenschriften wie den Neumen dar, mit denen selbst Tondauern entweder gar nicht oder nur ungenau wiedergegeben werden konnten. In Mensuralnotation tauchen noch neumenartige Ligaturen mehrerer →Töne auf und Tondauern sind auf solche Werte beschränkt, die sich durch exakte Verdopplung oder Halbierung einer Grundnotendauer gewinnen lassen.

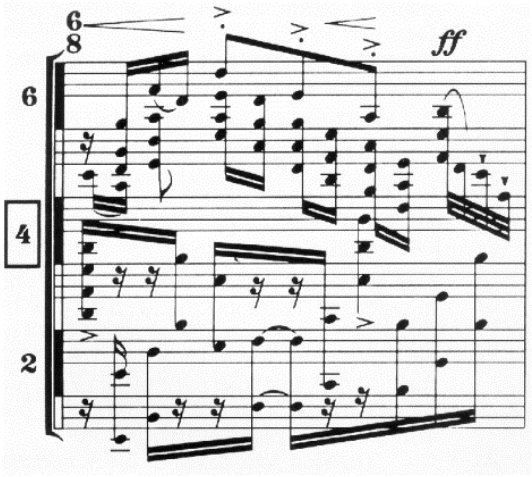


2. **Moderne Notenschrift** besteht seit dem 17. Jahrhundert und stellt eine Weiterentwicklung der Mensuralnotation dar, die sie aufgrund ihrer wesentlich höheren Aussagekraft fast vollkommen verdrängt. Die Interpretation von Noten kann durch →Taktangaben, Notenschlüssel, Versetzungs- und →Artikulationszeichen beeinflusst werden. Töne können auf Hilfslinien auch außerhalb des Systems notiert werden. Durch →Verlängerungspunkte, →Bindebögen und →anomale Teilungen sind alle denkbaren Ton- und auch Pausendauern notierbar. Da seit dem 17. Jahrhundert weitere Notenschriften entwickelt wurden, ist die Bezeichnung „modern“ nicht mehr ein-



deutig. Daher wird zur Bezeichnung dieser speziellen Notenschrift immer häufiger das aus dem Englischen abgeleitete Akronym „CMN“ (*common musical notation*) verwendet (vgl. [Sel97a, S. 3]).

3. **Chromatische Notenschrift nach Ailler-Brennink** wird seit dem 20. Jahrhundert vor allem der Übersichtlichkeit halber bei atonalen Kompositionen eingesetzt.[17] Notiert man atonale Werke in CMN, so benötigt in der Regel jede Note ein Versetzungszeichen, was in der für tonale Musik ersonnenen CMN nur als Ausnahme vorkommt. Die chromatische AB-Notation tritt den dadurch entstehenden Lesbarkeitsproblemen durch ein Niederschrift auf 4 System- und 2 Hilfslinien entgegen, bei der jede Tonstufe immer einem Halbton entspricht, und somit sowohl Versetzungszeichen als auch Notenschlüssel zur Zuordnung von Tönen zu Notenlinien überflüssig macht, da die 12 chromatischen Töne immer den gleichen 6 Linien oder Zwischenräumen zugeordnet sind.



   Die chromatische AB-Notation ist die verbreitetste,[BK97] aber nicht die einzige zur Niederschrift atonaler Werke vorgeschlagene Notenschrift.[Ree97]

Die Frage, auf welche Weise eine Partitur dargestellt werden muss, hängt nicht von der Art der Partitur, sondern ausschließlich von der Notationsart ab, die der Herausgeber des Werks zu dessen Satz verwendet hat:

- Da chromatische Notation auch zusehends für nicht-atonale Werke populär wird, kann ein nach thematischen Gesichtspunkten komponiertes Werk sowohl in Mensuralnotation, CMN oder auch AB-Notenschrift vorliegen.

- Für leitmotivische Vokalwerke ist CMN genauso geeignet wie AB-Notation.

- Mensuralnotation wird von Notensetzern aufgrund ihrer beschränkten Ausdruckskraft lediglich für frühe thematisch orientierte Kompositionen verwendet.

Da die einzigen Unterschiede im Kontext *Handhabung & Präsentation* sich auf die Art der Partiturdarstellung beziehen und diese unabhängig von den analyseartbedingten Unterschiede im Kontext *Anwendungsbereich* sind, sind damit die Voraussetzungen zur Modellierung von zwei voneinander unabhängigen Anpassungsstellen des Kernsystems einer Anwendungsfamilie gegeben.

[17] Das Notationsbeispiel zeigt ebenso wie bei dem CMN-Beispiel den zweiten Takt von Arnold Schönbergs *Klavierstück Opus 11, Nr. 3*.

3.2.3 Kontextspezifikation

Aufbauend auf der Spezifikation der Einflüsse aus dem *Anwendungsbereich* habe ich in diesem Abschnitt untersucht, (a) welche Formen der *Handhabung & Präsentation* für Anwendungssysteme zur Unterstützung der musikwissenschaftlichen Analyse geeignet sind und dabei im Hinblick auf eine Modellierung dieser Analysesysteme als eine Anwendungsfamilie herausgearbeitet, (b) welche Gemeinsamkeiten und Unterschiede sich feststellen lassen und (c) in welchem Verhältnis diese zu den Gemeinsamkeiten und Unterschieden des *Anwendungsbereichs* stehen:

(a) Ich habe gezeigt, dass das eine Spezialisierung des WAM-Leitbilds des *Arbeitsplatzes für eigenverantwortliche Expertentätigkeit* für die musikwissenschaftliche Analyse geeignet ist und in Abschnitt 3.2.1 dargelegt, wie die Analysearbeitsmittel und -tätigkeiten mit Hilfe der WAM-Entwurfsmetaphern umgesetzt werden können.

(b) Während die Prinzipien der Handhabung und der Präsentation allen Analysearten *gemein* sind, ergeben sich *Unterschiede* bezüglich der vom Herausgeber einer Partitur gewählten Notationsart: Mensuralnotation, CMN oder chromatische AB-Notation (siehe Abschnitt 3.2.2).

(c) Da die Unterschiede in den beiden Kontexten *Anwendungsbereich* und *Handhabung & Präsentation* auf voneinander unabhängige Einflüsse zurückgehen (Analyseart vs. Partiturart), sind bezüglich dieser beiden Kontexte die Voraussetzungen zur Modellierung von voneinander unabhängigen Anpassungsstellen des Kernsystems einer Anwendungsfamilie gegeben.

In Abschnitt 3.3 betrachte ich abschließend den Kontext *verwendete Technik*, um zu zeigen, dass sich die bisher spezifizierten Anwendungssysteme auch tatsächlich mit vertretbarem ökonomischem Aufwand umsetzen lassen. Da ich demonstrieren kann, dass sich auch aus technischer Sicht deutlich mehr Gemeinsamkeiten als Unterschiede ergeben und diese Unterschiede von denen des *Anwendungsbereichs* und der *Handhabung & Präsentation* unabhängig sind, kann ich schließlich in Abschnitt 3.4 ein Kernsystem einer Anwendungsfamilie zur Unterstützung der musikwissenschaftlichen Analyse skizzieren und in Abschnitt 3.5 die Ergebnisse dieses Kapitels zusammenfassen.

3.3 Verwendete Technik

Im anwendungsorientierten Softwareentwicklungsprozess hat der Kontext *verwendete Technik* den geringsten Einfluss auf das Modell des zu entwickelnden Anwendungssystems, da die für die Anwender primär relevante fachliche und benutzungsbezogene Gestaltung sich ausschließlich aus den Kontexten *Anwendungsbereich* und *Handhabung & Präsentation* ergibt. Trotzdem ist seine Untersuchung unerlässlich, da sich hier herausstellen muss, ob das soweit spezifizierte Anwendungssystem überhaupt technisch machbar und ökonomisch realisierbar ist (vgl. [FZ99, S. 775]). Dies gewinnt im Fall der musikwissenschaftlichen Analyse besonders deshalb große Bedeutung, weil

keine vergleichbaren Analysesysteme existieren, die die bisher spezifizierten Anforderungen erfüllen. Da es außer den von mir bereits in den Abschnitten 3.1.1 und 3.2.1 abgesteckten Grenzen des Rechnereinsatzes keine weiteren Tätigkeiten im Rahmen der musikwissenschaftlichen Analyse gibt, für die eine Unterstützung technisch nicht machbar wäre, kann ich mich dabei ausschließlich auf Aspekte der Ressourcenökonomie beschränken.

Der erste Schritt besteht dabei stets in der Auswahl einer bestimmten Modellierungstechnik samt Programmiersprache und dazu passender Bibliotheken und Rahmenwerke (vgl. [FZ99, S. 785f]). Aus dreierlei Gründen empfiehlt es sich dabei aufgrund der konkreten Anforderungen aus den Kontexten *Anwendungsbereich* und *Handhabung & Präsentation*, auf einen objektorientierten Entwurf und eine Implementation basierend auf dem *JWAM*-Rahmenwerk zurückzugreifen:

1. Mit den Mitteln des objektorientierten Entwurfs ist es besser als mit sogenannten strukturierten Ansätzen möglich, Gegenstände und Begriffe des fachlichen Modells ohne Strukturbrüche in ein softwaretechnisches Modell zu übertragen (vgl. [Bäu98, S. 35] und [FZ99, S. 773f]) und somit Verständnisprobleme und die daraus entstehenden zusätzlichen Entwicklungskosten zu vermeiden.

2. Da die Analyse des *Anwendungsbereichs* und die Gestaltung der *Handhabung & Präsentation* sich an WAM-Prinzipien orientieren (siehe S. 24 und S. 49), ist es vorteilhaft, zur Implementation auf ein Rahmenwerk zurückzugreifen, das es ermöglicht, diese Prinzipien bruchlos konstruktiv umzusetzen (vgl. [FZ99, S. 773] und [Bäu98, S. 58]). Dies leistet das am Arbeitsbereich Softwaretechnik der Universität Hamburg entwickelte und von der Firma Apcon WPS professionalisierte *JWAM*-Rahmenwerk (vgl. [WPS01], [LRWZ01] und [GLL+99]). Seine Struktur orientiert sich an den drei Kontexten des Softwareentwicklungsprozesses, wobei für *verwendete Technik* noch die Unterscheidung in die konkreten Schnittstellen der *Systembasis* (z.B. Schnittstelle einer bestimmten relationalen Datenbank) und deren abstrakter *Technologie* durchgeführt wird (beispielsweise das relationale Datenmodell). Dadurch, dass *JWAM* die Infrastruktur für nach dem Leitbild des Arbeitsplatzes für eigenverantwortliche Expertentätigkeit entwickelte Anwendungssysteme zur Verfügung stellt, können solche Systeme in erheblich kürzer Zeit und damit kostengünstiger erstellt werden als ohne ein entsprechendes Rahmenwerk.

3. *JWAM* ist in Java implementiert und erfüllt damit die in Abschnitt 3.1.4 gestellte Soll-Anforderung nach maximaler Plattformunabhängigkeit. Während auch für andere Programmiersprachen plattformübergreifende Quellcodestandards wie z.B. ANSI C bestehen, werden von den weitverbreiteten objektorientierten Programmiersprachen nur Java-Programme nicht direkt in eine plattformabhängige Maschinensprache übersetzt, sondern in eine plattformunabhängige – „bytecode“ genannte – Maschinensprache einer virtuellen Java-Maschine.[LY99, S. 1] Die „Java Virtual Machine“ kann als eine tatsächliche Maschine realisiert werden, ist aber meist selbst ein Programm, dass innerhalb eines einbettenden Betriebssystems läuft. Durch diese zusätzliche Indirektion zwischen Übersetzer und Maschine können Java-Programme auf jedem Rechner, auf dem eine Virtual Machine installiert ist, ohne weiteres Zutun der Anwender sofort ausgeführt werden. Da sich Musikwissenschaftler fast aus-

schließlich PCs und Workstations bedienen, und Java Virtual Machines für annähernd alle PC- und Workstation-Betriebssysteme vorliegen,[18] sind Java-Programme damit auf über 95% aller für Analysesysteme relevanten Arbeitsplatzrechner sofort ausführbar, ohne dass neue Hardware oder Betriebssystem-Software angeschafft werden muss.[19]

Zusätzlich zu diesen allgemein softwaretechnischen Festlegungen auf Modellierungstechnik, Programmiersprache und Rahmenwerke müssen noch die Auswirkungen zweier weiterer Einflüsse untersucht werden, die sich aus der sozialen Einbettung des hier beschriebenen Entwicklungsprozesses ergeben:

1. Ebenso wie für die grundlegende Softwarearchitektur und -infrastruktur ist es auch für die Realisierung der musikwissenschaftlichen Funktionalität wünschenswert, weitestgehend auf bestehende MDV-Systeme zurückgreifen zu können. Da die in musikwissenschaftlichen Instituten vorhandene Infrastruktur äußerst heterogen ist (siehe Abschnitt 3.1.4), ergeben sich daraus hohe Anforderungen an die technische Flexibilität der Anwendungsfamilie. In Abschnitt 3.3.1 betrachte ich zunächst, inwiefern existierende MDV-Systeme geeignet sind, um die im Kontext *Anwendungsbereich* spezifizierte Funktionalität als Subsystem zur Verfügung zu stellen und welche technischen Realisierungsvarianten unterstützt werden müssen.

2. Ein Analysesystem ist nur dann praxistauglich, wenn genügend Partituren in analysefähiger, binärer Form vorliegen und die Analyseergebnisse möglichst einfach mit Standardsoftware weiterverarbeitet werden können (siehe Abschnitt 3.1.4). In Abschnitt 3.3.2 lege ich dar, welches Musikdatenformat unter diesen Umständen am besten geeignet ist.

3.3.1 Musikdatenverarbeitungssubsysteme

Systeme zur Verarbeitung von Musikdaten lassen sich grob danach einteilen, ob sie sich auf (1) die Aufzeichnung und Wiedergabe von Akustikdaten, (2) deren Synthese, (3) die Erzeugung, Wiedergabe und das Einlesen von Notensatzdaten oder (4) musikwissenschaftliche Analyse beziehen (vgl. [Kor96a, S. 67ff] und [Sel97a]). Im Rahmen der geforderten Analysesysteme können Musikdatenverarbeitungsysteme zum Einsatz kommen, die Daten aller dieser vier Arten verarbeiten:

(1) *Akustikdaten*. Während der Suche soll die entsprechende Stelle des Werks direkt von einem Tonträger bzw. einer digitalen Tonaufzeichnung wiedergegeben werden (siehe Abschnitt 3.1.1).

(2) *Synthetisierte Daten*. Um die notierten Themen, Leitmotive oder Sets besser hören zu können, sollen sie ohne den „Hintergrund“ der anderen →Orchesterstimmen einzeln wiedergegeben werden. Dazu müssen Akustikdaten aus den Exzerptdaten synthetisiert werden (siehe Abschnitt 3.1.1).

[18] Für alle MS-Windows-, Apple- und fast alle UNIX-Betriebssysteme sind Implementationen der Java Virtual Machine vorhanden (vgl. [Sun01]).
[19] MS-Windows-, MacOS- und Linux-Systeme machten Anfang 2000 95% aller Betriebssysteminstallationen auf PCs und Workstations aus (vgl. [OSU00]).

(3) *Notensatzdaten*. Um die Partitur analysieren und Exzerpte betrachten zu können, müssen diese wie in der gedruckten Partitur dargestellt werden (siehe Abschnitt 3.2.2).

(4) *Analysedaten*. Um die Partituren und Exzerpte nach bestimmten musiktheoretischen Kriterien durchsuch- und vergleichbar zu machen, müssen sie in einem analysefähigen Datenformat vorliegen. Nur dann können gefundene Strukturelemente nicht nur bezüglich ihres graphischen Erscheinungsbildes, sondern auch anhand ihrer musiktheoretischen Eigenschaften untersucht und exzerpiert werden (siehe Abschnitt 3.1.4).

Von diesen vier Arten von MDV-Systemen werden spezielle Akustikdatensysteme für das geforderte Analysesystem nicht benötigt, da Akustikdaten lediglich wiedergegeben werden müssen und dies bereits mit Hilfe von Standardoperationen der Java Virtual Machine möglich ist.

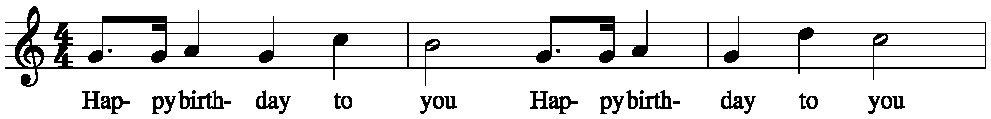
Um nur ein Minimum an externen Subsystemen integrieren zu müssen, sollte von den verbleibenden Musikdatenverarbeitungssystemen dasjenige ausgewählt werden, das den größtmöglichen Anteil der geforderten Funktionalität abdeckt. Da kein für die Integration in Java geeignetes Notensatzsystem vorliegt, erfüllt das 1994 von David Huron entwickelte *Humdrum Toolkit*[Hur94]dieses Kriterium am besten.[20] Es besteht aus einer Sammlung von über 40 Datenformaten und über 70 Werkzeugen, mit denen die Daten sowohl analytisch als auch zu Synthesezwecken bearbeitet und konvertiert werden können. Aus der Vielzahl der Eigenschaften *Humdrums* stelle ich im Folgenden nur diejenigen vor, die sich unmittelbar aus den Anforderungen ergeben:

- Zur Unterstützung der **Verwaltung von Musikdaten** setzt *Humdrum* auf vorhandenen Systemprogramme wie **`sort`** und **`grep`** und legt zu diesem Zweck die Musikdaten in reinen ASCII-Dateien ab, die nach *Humdrum*-spezifischen Kriterien aufgebaut sind:

  - Globale Kommentare enthalten textuelle Informationen über das gesamte Werk und befinden sich konventionsgemäß am Anfang einer Datei. Sie sind durch „`!!`“ am Zeilenanfang gekennzeichnet und können frei formuliert Texte enthalten. Für eine erleichterte Verwaltung werden aber bestimmte dreibuchstabige Bezeichner nach „`!!!`“ empfohlen, die die Bedeutung des jeweiligen Kommentars festlegen – z.B. „`!!!COM: Ludwig van Beethoven`“, um den Komponisten (engl. *composer*) zu kennzeichnen.

  - Die eigentliche musikalische Information wird tabellarisch gespeichert, wobei Zeilen Gleichzeitigkeit und *spines* genannte Spalten Zugehörigkeit zu einer logischen Sequenz – z.B. einer Orchesterstimme – bedeutet. Die Art bzw. das Format der Sequenz muss zu Beginn des Spines in einem mit „`**`“ beginnenden *exclusive interpretation record* festgelegt werden („`**Tonh`“), die abschnittsweise

[20] Es gibt eine unüberschaubare Vielzahl von Kleinstsystemen, die zumeist von Musikwissenschaftlern erstellt wurden, um spezifische Analyseaufgaben zu erleichtern (vgl. z.B. [Alp74], [Sol84], [OMa92], [BH93] und [Net98b]). Für diese Systeme besteht oft nur unzureichende Dokumentation und es nicht klar, ob sie weiterhin gepflegt werden oder nicht. Aus diesen Gründen werde ich diese Systeme nicht weiter betrachten.




```
!!!COM: Mildred and Patti Hill
!!!OTL: Happy Birthday
**kern          **silbe     exclusive interpretation records
*clefG2         *           Violinschlüssel
*k[]            *           keine Tonartversetzungszeichen
*M4/4           *           4/4-Takt
*C:             *           C-Dur
!re A4=440Hz    !(lyrics)   Spine-lokale Kommentare
=1              =1          1. Takt
8.g/L           Hap-
16g/Jk          -py
4a/             birth-
4g/             -day
4cc\            to
=2              =2          2. Takt
2b\             you ;
8.g/L           hap-
16g/Jk          -py
4a/             birth-
=3              =3          3. Takt
4g/             -day
4dd\            to
2cc\            you ;
*-              *-
```

Abb. 3.3-1 *Happy Birthday* als aus einem *Kern*- und einem *Silbe*-Spine bestehende *Humdrum*-Datei.

Zu Beginn der Datei befinden sich die globalen Kommentare, die Auskunft über Komponisten und Titel geben. Danach teilt sich der Inhalt auf die zwei Spines in *Kern* und *Silbe* auf, die durch Taktstriche (=) gegliedert sind. *Kern* enthält zu jeder Note die Dauer in reziproker Notation (4 steht für eine Viertelnote), die Tonhöhe (wobei die Oktave durch die Anzahl der Tonbuchstaben kodiert ist) sowie zusätzlich rudimentäre Notensatzinformationen (z.B. `/` Hals nach oben, `\` Hals nach unten, `L` Querbalkenanfang, `J` Querbalkenende).

um beliebig viele, mit „`*`“ beginnenden *tandem interpretation records* ergänzt werden kann („`*M4/4`“ für Viervierteltakt und „`*Ivioln`“ für Violine). Eine Datei kann beliebig viele Spines besitzen, die durch Tabulator-Zeichen getrennt sein müssen.

Kern[21] ist das zentrale *Humdrum*-Format, das die elementarsten Aspekte der graphischen Repräsentation einer Orchesterstimme in CMN erlaubt. Für eine ansprechende graphische Partiturdarstellung polyphoner Werke reichen die *Kern*-Layout-Elemente jedoch nicht aus (vgl. [Hur97]).

- Zum **Exzerpieren** eines beliebigen Partiturausschnitts können die *Humdrum*-Werkzeuge **`extract`** und **`yank`** verwendet werden: Mit **`extract`** werden aus der die gesamte Partitur enthaltenden Datei die Spines der gewünschten Orchesterstimmen in

[21] Bei konsequenter Einhaltung der *Humdrum*-Syntax müsste stets „`**kern`“ geschrieben werden. Der besseren Lesbarkeit halber wird jedoch im weiteren Verlauf nach einmaliger Erwähnung der korrekten Bezeichnung im Folgenden immer die kursive, „`**`“-lose Form „*Kern*“ verwendet.

voller Länge extrahiert und mit **`yank`** auf den gewünschten zeitlichen Abschnitt „zurechtgeschnitten“.

- Zum **Vergleichen und** zum **Suchen** offeriert *Humdrum* die Werkzeuge **`patt`**, **`pattern`** und **`simil`**. Während die ersten beiden starre textbasierte Mustervergleiche durchführen, sind mit **`simil`** die in Abschnitt 3.2.1 geforderten, auf Ähnlichkeit beruhende Vergleiche möglich. Der Grad der Ähnlichkeit wird dabei anhand der Anzahl der Editierschritte bestimmt („Einfügen“, „Löschen“, etc.), die ausgeführt werden müssten, um die eine Folge von musikalischen Daten in die andere zu transformieren (vgl.[HO92]).

- Die **Synthese von Akustikdaten** ist in *Humdrum* mit einer Kombination der Werkzeuge **`midi`** und **`smf`** möglich. **`midi`** akzeptiert z.B. *Kern*-Dateien und wandelt sie in das *Humdrum*-Format „`**midi`“ um, das bereits alle strukturellen Charakteristika des zur Instrumentenansteuerung gedachten MIDI-Formats[22] besitzt, aber noch als *Humdrum*-ASCII-Datei vorliegt und mit *Humdrum*-Werkzeugen bearbeitet werden kann. Mit **`smf`** können dann *Midi*-Dateien direkt in Dateien im eigentlichen binären MIDI-Format umgewandelt werden. Diese sind dann entweder von einem MIDI-Instrument oder einer MIDI-fähigen Soundkarte des Rechners sofort abspielbar. Java unterstützt die Wiedergabe von MIDI-Dateien seit Version 1.3.

Hinsichtlich der gestellten Anforderungen weist *Humdrum* in der gegenwärtig offiziell verfügbaren Version 1.2 Lücken im Bereich der Unterstützung der Set Theory auf.[23] Um auch diese Teile des MDV-Systems nicht selbst erstellen zu müssen und Set-Theory-Daten auch über das Analysesystem hinaus einfach weiterverwenden zu können, bietet sich der Einsatz des 1986 von Alexander Brinkman und Craig Harris entwickelten *Contemporary Music Analysis Package* (*CMAP*) an, das unter den MDV-Systemen zur Unterstützung der Set-Theory-Analyse den größten Funktionsumfang besitzt. Insbesondere ist es mit den *CMAP*-Werkzeugen **`getset`** und **`Kh`** möglich, die in Abschnitt 3.1.3 geforderte Unterstützung bei der Bestimmung der Set Class sowie bei der Berechnung der Kh-Relation zu leisten (vgl. [Cas94, S. 83f]). Da *CMAP* ebenso wie *Humdrum* rein mit ASCII-Daten arbeitet, aber im Gegensatz zu *Humdrum* kein eigenes Partiturdatenformat definiert (vgl. [Cas94, S. 77ff]), sind *CMAP*-Werkzeuge leicht in ein *Humdrum*-MDV-Subsystem integrierbar.

Aus funktionaler Sicht ist damit die Kontextspezifikation für die *verwendete Technik* vollständig: Die Anwendungssysteme bedienen sich intern *Humdrums* und *CMAPs*, um die analytische Funktionalität zu realisieren und um Akustikdaten zu synthetisieren, die dann mit Hilfe von Operationen der Java Virtual Machine wiedergegeben werden können. Lediglich die Operationen zur Darstellung der Partituren und Exzerpte müssen als komplette Eigenleistung entworfen und implementiert werden.

[22] Das Musical Instrument Digital Interface ist eine standardisierte Hardwareschnittstelle zur Verbindung von elektronisch ansteuerbaren Musikinstrumenten. Das MIDI-Format legt dabei fest, in welcher Form die Daten ausgetauscht werden.[Sel97a, S.6]

[23] Inoffiziell ist auf Anfrage bereits Version 2.3 zu haben, die auch zahlreiche Werkzeuge zur Analyse von für die Set Theory relevanten Strukturelemente bereitstellt. David Hurons Versionsfreigabepolitik erlaubt die offizielle Freigabe aber erst, nachdem die neue Version auf möglichst vielen Plattformen getestet worden ist, so dass mit einer Verfügbarkeit in naher Zukunft nicht zu rechnen ist.

Abgesehen von den in Abschnitt 3.3.2 behandelten Fragen des verwendeten Datenformats, verbleibt aus ökonomischer Sicht jedoch ein weiterer diskussionsbedürftiger Punkt, der mit der Forderung nach Plattformunabhängigkeit aus Abschnitt 3.1.4 zusammenhängt: *Humdrum* und *CMAP* sind beide nur in einer UNIX-Shell voll nutzbar, so dass auf Nicht-UNIX-Systemen zusätzliche Software notwendig ist, um diese UNIX-Shells zu emulieren.[24] Obwohl sowohl *Humdrum* und *CMAP* kostenfrei beziehbar sind,[25] gilt dies nicht für die Emulationssoftware von Mortice Kern Systems (MS-Windows) und Tenon Intersystems (MacOS)[26], die mit nicht unter DM 800 zu Buche schlagen.

Die finanziellen Anforderungen lassen sich zusammen mit Anforderungen an CPU-Zeit, Platz im Hauptspeicher und auf Datenträgern unter dem Begriff „Ressourcen“ subsumieren. Da sich keine einheitlichen Aussagen darüber treffen lassen, welche Ressourcen bestimmten Musikwissenschaftlern in welchem Umfang zur Verfügung stehen (siehe Abschnitt 3.1.4), sollte das Kernanalysesystem in drei unterschiedlichen technischen Varianten nutzbar sein:

(1) Mit einem MDV-Subsystem, das auf *Humdrum* und *CMAP* zurückgreift, die in einer echten oder emulierten UNIX-Shell laufen. Auf allen Plattformen entsteht zusätzlicher Ressourcenbedarf. Unter MS-Windows und MacOS werden außerdem kostenpflichtige UNIX-Shell-Emulatoren benötigt.

(2) Eine Variante ohne *Humdrum* und *CMAP* für diejenigen Anwender, denen keine zusätzlichen Ressourcen zur Verfügung stehen bzw. die diese bewusst nicht nutzen wollen.

(3) Eine Variante, in der die von *Humdrum* und *CMAP* erbrachte Funktionalität von einem zukünftigen MDV-System implementiert wird, dessen Ressourcenbedarf geringer ist als bei Variante (1).

Da MS-Windows-Varianten auf PCs und Workstations die mit am Abstand am häufigsten installierten Betriebssysteme sind[27] und gleichzeitig von knappen Ressourcen in musikwissenschaftlichen Instituten auszugehen ist, ist es somit für die Praxistauglichkeit des Systems entscheidend, dass selbst in Variante (2) ohne *Humdrum* und *CMAP* ein Höchstmaß an Funktionalität zur Verfügung steht. Die Lösung dieses technischen Entwurfsproblems, einerseits existierende Subsysteme maximal zu nutzen und andererseits auch in deren Abwesenheit noch ein Höchstmaß an Funktionalität zu liefern, diskutiere ich ausführlich in Kapitel 6.

Damit die Partitur- und Analysedaten unabhängig von einem Wechsel der Plattform und damit eventuell auch des MDV-Subsystems verwendet werden können, muss sicherge-

[24] Für *CMAP* existiert zwar eine Portierung auf MacOS (vgl. [Cas94, S. 96]), doch da *CMAP* die geforderte Funktionalität nicht alleine bereitstellen kann, reicht dies für ein brauchbares MDV-Subsystem nicht aus.
[25] Informationen zum Bezug des *Humdrum*-Toolkits stehen auf einer Website zur Verfügung:
`http://www.music-cig.ohio-state.edu/Humdrum/FAQ.html#Copy`.
CMAP kann augenblicklich nur direkt von Peter Castine bezogen werden.
[26] MKS Toolkit Sales, 185 Columbia Street West, Waterloo ON N2L 5Z5, Canada
`http://www.mks.com/solution/tk`; Preis: USD 399 (ca. DM 900)
Tenon Intersystems, 1123 Chapala Street, Santa Barbara, CA 93101, USA
`http://www.tenon.com/products/machten`; Preis: USD 348 (ca. DM 800) inkl. Service für ein Jahr
[27] MS-Windows-Systeme waren Anfang 2000 auf fast 90% aller PCs und Workstations installiert (vgl. [OSU00]).

stellt werden, dass diese Daten stets in einem einheitlichen Format vorliegen. Die Wahl eines solchen Formats ist die letzte Entwurfsentscheidung, die im Rahmen des Kontexts *verwendete Technik* diskutiert werden muss.

3.3.2 Musikdatenformate

Bereits zu Beginn des letzten Abschnitts habe ich die vier verschiedenen Arten von Musikdatenformaten diskutiert (Akustik-, Akustiksynthese-, Notensatz- und Analyseformate) und daraufhin gezeigt, dass das geforderte einheitliche interne Musikdatenformat der Analysesysteme sowohl die Charakteristika eines Notensatzformats als auch eines Analyseformats aufweisen muss:

- Die Eigenschaften eines *Analyseformats* sind notwendig, um die im Kontext *Anwendungsbereich* geforderte funktionale Unterstützung beim Suchen, Vergleichen, Exzerpieren sowie bei bestimmten Berechnungen gewährleisten zu können (siehe Abschnitt 3.1.4).

- Die Eigenschaften eines *Notensatzformats* sind notwendig, um die im Kontext *Handhabung & Präsentation* geforderte Darstellung der Partituren und Exzerpte zu ermöglichen (siehe Abschnitt 3.2.2).

Das aus funktionalen Gründen in Abschnitt 3.3.1 bevorzugte *Humdrum-Kern* besitzt jedoch nur rudimentäre Notensatzinformationen, die für eine ansprechende Darstellung von Partituren und Exzerpten nicht ausreichen (siehe S. 63). Umgekehrt ähneln Notensatzformate in ihrer Struktur grundsätzlich graphischen Seitenbeschreibungssprachen wie PostScript, die das Layout zu Ungunsten der logischen Struktur in den Vordergrund stellen (vgl. [Smi97], [Dyd97] und [CS97, S. 585f]). So kann zwar zu jeder Note z.B. die exakte vertikale Position spezifiziert werden, aber die für die Analyse relevante Tonhöhe muss unter Umständen erst umständlich über die Auswertung des Schlüssels, der Tonartversetzungszeichen, der Vorzeichen und der Frage, ob es sich um ein →transponierendes Instrument handelt, ermittelt werden. Da die Notensatzdateien nicht stimmen- sondern seitenorientiert aufgebaut sind, wird die partiturweite Suche in einer bestimmten Stimme durch die zahlreichen Seitenumbrüche erheblich erschwert.

Da somit weder reine Analyseformate noch reine Notensatzformate als Datenformat für die Analysesysteme geeignet sind, müssen Mischformate in Betracht gezogen werden, die qualitativ die Ansprüche beider Formatarten erfüllen. Für die Praxistauglichkeit der Analysesysteme ist es zudem wichtig, dass zusätzlich die anfangs genannten ökonomischen Randbedingungen beachtet werden:

(1) Anwendern muss ein möglichst großer Vorrat an analysefähigen Partituren zur Verfügung stehen, da aus Anwendersicht selbst die beste Rechnerunterstützung wertlos ist, wenn es keine Daten gibt, mit denen gearbeitet werden kann. Die Notensatzformate, in denen die meisten Partituren vorliegen, sind diejenigen, die von Musikverlagshäusern wie Bärenreiter, C.F. Peters und Schott's Söhne verwendet werden:

   - *DARMS* (*Digital Alternate Representation of Musical Scores*) existiert bereits seit 1963 und hat zahlreiche Dialekte entwickelt, die die Konvertierung erschwe-

ren (vgl. [Sel97b, S. 163]). Zum jedem dieser Dialekte gibt es spezielle Eingabeprogramme wie z.B. *The Note Processor* für *N-P-DARMS* (vgl. [Dyd97, S. 192]).

- *SCORE* geht auf das Jahr 1971 zurück und wird als externes Format des gleichnamigen Notensatzprogramms *SCORE* verwendet (vgl. [Smi97, S. 252]).

- *ETF* (*Enigma Transportable File Format*) ist das externe Format des Notensatzprogramms *Finale* und wurde erstmals 1987 eingesetzt. *ETF*-Daten können mit *FinalSCORE* nach *SCORE* konvertiert werden (vgl. [CS97, S. 585f]).

(2) Damit die Analysesysteme keine Insellösung darstellen, sollten die Analyseergebnisse sowohl zu Analysezwecken als auch zu Veröffentlichungszwecken möglichst in solchen Formaten vorliegen, die unmittelbar mit Standardsoftware weiterverarbeitbar sind. Als Standardsoftware dürfen dabei die oben erwähnten Notensatzprogramme sowie das *Humdrum-Toolkit* gelten, das zunehmenden als Standardplattform für musikwissenschaftliche Forschung eingesetzt wird (vgl. z.B. [ST96]).

Die qualitativen Anforderungen an Mischformate erfüllen nur *MuseData* und *Humdrum-Layout*:

- *MuseData* wird seit 1982 am Center for Computer Assisted Research in the Humanities beim Aufbau einer Musikdatenbank eingesetzt (vgl. [Sel97a, S. 34] und [Hew97, S. 405]). Es ordnet die Daten ähnlich wie *Kern* tabellarisch an, wobei einige Spalten Layout- und andere Spalten analytischen Informationen vorbehalten sind.

  Aus ökonomischer Sicht erscheint *MuseData* wenig vorteilhaft, da es außerhalb des CCARH bis 1997 praktisch unbekannt gewesen ist und von keiner Standardsoftware verwendet wird.

- *Humdrum-Layout* ist ein von mir im Rahmen meiner Diplomarbeit entwickeltes *Humdrum*-Format, das es ermöglicht, auch in *Humdrum* qualitativ hochwertige Layoutinformationen zu repräsentieren (vgl. [Kor96a, S. 77ff]). Zu jedem analytischen *Kern*-Datensatz können dabei in einem dazu synchronen Datensatz in einem parallelen „`**layout`“-Spine eine beliebige Menge an Layoutinformationen festgehalten und somit die begrenzten Ausdrucksmöglichkeiten des *Kern*-Formats überwunden werden.

  Ebenso wie *MuseData* ist *Humdrum-Layout* bisher wenig verbreitet. Im Gegensatz zu *MuseData* existiert aber – ebenfalls im Rahmen meiner Diplomarbeit erstellte – Software, mit deren Hilfe die geforderte Verbindung zu Standardsoftware hergestellt werden kann:

  1. *Import von Notensatzdaten*. Als einziges bekanntes Konvertierungsprogramm ist *scr2hmd* in der Lage, Notensatzdaten in Analysedaten umzuwandeln (vgl. [Sel97c, S. 547]). Da das Quelldatenformat *scr2hmds SCORE* ist und *SCORE*-Daten sowohl aus *ETF*-Daten als auch – über ein ebenfalls von mir entwickeltes Programm – aus *MuseData*-Daten gewonnen werden können (siehe oben und

[CCA00]), stehen somit per *scr2hmd* alle in *SCORE*, *ETF* und *MuseData* gesetzten Partituren zu Analysezwecken zur Verfügung.

2. *Export von Notensatzdaten*. Mit einem weiteren Programm aus dem *scr2hmd*-Paket können *Humdrum-Layout*-Daten zurück in *SCORE*-Daten konvertiert werden. Diese Exzerpte können dann entweder mit *SCORE* nachbearbeitet oder unmittelbar mit dem kostenfrei beziehbaren Programm *ScorePreview*[28] in PostScript-Dateien umgewandelt und in zur Veröffentlichung vorgesehene Dokumente eingebunden werden.

3. *Export von Analysedaten*. Ein Export ist nicht notwendig, da die *Humdrum-Kern*-Daten durch den parallelen *Humdrum-Layout*-Spine lediglich ergänzt, nicht aber modifiziert werden und somit sofort verwendbar sind. Um externen Speicherplatz zu sparen, kann der *Layout*-Spine jedoch jederzeit mit **`extract`** entfernt werden (siehe S. 63).

3.3.3 Kontextspezifikation

Mit dieser Entscheidung für eine Kombination aus *Humdrum-Kern* und *Humdrum-Layout* als internes Datenformat der Analysesysteme, habe ich auch für den letzten Kontext des Entwicklungsprozesses *verwendete Technik* sämtliche Einflüsse hinsichtlich der (a) besonderen Anforderungen musikwissenschaftlicher Analyseanwendungen sowie (b) deren Gemeinsamkeiten und Unterschiede diskutiert und (c) herausgearbeitet, wie diese Unterschiede sich zu denjenigen der Kontexte *Anwendungsbereich* und *Handhabung & Präsentation* verhalten:

(a) Eine Kombination aus dem Java-basierten *JWAM* mit den MDV-Systemen *Humdrum* und *CMAP* erlauben ein Maximum an Wiederverwendung bestehender softwaretechnischer Entwürfe und Funktionalität. Eine Mischung aus *Humdrum-Kern* und *Humdrum-Layout* erfüllt zudem die funktionalen Anforderungen am besten und maximiert die praktische Brauchbarkeit für die Anwender.

(b) Isoliert betrachtet ergeben sich für den Kontext *verwendete Technik* aus rein softwaretechnischer Sicht ausschließlich *Gemeinsamkeiten* für alle denkbaren Analysesysteme, so dass eine Anwendungsfamilie leicht realisierbar ist. Angesichts der sozialen und technischen Einbettung ergeben sich jedoch drei *unterschiedliche* Konfigurationen, die davon abhängen, welche Ressourcen (Speicherplatz, CPU-Zeit, Kosten für UNIX-Shell-Emulator, etc.) Anwender aufzuwenden bereit bzw. in der Lage sind. Aus diesem Grund müssen die Analysesysteme so modelliert werden, dass sie bei minimaler Einschränkung der Funktionalität auch ohne *Humdrum* und *CMAP* sinnvoll einsetzbar sind.

(c) Da die Motivationen für die Unterschiede aller drei Kontexte voneinander unabhängig sind (Analyseart im *Anwendungsbereich*, Partiturart in *Handhabung & Präsentation* sowie finanzielle Überlegungen in *verwendete Technik*), bilden sie eine geeignete Grundlage für die Modellierung dreier voneinander unabhängiger Anpassungsstellen des Kernsystems einer Anwendungsfamilie.

[28] `http://ace.acadiau.ca/score/SCOREPREV/SCOREPREV.HTM`

3.4 Skizze eines Kernanalysesystems

Um feststellen zu können, ob sich verschiedene Anwendungssysteme zur Unterstützung der musikwissenschaftlichen Analyse als eine Anwendungsfamilie mit Kernsystem, Anpassungsstellen und Komponenten auffassen lassen, habe ich in den bisherigen Abschnitten dieses Kapitels in Ermangelung bestehender Analyseprogramme die Einflüsse charakterisiert, die auf solche Anwendungssysteme einwirken. Die dabei individuell für die einzelnen Entwicklungskontexte identifizierten Gemeinsamkeiten und voneinander unabhängigen Unterschiede kombiniere ich in diesem Abschnitt konstruktiv zur Skizze konkreter Anwendungssysteme und zeige, dass diese sich als eine Anwendungsfamilie mit voneinander unabhängigen Anpassungsstellen auffassen lassen.

In den Abschnitten 3.4.1 und 3.4.2 entwickle ich zunächst zwei neuartige Gestaltungsansätze für die zentralen Materialien Partitur und Notiz. Die Beschreibung dieser Neuentwicklung ist für die Skizze des Kernanalysesystems notwendig, da kein existierendes musikwissenschaftliches Analysesysteme eine graphische Handhabung und Präsentation dieser Materialien ermöglicht und ich daher nicht – wie bei der Beschreibung der Beispiel-Anwendungsfamilien in Kapitel 2 – zum Zwecke der Erklärung auf Analogien mit bekannten Anwendungen aufbauen kann. Im darauffolgenden Abschnitt 3.4.3 fokussiere ich zunächst auf die funktionalen Aspekte des Kernanalysesystems und beschreibe die Werkzeuge, die es den Anwendern ermöglichen, die Materialien mit der in Abschnitt 3.1.4 geforderten maximalen Rechnerunterstützung zu bearbeiten. In Abschnitt 3.4.4 wende ich mich darauf aufbauend den Anwendungsfamilien-bezogenen Aspekten des Systems mit seinen in den Kontexten *Anwendungsbereich*, *Handhabung & Präsentation* und *verwendete Technik* beobachteten Unterschieden zu und identifiziere und klassifiziere die Anpassungsstellen des Kernsystems gemäß den in Kapitel 2 entwickelten Kategorien.

3.4.1 Das Material Partitur

Gedruckte Partituren sind aus ökonomischen Gründen meistens so gesetzt, dass sie sich auf einer minimalen Anzahl von Seiten bzw. Blättern unterbringen lassen. Zu diesem Zweck werden pausierende Stimmen im Allgemeinen nicht notiert und der so gewonnene Platz genutzt, um mehrere →Akkoladen auf einer Seite drucken zu können (siehe Abb. 3.1-1 auf S. 26). Da die aktiven Stimmen innerhalb eines Stücks rasch wechseln können, sind die in einer Akkolade zusammengefassten Stimmen unter Umständen jedesmal unterschiedlich.

Im nebenstehenden *Rheingold*-Beispiel (siehe Abb. 3.4-2) passen auf die erste Seite noch drei Akkoladen, während die zweite Seite aufgrund weiterer einsetzender Stimmen nur noch für eine Akkolade ausreicht. Ab Seite 8 muss der Stimmenfülle sogar durch quer über zwei Seiten gedruckte Akkoladen Rechnung getragen werden.

Bei einer *mimetischen Partiturdarstellung* in den Softwarewerkzeugen bleibt das Layout der gedruckten Partitur erhalten. D.h. die Partitur bleibt weiterhin in Seiten aufgeteilt und kann – ähnlich wie mit *ghostview* oder dem *Acrobat Reader* – dementsprechend seitenweise traversiert werden.

Neben den *Vorteilen* der einfachen Umsetzung, der unmittelbaren Wiedererkennbarkeit und des minimalen Platzbedarfs hat diese Partiturdarstellung die folgenden *Nachteile*:

- **Sie ist unpraktisch, um darauf Markierungen anzubringen.** Die Umbrüche zwischen Akkoladen sind rein durch die Seitengröße bestimmt und korrespondieren in keiner Weise mit der Struktur des Hörereignisses. Ein akkoladen- oder sogar seitenumbruchübergreifend gesetztes Thema, Leitmotiv oder Set muss dementsprechend unübersichtlich in zwei Teilschritten markiert werden. Unabhängig von den Umbrüchen sind oft mehrere Stimmen in einem →System notiert (in Abb. 3.4-2 mit ≡ markiert), so dass die einzelnen Bestandteile (z.B. melodisch, harmonisch, rhythmisch; siehe S. 33) nicht einfach durch rechteckige Bereiche, sondern nur durch Einzelauswahl individueller Noten markiert werden können.

- **Das Traversieren der Partitur ist unübersichtlich.** Wie oben dargelegt, können sich die vertikalen Positionen von Stimmen von Seite zu Seite ändern. Da Softwarewerkzeuge aufgrund der begrenzten Auflösung von Monitoren nur ein Teil einer Seite in brauchbarer Vergrößerungsstufe darstellen können, müssen sich die Anwender stets vertikal neu orientieren, wenn sie durch die Partitur blättern.

Im Gegensatz dazu wird bei der von mir vorgeschlagenen *kanonischen Partiturdarstellung* das vom Druck her bekannte Layout so modifiziert, dass stets alle – auch die pausierenden – Stimmen notiert und vormals in einem System gemeinsam gesetzte Stimmen auf einzelne Systeme aufgeteilt werden. Sie entspricht damit einem Layout, das der in Abb. 3.4-2 dargestellten Instrumentationsskizze entspricht. Ihre *Vorteile* sind:

- **Die Traversierung ist übersichtlich**, da jede Stimme stets ihre vertikale Position beibehält.

- **Markierungen können praktisch angebracht werden**, da keine rein Layoutbedingten Umbrüche und gemeinsam notierte Stimmen einer ununterbrochenen Blockmarkierung im Wege stehen.

Ihre *Nachteile* bestehen in höherem Platzbedarf und einer aufwendigeren Umsetzung, da zur Taktart passende Pausensymbole für die vormals nicht angezeigten Stimmen eingefügt werden müssen.

Der Platzbedarf ist jedoch anders als für gedruckte Partituren unkritisch, da er keine zusätzlichen Kosten verursacht. Der höhere Aufwand kann durch geeignete Konvertierungsprogramme wie *scr2hmd* problemlos bereits im Vorfeld der Analyse abgefangen werden. Aufgrund der die Nachteile überwiegenden Vorteile verwende ich im Folgenden eine kanonische anstelle einer mimetischen Partiturdarstellung.

3.4.2 Das Material Notiz[29]

Aus der Beschreibung der Tätigkeiten bei der Einzelwerk- und der vergleichenden Analyse (siehe S. 43) geht hervor, dass Notizen primär dazu dienen, um deren Exzerpte mit anderen Partiturstellen oder weiteren Notizen zu verglei-

[29] Der nachfolgende Abschnitt beruht weitgehend auf meinem Artikel [Kor98a].

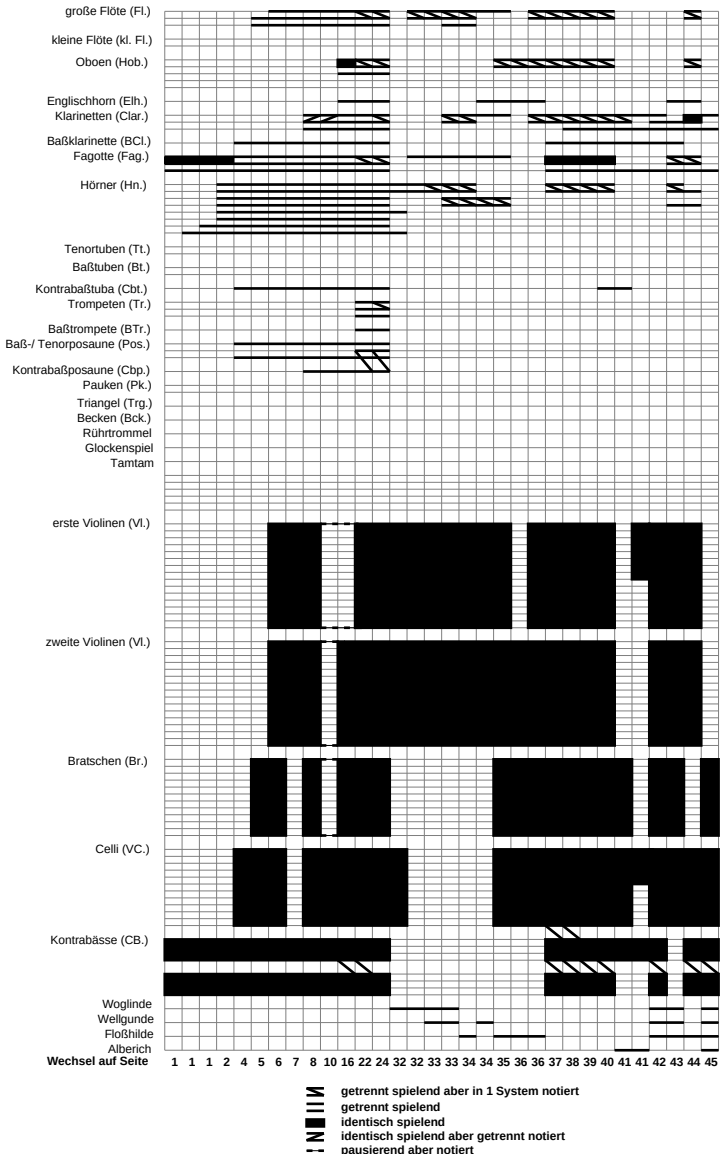


Abb. 3.4-2 Instrumentationsskizze des Anfangs derselben Partitur: Vorspiel (Seiten 1-31) und 1. Szene (Seiten 32-45)

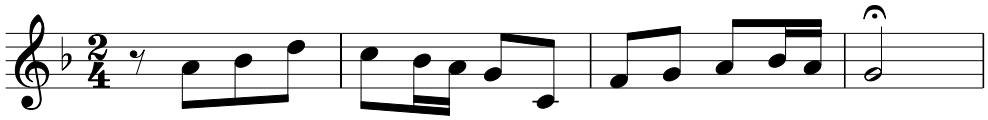
Die Anzahl der gleichzeitig aktiven Stimmen steigt zwar zunächst stetig an, schwankt danach aber fortwährend von 7 Stimmen auf Seite 36 bis zu 37 Stimmen auf Seite 24. Die Stimmen sind daher auf so gut wie jeder Seite vertikal unterschiedlich angeordnet.

chen. Wenngleich im letzteren Fall auch alphanumerische und ordinale Notizbestandteile eine Rolle spielen (Bezeichnung, Variantennummer, Position in der Partitur), so bilden die musikalischen Notizbestandteile doch den Kern der Vergleichstätigkeit.

Wie ich in Abschnitt 3.2.1 dargelegt habe, ist dabei eine absolut identische Wiederkehr eines Strukturelements sehr unwahrscheinlich, so dass als Grundlage für Vergleiche abstraktere Formen der Strukturelementbestandteile alleine oder in Kombination herangezogen werden müssen (siehe S. 56). Die Möglichkeiten zur Bildung solcher Abstraktionen sind unbegrenzt und ein wichtiger Gegenstand musikwissenschaftlichen Forschens.[30] Eine Abstraktion muss lediglich das Kriterium erfüllen, unzweideutig aus mu-

[30] Siehe beispielsweise die Diskussion um die verschiedenen Arten von Set Classes in Abschnitt 2.1.3.

sikalischem Material ableitbar und somit in einem wissenschaftlichen Diskurs verwendbar zu sein. Die folgende Aufstellung enthält am Beispiel des ersten Themas aus Ludwig van Beethovens *Symphonie Nr. 6* acht gebräuchliche Abstraktionen des melodischen Bestandteils eines Themas.



	Mögliche textuelle Repräsentation[31]
Absolute Tonfolge	a1 h1- d2 c2 h1- a1 g1 c1 f1 g1 a1 h1- a1 g1
Tonfolge	a h- d c h- a g c f g a h- a g
Pitch-Class-Folge	9 A 2 0 A 9 7 0 5 7 9 A 9 7
Tonstufen	3 4 6 5 4 3 2 5 1 2 3 4 3 2
Tonales Intervall	* K2 G3 g2 g2 k2 g2 p5 P4 G2 G2 K2 k2 g2
Halbton-intervall	* +1 +4 -2 -2 -1 -2 -7 +5 +2 +2 +1 -1 -2
Feinkontur	* ^ / v v v v \ / ^ ^ ^ v v
Grobkontur	* / / \ \ \ \ \ / / / / \ \

Abb. 3.4-3 Die Melodische Komponente des ersten Themas aus Ludwig van Beethovens *Symphonie Nr. 6* und acht verbreitete Abstraktionen.

Diese und zahlreiche weitere Abstraktionen sind für Musikwissenschaftler implizit bereits im Exzerpt enthalten und bedürfen bei einer manuellen Analyse keiner expliziten Ausformulierung. Sie werden sofort erkannt und können – einzeln oder in Kombinationen – als Grundlage für Vergleiche und Suchen dienen.

Soll jedoch ein Rechner eingesetzt werden, um die Anwender bei ihren Analysetätigkeiten zu unterstützen, ergeben sich zwei Schwierigkeiten:

1. Im Gegensatz zu einem Musikwissenschaftler kann ein Rechner nicht erahnen, welche musikwissenschaftlichen Abstraktionen sich in einem Exzerpt verbergen. Werden die Abstraktionen nicht spezifiziert, kann ein Rechner – ebenso wie ein Laie (siehe Seite 54) – nur feststellen, ob der seltene Fall eintritt, dass ein Thema, Leitmotiv oder Set in exakter Form wiederkehrt.

2. Selbst wenn ein Rechner um sämtliche Abstraktionen eines Strukturelements wüsste, so wäre er aufs Raten angewiesen, sobald die Anwender alle ähnlichen Strukturelemente zu sehen wünschen, da es ihm nicht wie einem manuell analysierenden

[31] In der Textrepräsentation sind die Zeichen wie folgt zu deuten. Absolute Tonfolge: die Ziffern bezeichnen die Oktave; - zeigt ein ♭ an. Tonstufen: 1, 4, 5 stehen für Tonika, Subdominante, Dominante, etc. Tonale Intervalle: Großbuchstaben für steigende und Kleinbuchstaben für fallende Intervalle; g, k und p stehen für groß, klein und perfekt; 1, 2, 3 stehen für Prim, Sekunde, Terz, etc. Feinkontur: ^ und v bezeichnen Aufwärts- bzw. Abwärtsschritte von bis zu 2 Halbtönen, / und \ repräsentieren größere Schritte. Grobkontur: / und \ stehen für jede Art von Aufwärts- bzw. Abwärtsschritt. Alle Intervalle und Konturen: * bezeichnet den ersten Ton.

Musikwissenschaftler klar sein kann, hinsichtlich welcher der zahlreich vorhandenen abstrakten Eigenschaften Ähnlichkeit vorliegen soll.

Wenn also wie bei der manuellen Analyse nur das Exzerpt auf dem Notitzzettel vermerkt wird, lässt sich das geforderte Höchstmaß an Rechnerunterstützung nicht realisieren. Aus diesem Grunde schlage ich vor, die von den Anwendern gewünschten musikwissenschaftlichen Abstraktionen als zusätzliche Bestandteile auf dem Notizzettel aufzuführen und somit ein Arbeitsmittel zur Verfügung zu stellen, mit dem die Anwender ihre vormals implizit gemachten Abstraktionen explizit festzuhalten vermögen und bei Vergleichen genau formulieren können, auf welche Art von Ähnlichkeit sie aus sind.

Neben diesen *Vorteilen*, haben die erweiterten Notizzettel auch *Nachteile*:

(1) **Es ist aufwendiger, die Zettel auszufüllen.** Wenn die Anwender bei jedem Markierungsvorgang die neu hinzugekommen Einträge von Hand ausfüllen müssen, wird der Zeitvorteil, der dadurch entsteht, dass das Exzerpt automatisch vom Rechner eingetragen wird, vollkommen aufgezehrt.

(2) **Der Platzbedarf auf dem Bildschirm steigt.** Da auf einem graphisch dargestellten Arbeitsplatz nicht annähernd soviel Platz ist wie auf eine realen Schreibtisch, bedeuten größere Notizzettel, dass zunehmend größere Teile der anderen Arbeitsmittel verdeckt werden bzw. immer weniger Notizzettel gleichzeitig nebeneinander betrachtet werden können.

Für beide Probleme gibt es jedoch eine Lösung, so dass ich bei der folgenden Beschreibung möglicher Werkzeuge davon ausgehe, dass Notizzettel die expliziten Abstraktionen als separate Bestandteile enthalten. Die Lösungen sind:

(1) **Die expliziten Abstraktionen können automatisch eingetragen werden.** Wie ich weiter oben in diesem Abschnitt dargelegt habe, muss es zu jeder Abstraktion ein formal eindeutiges Verfahren geben, um Abstraktionen aus musikalischem Material zu gewinnen. Es ist daher möglich, für diese Verfahren geeignete Algorithmen zu entwickeln, auf die das unterstützende Anwendungssystem zurückgreifen kann, um die betreffenden Notizbestandteile auszufüllen. Mit den Befehlen `midi` und `getset` habe ich in Abschnitt 3.3.1 bereits zwei Beispiele aus *Humdrum* bzw. *CMAP* genannt, die solche Algorithmen verkörpern.

(2) **Die Darstellung der Notizzettel kann komprimiert werden.** Es ist möglich, die Bedienschnittstelle der mit Notizen arbeitenden Werkzeuge so zu gestalten, dass Notizbestandteile, die momentan nicht von Interesse sind, von den Anwendern ausgeblendet oder minimiert werden. Dadurch werden nur die wirklich benötigten Bestandteile angezeigt und der von den Notizzetteln verdeckte Anteil an der Darstellung des Arbeitsplatzes verringert sich. Beispiele hierfür in bestehender Software sind z.B. die „Toolbars“ des *Netscape Navigators*.

3.4.3 Die Werkzeuge

Während geeignet modellierte Materialien die unverzichtbare Grundlage für hilfreiche Softwarewerkzeuge sind, so sind es doch letztendlich die Werkzeuge, die den Kern der arbeitserleichternden Rechnerunterstützung ausmachen. Basierend auf der Beschreibungen der analyseartübergreifenden Tätigkeiten bei der Einzelwerkanalyse (siehe Abschnitt 3.1.3) sowie den Wünschen der Anwender (siehe Abschnitt 3.1.4) aus dem Kontext *Anwendungsbereich* ergibt sich in Verbindung mit den Gemeinsamkeiten der Einflüsse aus den Entwicklungskontexten *Handhabung & Präsentation* und *verwendete Technik* die folgende mögliche Umsetzung der Ist-Tätigkeiten und -arbeitsmittel in Softwarewerkzeuge, -behälter und -materialien.

Mögliche Ausgestaltung eines Kernanalysesystems

Dem Analysator wird eine in *JWAM* implementierte Werkzeugsammlung zur musikwissenschaftlichen Analyse zur Verfügung gestellt, mit der er auf einem elektronischen Schreibtisch mit Partituren, Notizzetteln und -katalogen arbeiten kann, deren Darstellung des Notenbildes dem einer gedruckten Partitur entspricht.

In einem Partituranalysewerkzeug kann die Partitur rein visuell oder mit automatischer Unterstützung nach Strukturelementen durchsucht und Markierungen vorgenommen werden. Entsprechend der Markierung wird in einem Notizeditor ein Notizzettel ausgefüllt, der alle automatisch ermittelbaren Einträge (Exzerpt, Ort des Auftretens, musikwissenschaftliche Abstraktionen, etc.) bereits enthält. Nachdem die fehlenden Einträge vom Analysator ausgefüllt worden sind (Grund des Auftretens, Art der Variierung, etc.), wird die Notiz in den als Behälter fungierenden Katalog aufgenommen, der mit Hilfe eines Katalogverwaltungswerkzeugs anhand einzelner oder beliebig kombinierter Notizbestandteile sortiert und durchsucht werden kann. Jede Notiz kann auch Grundlage für eine ebenso flexible Suche nach ähnlichen Strukturelementen in der Partitur mit dem Partituranalysewerkzeug herangezogen werden.

Auf Betriebssystemebene liegen Partituren und Kataloge in den *Humdrum*-Formaten *Kern* und *Layout* vor, so dass sie unabhängig vom Anwendungssystem anderweitig verwendet werden können.

Dieses Kernanalysesystem entspricht demjenigen des *JRing*-Analysesystem und stellt nur eine von zahlreichen Optionen dar, die sich durch die Einflüsse aus den Entwicklungskontexten eröffnen.[32] In alternativen Entwürfen könnten die Werkzeuge und Subwerkzeuge beispielsweise unterschiedlich abgegrenzt sein oder zusätzliche Dienste anbieten, ebenso wie es in Kapitel 2 möglich gewesen wäre, zur Anschauung anstelle auf den *Netscape Navigator* und *GIMP* auf den *Microsoft Internet Explorer*, *Adobe Photopaint* oder fiktive Web-Browser und Bildbearbeitungssoftware zurückzugreifen. Diese alternativen Entwürfe werde ich im Weiteren nicht betrachten, da abweichende Gestaltungsdetails die beiden Hauptziele dieses Kapitels nicht berühren:

(1) Zu demonstrieren, dass die geforderte maximale Unterstützung für verschiedene Arten der musikwissenschaftliche Analyse tatsächlich möglich ist.

[32] Der Name *JRing* setzt sich aus den Bestandteilen „J“ für Java und „Ring“ in Anlehnung an Wagners *Der Ring des Nibelungen* zusammen. David Huron hatte das aufbauend auf meiner Diplomarbeit in C++ implementierte leitmotivische Analysesystem „Ring“ getauft, da *Der Ring des Nibelungen* das größte leitmotivisch komponierte Werk ist und daher dort die Vorteile der Rechnerunterstützung am klarsten hervortreten.

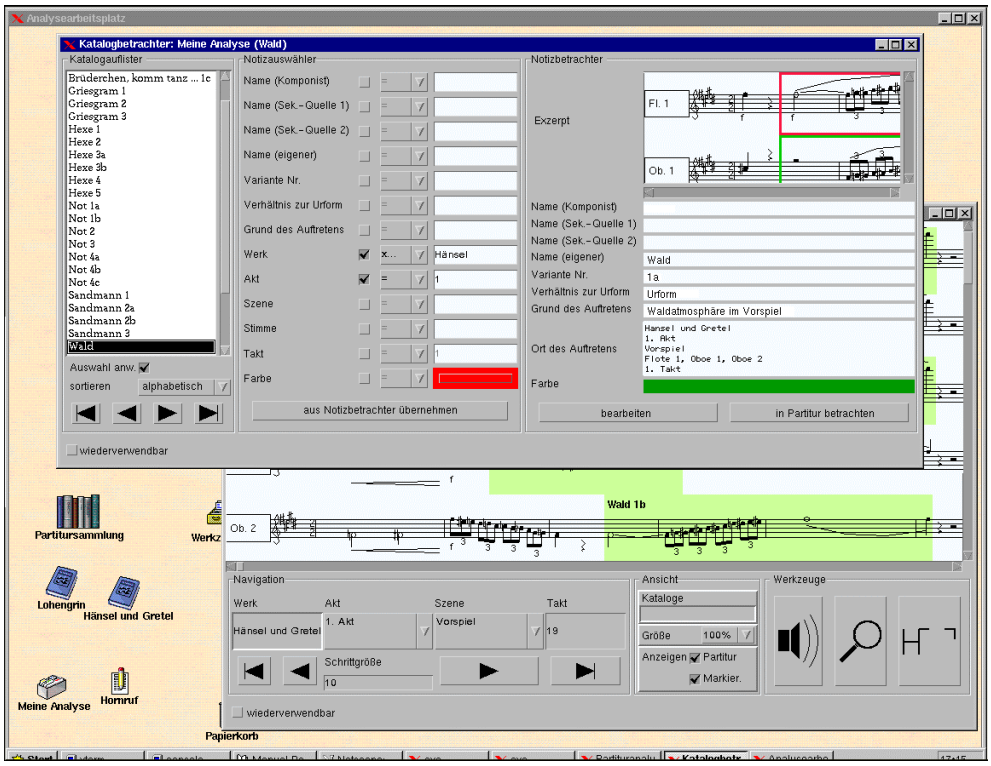


Abb. 3.4-4 Ansicht *JRings* während der leitmotivischen Analyse von Humperdincks *Hänsel und Gretel* in einer CMN-Partitur.

Unten links sind auf der Arbeitsfläche abgelegte Partituren, eine „Hornruf“ benannte Notiz, die im Entstehen begriffene Leitmotivtabelle „Meine Analyse“ und eine Partitursammlung zu sehen. Außerdem sieht man die für Arbeitsflächen üblichen Spezialbehälter Werkzeugkasten und Papierkorb. In der Mitte ist eine Katalogbetrachter mit der Übersichtlichkeit halber nicht angezeigten expliziten Abstraktionen und ein Partituranalysewerkzeug mit kanonischer Partiturdarstellung zu sehen.

(2) Zu zeigen, dass die spezifizierten Anwendungssysteme trotz all ihrer Unterschiede eine Anwendungsfamilie mit Kernsystem, Anpassungsstellen und Komponenten bilden.

Im Rest dieses Abschnitts beschreibe ich die Funktionalität des Analysesystems anhand eines konkreten *JRing*-Familienmitglieds für die leitmotivische Analyse in einer CMN-Partitur, bevor ich in Abschnitt 3.4.4 die drei Anpassungsstellen identifiziere, durch deren Konfiguration sich alle in den Kontexten *Anwendungsbereich*, *Handhabung & Präsentation* und *verwendete Technik* spezifizierten Anwendungsfamilienmitglieder erzeugen lassen.

Neben den weiter unten beschriebenen Katalogbetrachtern sind **Partituranalysewerkzeuge** das Kernstück des Analysesystems. Mit ihnen kann eine kanonisch dargestellte Partitur mit den darin befindlichen Markierungen auf vielfältige Weise betrachtet, traversiert und untersucht werden. Es gibt drei Subwerkzeuge:

- Mit dem **Wiedergabe-Subwerkzeug** kann die Partitur in Gänze oder ab der angezeigten Stelle von CDs oder – falls diese nicht verfügbar sind – in MIDI-Form wiedergegeben werden. Die Partiturdarstellung kann dabei automatisch nachgeführt werden.

- Das **Such-Subwerkzeug** erlaubt es, die Partitur nach einzelnen Kriterien oder beliebigen Kriterienkombinationen zu durchsuchen. Dabei kann für jedes Kriterium separat angegeben werden, ob Treffer exakt sein müssen oder ob bereits ein gewisses Maß an Ähnlichkeit genügt. Ein Trefferauflister stellt die Ergebnisse dar.

- Um neue Markierungen in der Partitur vorzunehmen oder bestehende Markierungen zu modifizieren, dient das **Markierungs-Subwerkzeug**. Mit ihm können die einzelnen melodischen, harmonischen und rhythmischen Leitmotivbestandteile angelegt, verändert oder gelöscht werden.

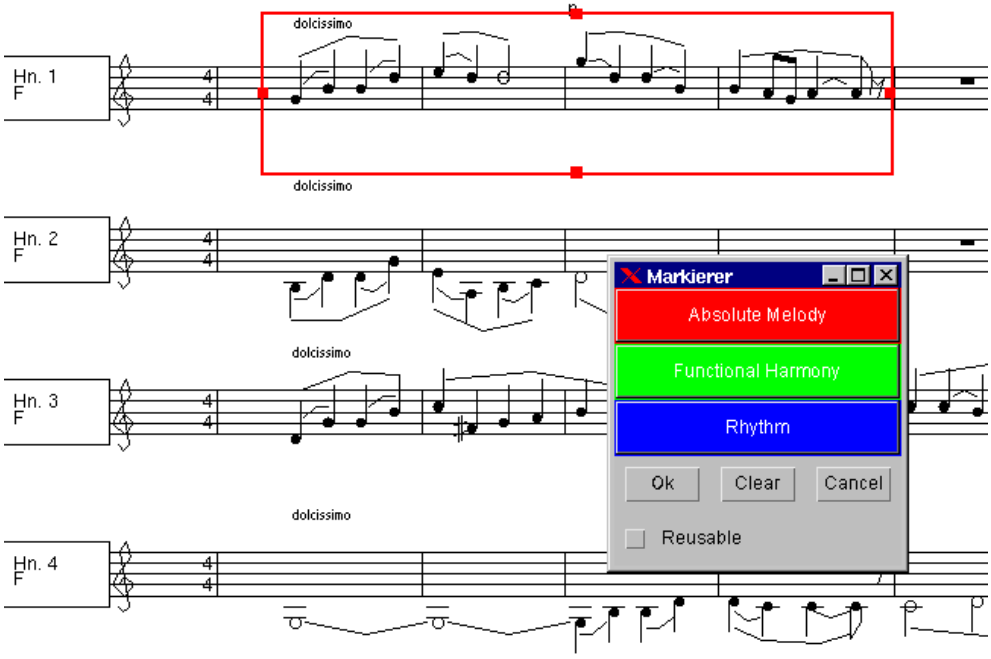


Abb. 3.4-5 Das Markierungs-Subwerkzeug eines Partituranalysewerkzeugs.
Den einzelnen Leitmotivbestandteilen sind unterschiedliche Farben zugeordnet, so dass auf einen Blick sichtbar ist, welche Bestandteile bereits markiert sind.

Der zweite Kernbestandteil des Analysesystems sind **Katalogbetrachter**, mit denen Notizkataloge angezeigt, sortiert und nach Einträgen durchsucht werden können. Sie besitzen drei Subwerkzeuge:

- Der **Katalogauflister** zeigt die Namen der im Katalog vorhandenen Notizen an. Er kann die Liste auf Wunsch alphabetisch oder chronologisch sortieren. Um die Auswahl der angezeigten Notizen einzuschränken, dient der

- **Notizauswähler**. Er ähnelt in seiner Funktionalität dem Such-Subwerkzeug des Partituranalysewerkzeugs, arbeitet aber auf den Notizen des Kataloges anstatt auf der Partitur. Die Ergebnisse werden nicht in einem separaten Trefferauflister, sondern direkt im Katalogauflister angezeigt. Je nachdem, ob der Anwender organisatorische oder wissenschaftliche Ziele verfolgt, kann er so z.B. die Anzeige im Auflister auf alle Leitmotive des ersten Aktes beschränken oder aber all diejenigen Leitmotive auflisten, die bestimmten melodischen Kriterien genügen.[33]

[33] Beide Kriterien können natürlich auch kombiniert werden, um nur diejenigen Leitmotive des ersten Akts aufzulisten, die bestimmte melodische Charakteristika aufweisen.

- Wenn der Anwender einen Eintrag im Katalogauflister auswählt, dann wird die dazugehörige Notiz mit all ihren textuellen und graphischen Bestandteilen im **Notizbetrachter** angezeigt. Bei Bedarf kann sie editiert oder akustisch wiedergegeben werden. Außerdem besteht ein Verweis auf die Original-Partiturstelle, um das Leitmotiv im vollen musikalischen Kontext betrachten zu können.

Da die automatisch ermittelbaren Notizbestandteile wie das Exzerpt, Position und musikwissenschaftliche Abstraktionen bereits automatisch ermittelt werden, müssen Anwender **Notizeditoren** nur noch dazu aufrufen, um diejenigen Merkmale einzutragen, die auf Expertenwissen oder musischem Empfinden beruhen; z.B. Leitmotivnamen gemäß Sekundärquellen, Variantennummer, Art und Grund der Variierung sowie Grund des Auftretens. Besonders praktisch ist die automatische Ermittlung einiger Bestandteile im Fall der Set-Theory-Analyse: Da die Set Class automatisch aus dem Exzerpt berechnet werden kann, kann das bei der manuellen Analyse benötigte explizite Arbeitsmittel der Set-Class-Tabelle, in dem die Anwender die Set Class anhand der zuvor berechneten Prime Form nachschlagen (siehe Abschnitt 3.1.3), vollkommen entfallen.

Mit den beschriebenen Arbeitsmitteln erfüllt das beschriebene Anwendungssystem die Anforderungen aller drei Entwicklungskontexte zur Gänze. Insbesondere wird die gewünschte maximale Unterstützung bei der Notizverwaltung, der Suche und beim Exzerpieren (siehe Abschnitt 3.1.4) gewährleistet, ohne die Anwender in ihren Möglichkeiten einzuschränken oder im Sinne des Leitbilds *Fabrik* zu bevormunden bzw. im Sinne des Leitbilds *Objektwelten* mit unverständlichen technischen Artefakten zu konfrontieren.

3.4.4 Die Anpassungsstellen und Komponenten

Wie ich in Kapitel 2 gezeigt habe, müssen sowohl das Kernsystem als auch dessen Anpassungsstellen und die dazu passenden Komponenten beschrieben werden, um eine Anwendungsfamilie zu charakterisieren. Nachdem ich in den Abschnitten 3.4.1 bis 3.4.3 nun aufbauend auf den Gemeinsamkeiten der Einflüsse aus den Entwicklungskontexten die Materialien und Werkzeuge des Kerns einer Anwendungsfamilie von musikwissenschaftlichen Analysesystemen herausgearbeitet habe, bleibt in diesem Abschnitt noch zu zeigen, wie die in den Abschnitten 3.1 bis 3.3 spezifizierten Unterschiede sich als konkreten Anpassungsstellen und Komponenten charakterisieren lassen. Da sich in der Familie von Analysesystemen im Gegensatz zu den in Kapitel 2 eingeführten Beispielen sämtliche Arten von Anpassungsstellen (Mehrfach-, Einfach und Zwangsanpassungsstellen) sowie Komponenten identifizieren lassen (Baustein- und Dienstleisterkomponenten), dient diese Anwendungsfamilie im weiteren Verlauf dieser Arbeit als Hauptbeispiel für die Diskussion der softwaretechnischen Modellierung von Anwendungsfamilien.

(1) **Anpassungsstelle *Notizbestandteile*** (Kontext *Anwendungsbereich*). Um eine bestimmte Art von Analyse durchführen zu können, müssen die Anwender dafür sorgen, dass sie die für die jeweilige Analyseart relevanten Eigenschaften auch in Notizen festhalten können. Zu diesem Zweck müssen sie dem Kernsystem an der Anpassungsstelle *Notizbestandteile* Komponenten hinzufügen, die geeignet sind, um die gewünschten Eigenschaften zu repräsentieren. Für eine thematische Analyse kann

eine Notiz beispielsweise aus den in Abb. 3.1-6 auf S. 31 auf dargestellten Elementen bestehen (Bezeichnung, Grund des Auftretens, Variantennummer, Variantenvorkommen), zu denen je nach Analyseziel noch beliebige musikwissenschaftliche Abstraktionen hinzukommen können (absolute Tonfolge, Tonfolge, Tonstufen, totales Intervall, Halbtonintervall, Feinkontur, Grobkontur; siehe Abb. 3.4-3 auf S. 72). Für leitmotivische und set-theoretische Analysen ergeben sich die in Abschnitt 3.1.2 bzw. 3.1.3 diskutierten abweichenden Elemente.

Da es von der Auswahl bestimmter Notizbestandteile abhängt, wie Notizen und die mit Notizen arbeitenden Werkzeuge aufgebaut sind, handelt es sich bei den Notizbestandteilkomponenten gemäß der Klassifikation in Kapitel 2 um *Bausteinkomponenten*. Jede dieser Komponenten implementiert dabei (a) einen Algorithmus, mit dem eine bestimmte Abstraktion aus einem gegebenen Exzerpt abgeleitet werden kann (siehe Abschnitt 3.4.2) und (b) einen graphischen Baustein, um den Notizbestandteil darzustellen. Den einzigen Sonderfall bilden Bausteinkomponenten für textuelle Notizelemente: Da sich beispielsweise der Grund des Auftretens eines Leitmotivs nicht algorithmisch aus dem Exzerpt ableiten lässt, implementieren textuelle Notizbausteinkomponenten keinen Algorithmus.

Aufgrund der Tatsache, dass Anwender ihre Notizen aus beliebigen Bestandteilen zusammenstellen können und dementsprechend beliebig viele Notizbausteinkomponenten an der Anpassungsstelle vorhanden sein können, handelt es sich um eine *Mehrfachanpassungsstelle*. Die einzigen Bestandteile, die immer vorhanden sein müssen, sind diejenigen für das Exzerpt, die Position in der Partitur und die Farbe der Markierung. Da sie in sämtlichen Analysearten vorhandenen sind, handelt es sich bei ihnen nicht um Unterschiedlichkeiten verkörpernde Komponenten, sondern um zum Kernsystem gehörende interne Bausteine (siehe Begriff 2.3 auf S. 20).

(2) **Anpassungsstelle *Partitur-Engine*** (Kontext *Handhabung & Präsentation*). Ebenso wie im Beispiel der Familien von Internetanwendungen eine HTML-Engine notwendig ist, um HTML-Daten zu lesen und für die Darstellung entweder im Browser oder im Mail-Client aufzubereiten, so muss auch in jedem Analysesystem eine bestimmte Partitur-Engine-Komponente vorhanden sein, um die Layoutdaten der jeweiligen Partiturart zu lesen und den Partituranalysewerkzeugen sowie Katalogbetrachtern zur Verfügung zu stellen. Neben den in Abschnitt 3.2.2 diskutierten Notationsarten CMN sowie Mensural- und chromatischer AB-Notation sind Partitur-Engines für alle Notenschriften denkbar, für die Layoutdaten verfügbar sind.

Da die graphische Darstellung von Partituren und Exzerpten für Analyseanwendungen genauso essentiell ist, wie die Darstellung von Hypertexten für Internetanwendungen, handelt es bei der Anpassungsstelle *Partitur-Engine* wie auch bei der Anpassungsstelle für HTML-Engines um eine *Zwangsanpassungsstelle*.

Partitur-Engines kontrollieren im Gegensatz zu Notizbestandteilkomponenten keinen eigenen Bereich der Bedienschnittstelle eines Werkzeugs, sondern liefern lediglich als *Dienstleisterkomponente* einen (unverzichtbaren) Teil der zur Darstellung benötigten Daten. Diesen Daten fügen die verschiedenen Werkzeuge noch eigene Layoutdaten hinzu, bevor sie Partituren und Exzerpte darstellen. Notizbetrachter er-

gänzen beispielsweise die Bestandteile des notierten Strukturelements, während Partituranalysewerkzeuge die Markierungen aller Strukturelemente der ausgewählten Kataloge hinzufügen.

(3) **Anpassungsstelle MDV-Subsystem** (Kontext *verwendete Technik*). Je nachdem, welches MDV-Subsystem an der Anpassungsstelle *MDV-Subsystem* vorhanden ist, stehen den Anwendern unterschiedliche musikwissenschaftliche Kriterien zur Verfügung, um Partituren und Kataloge zu durchsuchen bzw. zu sortieren. Falls es sich dabei um eine Kombination aus *Humdrum* und *CMAP* handelt, sind dies beispielsweise die in Abschnitt 3.3.1 diskutierten Möglichkeiten.

MDV-Subsysteme verwalten ebenso wie Partitur-Engines keine eigene Bedienschnittstelle, sondern sind lediglich *Dienstleisterkomponenten*, auf die das Such-Subwerkzeug eines Partituranalysewerkzeugs oder der Notizauswähler eines Katalogbetrachters zurückgreifen kann, um Suchen und Vergleiche durchzuführen. Die MDV-Subsystem-Komponente liefert Ergebnisse zurück, die dann von den aufrufenden Werkzeugen je nach Bedarf aufbereitet werden.

Im Gegensatz zur Anpassungsstelle *Partitur-Engine* muss an der Anpassungsstelle *MDV-Subsystem* nicht unbedingt eine Komponente vorhanden sein, damit das Analysesystem nutzbar ist. Obgleich ohne MDV-Subsystem-Komponente keine Rechnerunterstützung bei musikwissenschaftlichen Suchen, Vergleichen und Berechnungen mehr gewährt werden kann, so können Partituren und Notizen weiterhin auf einfache Weise angefertigt und betrachtet sowie nach Augenschein und Gehör durchsucht werden. Die Anpassungsstelle *MDV-Subsystem* ist daher eine *Einfachanpassungsstelle*.

3.5 Zusammenfassung und Ausblick

In diesem Kapitel habe ich zwei der fünf zentralen Fragestellungen dieser Arbeit beantwortet und die Grundlage dafür gelegt, um die anderen drei zu untersuchen: Ich habe (1) die besonderen Anforderungen herausgearbeitet, die an Anwendungssysteme zur Unterstützung der musikwissenschaftlichen Analyse gestellt werden und (2) gezeigt, dass diese Systeme sich trotz ihrer erheblichen Unterschiede als eine Anwendungsfamilie mit Kernsystem, Anpassungsstellen und Komponenten auffassen lassen.

(1) Die besonderen Anforderungen habe ich gemäß den drei in [Bäu98] identifizierten Kontexten des Softwareentwicklungsprozesses getrennt voneinander untersucht. Im Sinne der Anwendungsorientierung bildete der *Anwendungsbereich* (Abschnitt 3.1) dabei den Ausgangspunkt. Obgleich sich die thematische, die leitmotivische und die Analyse nach der Set Theory in ihrem theoretischen Hintergrund, ihren Analysezielen, ihrer Terminologie, ihrer Gewichtung der Ergebnisse und im Falle der Verwendung existierender Kataloge auch hinsichtlich ihrer Arbeitsmittel unterscheiden,

konnte ich zeigen, dass sie sich auf der Ebene der konkreten Analysetätigkeiten und der Arbeitsmittel Partitur, Notiz und selbsterstelltem Katalog gleichen.

Für die *Handhabung & Präsentation* (Abschnitt 3.2) habe ich zunächst Musikwissenschaftler als im Umgang mit Rechnern im Allgemeinen äußerst unerfahrene Fachleute ihres Gebiets charakterisiert und dann ein adäquates Leitbild und dazu passende Entwurfsmetaphern ausgewählt. Die die Anwender bevormundenden bzw. sie mit rein technisch motivierten Konstrukten konfrontierenden Leitbilder der *Fabrik* und der *Objektwelten* wurden dabei zugunsten einer Spezialisierung des WAM-Leitbilds des *Arbeitsplatzes für eigenverantwortliche Expertentätigkeit* verworfen. Vor dem Hintergrund der Modellierung von Tätigkeiten entweder mit Softwarewerkzeugen oder -automaten habe ich anhand des aktuellen Standes der Musikwissenschaft die Grenzen abgesteckt, innerhalb derer eine Automatisierung der Analyse vertretbar ist.

Im Rahmen der Diskussion des Kontexts *verwendete Technik* (Abschnitt 3.3) habe ich dargelegt, dass es möglich ist, bestehende softwaretechnische Rahmenwerke (*JWAM*) und MDV-Systeme (*Humdrum* und *CMAP*) wiederzuverwenden und somit Analysesystem mit zumutbarem Aufwand zu realisieren. In der Frage des gemeinsam genutzten externen Formats für Partitur- und Analysedaten habe ich anhand der technischen und auf die Verbreitung bezogenen Charakteristika der Formate herausgearbeitet, dass eine Kombination von *Humdrum-Kern* (für den analytischen Inhalt) und des von mir entwickelten *Humdrum-Layout* (für die Notensatzdaten) die günstigste Wahl darstellt.

Die isolierten Einflüsse der einzelnen Kontexte habe ich in Abschnitt 3.4 zu der skizzenhaften Spezifikation eines beispielhaften Analysesystem für die musikwissenschaftliche Analyse kombiniert, das die Einflüsse aus allen Kontexte berücksichtigt. Für die Materialien Partitur und Notiz habe ich dabei zwei neuartige Gestaltungsansätze entwickelt, die die Arbeit mit den Materialien erleichtern bzw. einige Arten der Rechnerunterstützung überhaupt erst ermöglichen. Bei der kanonischen Partiturdarstellung werden die Platzbeschränkungen gedruckter Partituren überwunden und alle Stimmen in stets gleicher vertikaler Abfolge gesetzt, auch wenn sie pausieren. Bei horizontalem Scrolling sind dadurch immer die gleichen Stimmen im Blickfeld. Außerdem können Themen, Leitmotive und Sets auf diese Weise in einem Stück markiert werden, da Umbrüche am Ende von Systemen und Seiten nicht mehr vorkommen. Für Notizen habe ich vorgeschlagen, die im Kopf der Anwender implizit vorhandenen musikwissenschaftlichen Abstraktionen des Exzerpts als explizite Notizbestandteile zu modellieren. Dadurch können die Anwender bei rechnergestützten Suchen und Vergleichen präzise angeben, hinsichtlich welcher Kriterien sie einen Vergleich wünschen und auf welche Weise der Vergleich durchgeführt werden soll. Ohne eine solche explizite Modellierung der Abstraktionen des Exzerpts wären rechnergestützte musikbezogene Vergleichs- und Suchverfahren zwangsläufig aufs Raten angewiesen.

Sowohl die getrennte Beschreibung der Einflüsse aus den drei Entwicklungskontexten als auch die Skizze des Analysesystems bilden eine geeignete theoretische Grundlage für weitere Diskussionen im Rahmen der Musikdatenverarbeitung. Zu-

gleich liegt mit *JRing* eine Analyseumgebung vor, die fast alle Aspekte des skizzierten Systems konkret umsetzt, so dass umfangreiche musikwissenschaftliche Analysen mit vertretbarem durchführbar sind und keine „enormous tasks“ mehr darstellen.

(2) Ich habe belegt, dass die Analysesysteme, die den beschriebenen Einflüssen gerecht werden, eine Anwendungsfamilie bilden, indem ich (a) gezeigt habe, hinsichtlich welcher Eigenschaften sich Anforderungen aus den drei Kontexten unterscheiden und (b) demonstriert habe, dass die Unterschiede voneinander unabhängig sind und somit die Charakteristika von Anpassungsstelle aufweisen.

- Im *Anwendungsbereich* bestehen die Unterschiede auf der Ebene der Arbeitsmittel ausschließlich im Aufbau der Notizzettel. Da die Notizbestandteile Bausteine darstellen, aus denen sowohl die Notizen als auch die Bedienschnittstellen der mit Notizen arbeitenden Werkzeuge beliebig aufgebaut werden können, habe ich gezeigt, dass die Anpassungsstelle für die Analyseart, zu der diese Notizbestandteile passen, eine Mehrfachanpassungsstelle ist. Zwar besteht bei der manuell durchgeführten Analyse nach der Set Theory das zusätzliche Arbeitsmittel „Set-Class-Tabelle“, doch habe ich gezeigt, dass eine solche Tabelle im Rahmen einer rechnergestützten Analyse nicht mehr als eigenständiges Material erforderlich ist und somit keine weitere durch die Analyseart bedingte Anpassungsstelle erforderlich ist.

- Die Einflüsse aus dem Kontext *Handhabung & Präsentation* unterscheiden sich hinsichtlich der Notationsart der verwendeten Partitur. Da diese Unterschiede in der Notation unabhängig von der verwendeten Kompositionstechnik – und damit

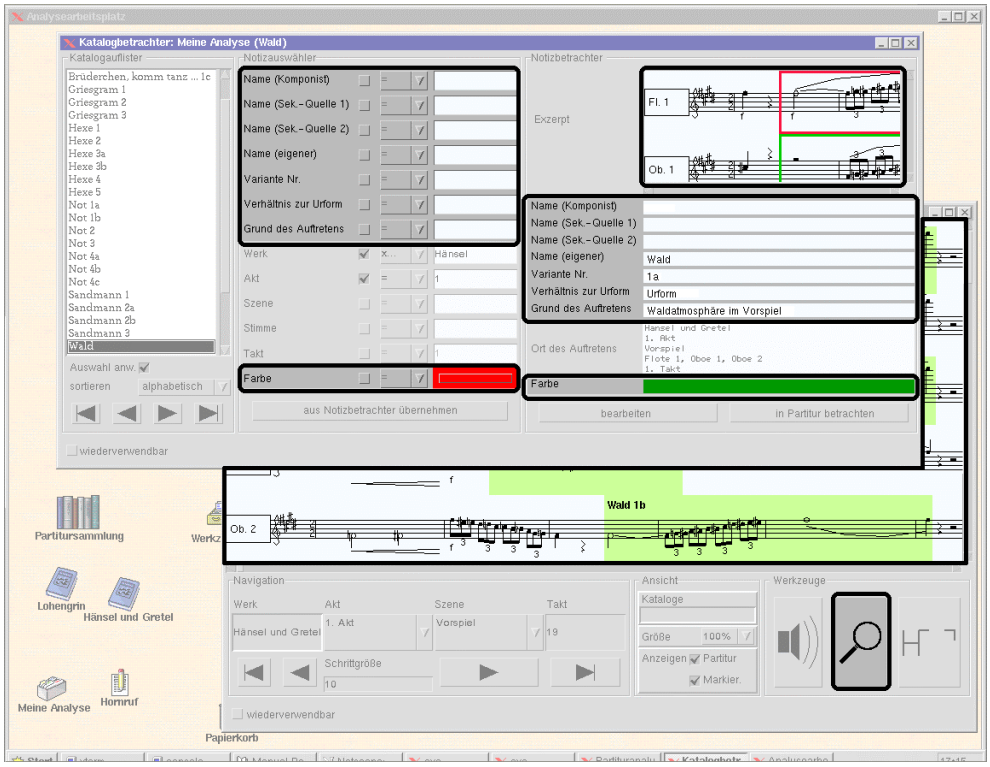


Abb. 3.5-1 Die Bestandteile der Bedienschnittstelle eines Analysesystems, die sich in Abhängigkeit der Analyseart, der Partiturart und des MDV-Subsystems ändern.

auch der Analyseart – sind, ist eine separate Anpassungsstelle für eine Partitur-Engine notwendig, die für die Darstellung der Partitur zuständig ist. Aufgrund der Tatsache, dass es für eine mit Softwarewerkzeugen durchgeführte Analyse unabdingbar ist, dass die Partitur graphisch dargestellt wird, ist diese Anpassungsstelle eine Zwangsanpassungsstelle.

- Die Unterschiede im Bereich der *verwendeten Technik* ergeben sich aus ökonomischen Gründen, die völlig unabhängig von der Analyse- und der Partiturart sind. Sowohl *Humdrum* als auch *CMAP* sind auf eine UNIX-Shell angewiesen, die auf Nicht-UNIX-Plattformen mit kommerziellen Emulatoren zur Verfügung gestellt werden muss. Um Anwender nicht zu zwingen, auch diese Produkte zu erwerben bzw. andere knappe Ressourcen in Anspruch zu nehmen, müssen technische Konfigurationen möglich sein, in denen entweder (a) *Humdrum* und *CMAP*, (b) ein funktional gleichwertiges Ersatzsystem oder aber (c) gar kein MDV-Subsystem verfügbar ist. Aufgrund dieser letzten Option ist die Anpassungsstelle keine Zwangs-, sondern eine Einfachanpassungsstelle.

Des Weiteren ist die Anwendungsfamilie zur Unterstützung der musikwissenschaftlichen Analyse komplexer als die in Kapitel 2 beschriebenen Anwendungsfamilien und weist sämtliche Arten von Anpassungsstellen auf. Ein zusätzlicher Unterschied besteht in zwei besonderen Anforderungen an die Art der Anpassbarkeit:

(1) Sämtliche Einflüsse können sich von Mal zu Mal ändern, weil z.B. (a) das vorliegende leitmotivische Werk unter Gesichtspunkten der Set Theory untersucht werden soll, (b) weil ein bestimmtes Werk nicht in CMN-, sondern in Mensuralnotation vorliegt oder (c) weil der Rechner, auf dem das letzte Mal gearbeitet wurde, besetzt ist und auf einen Rechner einer anderen Plattform ausgewichen werden muss. Die Anwender müssen das Kernanalysesystem dementsprechend von Sitzung zu Sitzung anpassen können.

(2) Da Musikwissenschaftler größtenteils Laien im Umgang mit Rechnern sind, muss der technische Prozess der Anpassung so gestaltet werden, dass auch dieser Anwenderkreis in auf verständliche Weise ausführen kann.

In den verbleibenden Kapiteln dieser Arbeit muss es nun darum gehen, Ansätze zu diskutieren, mit denen es möglich ist, diese sitzungsweise auf verständliche Weise anpassbaren Anwendungsfamilien softwaretechnisch zu entwerfen, zu implementieren und auszuliefern.

In Kapitel 4 untersuche ich zu diesem Zweck zunächst die softwaretechnischen Eigenschaften, die Kernsysteme und dazugehörige Komponenten aufweisen müssen, um durch die Anwender auf für sie verständliche Weise von Sitzung zu Sitzung anpassbar zu sein, und stelle dar, inwiefern die in der Literatur diskutierten Rahmenwerk- und Komponenten-Ansätze geeignet sind, um Anwendungsfamilien dementsprechend zu modellieren. In Kapitel 5 stelle ich den von mir entwickelten Einschub-Ansatz vor, mit dem die zuvor herausgearbeiteten Probleme der herkömmlichen Ansätze überwunden und Anwendungsfamilien bruchlos über den gesamten Entwicklungsprozess bis hin zu den Auslieferungseinheiten modelliert werden können.

Schließlich wende ich mich in Kapitel 6 dem Problem der Einbindung bestehender MDV-Systeme zu und entwickle zwei mit dem Einschub-Ansatz kombinierbare Konstruktionsansätze, mit denen es möglich ist, die komplexe, bereits in existierenden MDV-Systemen vorhandene Funktionalität maximal zu nutzen und zugleich in einem solchen Ausmaß von ihr unabhängig zu sein, dass auch ohne MDV-Systeme noch sinnvoll gearbeitet werden kann.

4 Ansätze zur Modellierung von Anwendungsfamilien

Der Übergang von einem rein fachlichen Modell des Anwendungsbereichs zu einem softwaretechnischen Modell des Anwendungssystems ist ein zentraler Schritt im Softwareentwicklungsprozess und bildet den thematischen Schwerpunkt der Literatur zum objektorientierten Entwurf. Auf der Ebene der Mikroelemente der Klassen und Operationen wird dabei übereinstimmend empfohlen, die fachlichen Gegenstände und ihre Umgangsformen in strukturähnliche Klassen und deren Operationen zu überführen sowie Begriffshierarchien als Klassenhierarchien nachzubilden (vgl. [Bla99a, S. 530], [WAM98, S. 22], [Bäu98, S. 10] und [Mey97, S. 117]).

fachliches Modell	**softwaretechnisches Modell**
Gegenstand	Objekt
Umgangsform	Operation
Begriff	Klasse
Generalisierung, Spezialisierung	Vererbung
Komposition	Aggregation, Assoziation
Begriffshierarchie	Klassenhierarchie

Diese Mikroelemente reichen aber nicht aus, um die Klassen und Klassenhierarchien großer Anwendungssysteme auf verständliche Weise zu modellieren. Daher werden seit einiger Zeit Entwurfsmuster und Modellarchitekturen als weitere Gestaltungsmittel diskutiert, in denen Erfahrungen aus anderen Projekten festgehalten und wiederverwendet können (vgl. [Szy98, S. 132 und S. 273], [Bäu98, S. 27] sowie [WAM98, S. 157]):

- *Entwurfsmuster* haben noch einzelne Klassen zum Gegenstand. Unabhängig von der oft vorgenommenen Unterteilung in Erzeugungs-, Strukturierungs- und Verhaltensmuster (vgl. [CS95], [GHJV98] und [CKV96]) helfen Entwurfsmuster primär dabei, Klassen zu verständlichen, maximal lose gekoppelten Einheiten zu gruppieren, um sie möglichst einfach und unabhängig voneinander modifizieren und austauschen zu können (vgl. [Pre97, S. 57ff]). Miteinander kombiniert und auf einen konkreten Gegenstand bezogen bilden sie die Grundlage für den Entwurf von Rahmenwerken (vgl. [BJ94]).

- *Modellarchitekturen* befassen sich nicht mehr mit Mikroelementen, sondern mit abstrakten Ordnungsprinzipien für Makroelemente wie Klassenbibliotheken und Rahmenwerken (vgl. [Bäu98, S. 87ff], [WAM98, S. 344f]). Häufig diskutiert werden dabei das Geheimnisprinzip (vgl. [Par72]), lose Kopplung und starke innere Kohäsion von Modulen (vgl. [CY79]), das Offen-Geschlossen-Prinzip (vgl. [Mey90]) sowie Architekturschichten (vgl. [GS96]).

Um festzustellen, inwiefern diese herkömmliche Art der softwaretechnischen Modellierung von Anwendungssystemen auf Anwendungsfamilien übertragbar ist, ist es notwen-

dig, die Eigenschaften von Anwendungsfamilien in Erinnerung zu rufen, die ich in Kapitel 2 herausgearbeitet habe:

(1) Anwendungsfamilien lassen sich als eine Menge von Anwendungssystemen eines Herstellers auffassen, die verschiedene Ausführungen ein und derselben chronologischen Version einer Anwendung darstellen (siehe Begriff 2.1 auf S. 13).

(2) Das die Gemeinsamkeiten verkörpernde Kernsystem kann mit austauschbaren Komponenten angepasst werden, die entweder Dienstleister oder Systembausteine sind (siehe Begriff 2.2 auf S. 17).

Da sich einzelne Mitglieder einer Anwendungsfamilie zur Laufzeit nicht von herkömmlichen, separat entwickelten Anwendungssystemen unterscheiden lassen, ergeben sich aus den unter Punkt (1) genannten Eigenschaften keine besonderen Anforderungen an die softwaretechnische Modellierung von Anwendungsfamilien. Die für Anwendungssysteme entwickelten, in der Literatur vorgeschlagenen Verfahren zur Strukturierung auf der Mikro- und Makroebene können daher unverändert auch auf Anwendungsfamilien angewendet werden. Das besondere Merkmal von Anwendungsfamilien sind lediglich die unter Punkt (2) genannten austauschbaren Dienstleister und Bausteine. Sie lassen sich wie folgt mit Entwurfsmustern im softwaretechnischen Modell repräsentieren:

- **Modellierung austauschbarer Dienstleister.** Wie bereits oben erwähnt, beinhalten die meisten Entwurfsmuster Lösungen für Situationen, in denen das Verhältnis zwischen Klassen so gestaltet werden soll, dass ein Bestandteil möglichst einfach ausgetauscht werden kann. Pree, der mit seinem Hot-Spot-Ansatz als einziger Autor die softwaretechnische Modellierung von Anwendungsfamilien im Sinne dieser Arbeit diskutiert (siehe S. 12), empfiehlt austauschbare Dienstleister nach dem Strategie-Muster zu realisieren (vgl. [Pre97, S. 42ff] und [GHJV98, S. 333ff]).

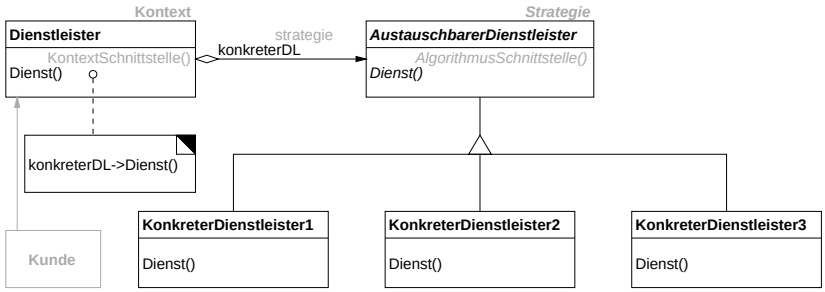


Abb. 4-1 Mit Hilfe des Strategie-Musters modellierter austauschbarer Dienstleister.

Die Bezeichnungen der beteiligten Klassen und Operationen gemäß [GHJV98] sind in grau angegeben. Um das Strategie-Muster besser mit anderen Modellierungsoptionen vergleichen zu können, greifen Kunden im Gegensatz zum reinen Strategie-Muster nicht direkt auf den austauschbaren Dienstleister zu, sondern bedienen sich eines stets gleichen Dienstleisters innerhalb des Kernsystems, der alleine den austauschbaren Dienstleister kennt und auf ihn zugreift.

Das Strategie-Muster ist ebenso wie der Hot-Spot-Ansatz auf den Austausch einer einzigen Operation – `Dienst()`– beschränkt (siehe S. 21). Um aber Anwendungsfamilien mit so komplexen Dienstleistern zu modellieren, wie ich sie in den Kapiteln 2 und 3 vorgestellt habe, reicht eine einzige Operation im Allgemeinen nicht

aus. Daher empfiehlt es sich, das in seiner Struktur gleiche, aber auf den Austausch kompletter Klassen abzielende Brücken-Muster zu verwenden (vgl. [GHJV98, S. 165ff]).

Das Brücken-Muster ist vollständig genug, um einen minimalen austauschbaren Dienstleister im softwaretechnischen Modell einer Anwendungsfamilie zu gestalten. Für einen realistischen Dienstleister in einem großen Anwendungssystem sollte das Modell allerdings um die folgenden drei Elemente erweitert werden:

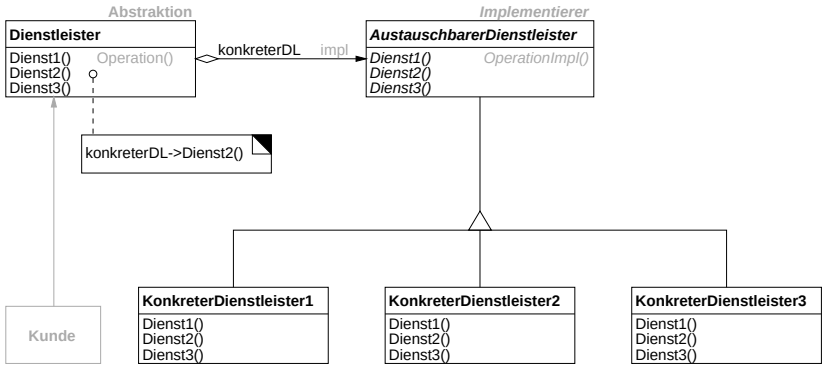


Abb. 4-2 Mit Hilfe des Brücken-Musters modellierter austauschbarer Dienstleister.

Die Klassenstruktur ist mit der des in Abb. 4-1 gezeigten Strategie-Musters identisch. Der einzige Unterschied besteht darin, dass der austauschbare Dienstleister nicht nur über eine einzige, sondern beliebig viele Operationen verfügen kann.

1. *Singletons.* In den Kapiteln 2 und 3 habe ich gezeigt, dass austauschbare Dienstleister oft von mehreren Systemteilen aus nutzbare, z.T. essentielle Dienste erbringen, die nur einmal im System vorhanden sind. Beispielsweise wird in einer Anwendung zur Nutzung des Internets die HTML-Engine sowohl vom Web-Browser als auch vom Mail-Client genutzt und im musikwissenschaftlichen Analysesystem greifen zahlreiche Werkzeuge auf die Partitur-Engine zurück. Die Lösung für diese Entwurfsanforderungen ist das Singleton-Muster, das gewährleistet, dass nur genau ein Exemplar einer Klasse in einem System vorhanden ist und dass dieses von überall aus gleichermaßen zugreifbar ist (vgl. [GHJV98, S. 139ff]).

2. *Nutzung interner Dienstleister.* Ebenfalls in Kapitel 2 habe ich dargelegt, dass Anwendungssysteme interne, nicht austauschbare Dienstleister besitzen können (siehe Begriff 2.3 auf S. 20), die ebenso wie austauschbare Dienstleister anderen Systemteilen zur Verfügung stehen, um ähnliche Funktionalität wie z.B. die Umrechnung von Zentimetern in Pixeln in einem Bildbearbeitungssystem nicht redundant an mehreren Stellen realisieren zu müssen. In allen diskutierten Beispielsystemen ist zudem ein zentraler interner Dienstleister denkbar, der sämtliche Zeichenoperationen realisiert und der von den einzelnen Systembestandteilen mit einer Liste abstrakter Zeichenbefehle aufgerufen wird. Um austauschbare Dienstleister möglichst schlank zu realisieren, ist es vorteilhaft, ihnen Zugriff auf die internen Dienstleister in Form einer Benutzt-Beziehung zu gewähren (vgl. [WAM98, S. 40]).

3. *Parameter-Klassen.* Austauschbare Dienstleister wie HTML-Engines und MDV-Subsysteme bieten potentiell komplexe Dienste an. Nur in seltenen Fällen werden Standarddatentypen wie Ganz- und Fließkommazahlen sowie Zeichenketten ausreichen, um die Operationen an deren Schnittstellen aufzurufen. Meistens werden dienstleisterspezifische Parameterklassen benötigt, die dementsprechend im softwaretechnischen Modell berücksichtigt werden müssen.

Aus diesen drei Erweiterungen des Brücken-Musters ergibt sich die folgende Struktur von Klassen, um austauschbare Dienstleister einer Anwendungsfamilie in allgemeiner Form zu gestalten.

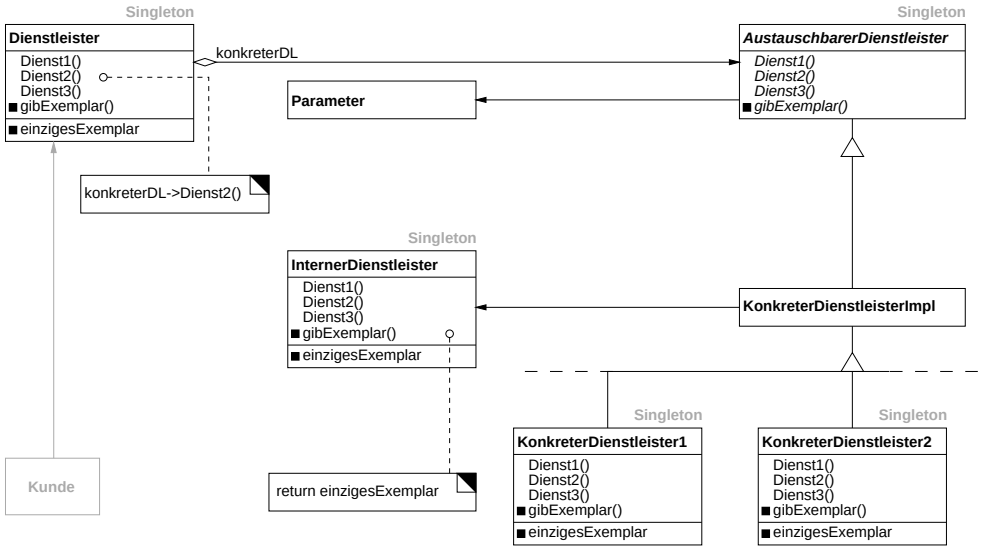


Abb. 4-3 Die Struktur austauschbarer Dienstleister im softwaretechnischen Modell einer Anwendungsfamilie.

Die Grundstruktur der Klassen entspricht derjenigen in Abb. 4-2. Der Übersichtlichkeit halber sind hier lediglich zwei konkrete Dienstleister dargestellt. Zusätzlich zu der in [GHJV98] eingeführten Notation sind Klassenoperationen und Klassenvariablen mit einem n gekennzeichnet.

- **Modellierung austauschbarer Bausteine.** Im Rahmen der Diskussion um die Eigenschaften von Bausteinkomponenten in Kapitel 2 habe ich herausgearbeitet, dass diese statt eines einzigen Bausteins einen Bauplan für eine bestimmte Art von Baustein verkörpern, der von einer geeigneten Fabrik verwendet werden kann, um beliebig viele gleichartige Bausteine zu erzeugen (siehe S. 19). Beispiele sind mehrfach vorhandene NAND-Elemente im Simulations-Beispiel (siehe Abb. 2-3 auf S. 18) oder mehrere Textbestandteile in Notizzetteln im musikwissenschaftlichen Analysesystem (siehe S. 78).

  Für diese Konstruktionsanforderungen wird in der Literatur das Fabrikmethoden-Muster empfohlen (vgl. [GHJV98, S. 115ff]). Da Bausteine einer Art – wie eben dargelegt – häufig mehrfach verwendet werden, wird das Fabrik-Muster oft mit dem Fliegengewicht-Muster kombiniert, so dass die Erzeugung zusätzlicher Bausteinexemplare nur ein Minimum an zusätzlichem Speicherplatz in Anspruch nimmt (vgl. [GHJV98, S. 199ff]).

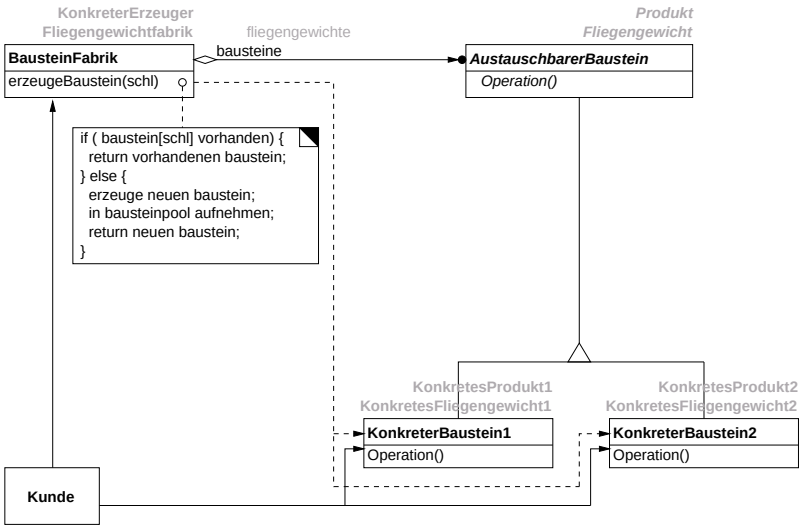


Abb. 4-4 Als Kombination aus dem Fabrikmethoden-Muster und dem Fliegengewicht-Muster modellierte austauschbare Bausteine.

Für große Anwendungssysteme empfiehlt es sich, diese minimale Fliegengewichtfabrik um genau die drei Elemente zu erweitern, die sich zuvor auch für austauschbare Dienstleister als sinnvoll erwiesen haben: (1) Eine Singleton-Operation für den Dienstleister – in diesem Fall die Fabrik –, (2) die Nutzung interner Dienstleister für leichtgewichtige Bausteine und (3) Parameter-Klassen zu deren Nutzung. Zusammen mit der zugrundeliegenden Fabrik ergibt sich damit die in Abb. 4-5 dargestellte Klassenstruktur, um austauschbare Bausteine in allgemeiner Form im softwaretechnischen Modell zu gestalten.

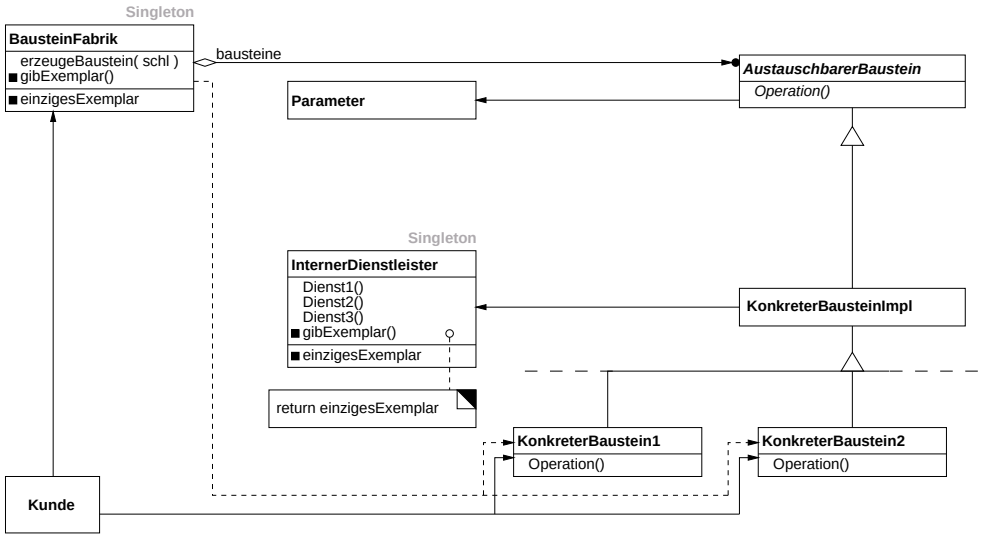


Abb. 4-5 Die Struktur austauschbarer Bausteine im softwaretechnischen Modell einer Anwendungsfamilie.

Mit den bisher diskutierten Gestaltungsmitteln ist es möglich, die austauschbaren Komponenten von Anwendungsfamilien als Klassen und Klassenhierarchien auszudrücken. Da das softwaretechnische Modell das Bindeglied zwischen dem fachlichen Modell und den Auslieferungseinheiten darstellt (siehe Abb. 1-1 auf S. 6), ist diese Abbildung des fachlichen Modells auf Klassen und Klassenhierarchien jedoch nur ein Aspekt im Rahmen der Gestaltung des softwaretechnischen Modells. Der zweite Aspekt besteht darin,

das Modell so in Auslieferungseinheiten zu partitionieren, dass – wie in Kapitel 3 gefordert – die Anwender in die Lage versetzt werden, das zukünftige Anwendungssystem sitzungsweise auf für sie verständliche Weise an ihre Anforderungen anzupassen. Diesen zweiten Aspekt diskutiere ich im Rest dieses Kapitels. Anwendungsfamilien, deren Mitglieder bereits durch die Entwickler zusammengestellt und als herkömmliche, durch die Anwender nicht weiter anpassbare Anwendungssysteme ausgeliefert werden (vgl. [CW98], [CHW98] und [GJKW98]), werde ich im Folgenden nicht weiter betrachten.

Für die Zerlegung von softwaretechnischen Modellen werden in der Literatur unterschiedliche Konzepte vorgeschlagen, die sich grob danach unterteilen lassen, ob die Struktur der Auslieferungseinheiten sich eher (1) wie ein Rahmwerk an den Gemeinsamkeiten, (2) wie Komponenten an den Unterschiedlichkeiten oder (3) an einer Mischform aus Gemeinsamkeiten und Unterschiedlichkeiten orientieren soll. In den Abschnitten 4.1 bis 4.3 prüfe ich, inwiefern sich diese Konzepte auf Anwendungsfamilien übertragen lassen, deren Mitglieder sitzungsweise auf verständliche Weise durch ihre Anwender zusammenstellbar sein sollen. Dabei untersuche ich in Abschnitt 4.1 die Option einer Auslieferung nur als Rahmenwerk, in Abschnitt 4.2 die Option einer Auslieferung nur als Komponenten sowie in Abschnitt 4.3 die Option einer gemischten Auslieferung, bevor ich die Ergebnisse in Abschnitt 4.4 zusammenfasse.

4.1 Auslieferung als reines Rahmenwerk

Ein Rahmenwerk stellt eine softwaretechnische Lösung für eine Reihe ähnlicher Probleme zur Verfügung, die den Kontrollfluss zur Laufzeit eines Softwaresystems festlegt (siehe Begriff 1.1 auf S. 6). Da Anwendungsfamilien aus einander ähnlichen Mitgliedern bestehen (siehe Begriff 2.1 auf S. 13), sind Rahmenwerke gemäß dieser Definition dazu geeignet, um Anwendungsfamilien zu realisieren. Das in Begriff 1.1 festgehaltene Rahmenwerk-Konzept ist jedoch zu abstrakt, um darüber entscheiden zu können, ob Rahmenwerke auch generell zur Auslieferung von Anwendungsfamilien in Frage kommen. Insbesondere lässt es offen, (1) ob die im Rahmenwerk verkörperte Lösung den Entwicklern oder aber den Anwendern zur Verfügung gestellt wird und (2) auf welche Weise diese Lösung nutzbar ist. Angesichts der Forderung, dass die Anwender die Mitglieder von Anwendungsfamilien selbständig sitzungsweise auf für sie verständliche Weise zusammenstellen sollen, kommen daher nur bestimmte Rahmenwerk-Unterarten für die Auslieferung in Betracht.

Zu Beginn dieses Abschnitts untersuche ich zunächst die Eigenschaften der in der Literatur diskutierten Rahmenwerk-Unterarten und arbeite darauf aufbauend deren individuelle Eignung bzw. Vor- und Nachteile für die Auslieferung von Anwendungsfamilien heraus.

Die primär diskutierten Rahmenwerk-Begriffe sind die des technischen und des Anwendungsrahmenwerks sowie des Black-Box- und des White-Box-Rahmenwerks:

- *Technische Rahmenwerke* (vgl. [WAM98, S. 119]) verkörpern eine Lösung eines funktional abgrenzbaren softwaretechnischen Problems, das sich aus jedem der drei beeinflussenden Kontexte *Anwendungsbereich*, *Handhabung & Präsentation* und *verwendete Technik* ergeben kann. Sie werden als „technisch“ bezeichnet, weil sie ausschließlich von Entwicklern genutzt werden können, um daraus komplette Anwendungen zu erstellen. Dementgegen enthalten

- *Anwendungsrahmenwerke* (vgl. [FZ99, S. 786] und [WAM98, S. 119]) den Kontrollfluss für eine kompletten Anwendung, die aber in den meisten Fällen noch ergänzt werden muss, um lauffähig zu sein. Da sie konstruktiv so gut wie immer aus mehreren technischen Rahmenwerken bestehen, sind sie wesentlich komplexer als diese.

  Vereinzelt werden auch technische Rahmenwerke, die aus dem Kontext *Anwendungsbereich* motivierte Funktionalität enthalten oder lediglich für die Erstellung von Anwendungsrahmenwerke hilfreiche Sammlungen von technischen Rahmenwerken umfassen, als Anwendungsrahmenwerke bezeichnet (vgl.[GJJ+00], [Goo98], [Ack94] und [Fir93]). Diese Interpretation kommt in dieser Arbeit nicht zum Tragen.

- Im Gegensatz zur weitgehend einheitlichen Verwendung der beiden vorangegangenen Rahmenwerkbegriffe werden die Begriffe *Black-Box-* und *White-Box-Rahmenwerk* von zahlreichen Autoren mit verschiedenen Konzepten in Verbindung gebracht, die sich zum einen auf die Sichtbarkeit der Implementation und zum anderen auf die Verwendung des Rahmenwerks beziehen:

  (1) *Black Box / White Box als Ausdruck der Sichtbarkeit der Implementation* (vgl. [GHJV98, S. 22], [Szy98, S. 33] und [Gri98, S. 18 und S. 85]). Ein Rahmenwerk wird als eine Black Box angesehen, wenn seine Implementation vor dessen Verwendern verborgen ist. Es wird als binäre Einheit ohne Quellcode ausgeliefert und ist nur über seine Schnittstelle nutzbar. Anwender und Entwickler können das Black-Box-Rahmenwerk dementsprechend ausschließlich parametrisierend verwenden, um z.B. eine bestimmte Dienstleistervariante auszuwählen oder die Bausteine zu spezifizieren, die an der Bedienschnittstelle zur Verfügung gestellt werden sollen. Auch zur Laufzeit ist es nicht möglich, das Innere einer Black Box von anderen Teilen des Anwendungssystems aus mit introspektiven Diensten[1] zu erkunden oder zu manipulieren. Ein Black-Box-Rahmenwerk ist daher zu jedem Zeitpunkt vor unbeabsichtigten oder bösartigen Veränderungen geschützt (vgl. [Szy98, S. 88]).

  Die Implementation einer White Box liegt dagegen offen und ist daher sowohl während der Entwicklung als auch zur Laufzeit jederzeit einseh- und veränderbar.

[1] Die Nutzung introspektiver Dienste wird auch Reflexion (vgl. [Szy98, S. 160]) oder Metaprogrammierung (vgl. [Fuc00, S. 470]) genannt. Bei geringen inhaltlichen Abweichungen ist es mit Technologien, die eines dieser Konzepte umsetzen, möglich, die Implementation eines Systems zur Laufzeit zu sondieren und z.T. auch zu verändern.

(2) *Black Box / White Box als Ausdruck der Verwendungsart* (vgl. [Mey97, S. 52][2], [Bäu98, S. 95] und [WAM98, S. 121]). Nach dieser Lesart ist ein Rahmenwerk genau dann ein Black-Box-Rahmenwerk, wenn es möglich ist, es ausschließlich per Parametrisierung seiner Schnittstelle zu nutzen. Ein White-Box-Rahmenwerk liegt dementgegen vor, wenn die Verwender unmittelbar auf die Implementation zugreifen, um die dortigen Klassen anwendungsspezifisch zu spezialisieren oder zu modifizieren.

Diese Art der Klassifikation hängt somit sowohl von den Entwicklern als auch von den Verwendern ab: Ein Rahmenwerk, das von dessen Entwicklern nicht für eine parametrisierende Verwendung vorgesehen ist, *kann* nur durch Spezialisierung genutzt werden und ist somit definitiv ein White-Box-Rahmenwerk. Andernfalls ist die Klassifikation entwicklerseitig unbestimmt und es hängt ausschließlich vom Einsatz durch die Verwender ab (parametrisierend oder spezialisierend), ob das Rahmenwerk als Black-Box- oder White-Box-Rahmenwerk gilt.

Da das Nebeneinander beider Lesarten unnötigerweise Missverständnissen Vorschub leistet und zu Formulierungen wie „verwendungsbezogene Black Box“ oder „sichtbarkeitsbezogene White Box“ zwingt (vgl. [Bäu98, S. 95ff]), trenne ich im Folgenden beide Konzepte auch begrifflich.

„Black Box“ und „White Box“ verwende ich von hier ab ausschließlich im Sinne von Interpretation (1) – also dann, wenn eine Aussage über die Sichtbarkeit der Implementation getroffen werden soll. Diese Verwendung deckt sich mit der Etymologie der Begriffe und korrespondiert auch mit der Verwendung in anderen Bereichen der Informatik – beispielsweise beim Testen (vgl. [FZ99, S. 783]) – und in der Alltagssprache. Für die in der Rahmenwerkdiskussion und somit auch diese Arbeit relevantere Verwendung im Sinne von Interpretation (2) führe ich die Begriffe des *Parametrisierungs-* und des *Spezialisierungsrahmenwerks* ein, die in zutreffender Weise nicht auf die Sichtbarkeit der Implementation, sondern ausschließlich auf die Verwendung abzielen.

Begriff 4.1 Parametrisierungsrahmenwerk *(parameterization framework)*

Ein White-Box- oder Black-Box-Rahmenwerk, das durch Parametrisierung verwendet werden kann, ist ein Parametrisierungsrahmenwerk.

Begriff 4.2 Spezialisierungsrahmenwerk *(specialization framework)*

Ein White-Box-Rahmenwerk, das durch spezialisierende Unterklassenbildung verwendet werden kann, ist ein Spezialisierungsrahmenwerk.

Von diesen Rahmenwerk-Arten kommen technische und Spezialisierungsrahmenwerke nicht in Frage, um Anwendungsfamilien auszuliefern, da sie nur während der Entwicklung zum Einsatz kommen können. Als Auslieferungsoption für durch die Anwender auf verständliche Weise anpassbare Anwendungsfamilien verbleiben somit lediglich Parametrisierungsanwendungsrahmenwerke. Damit die Anwender auf für sie verständli-

[2] Während alle anderen genannten Autoren, die Black-Box- und White-Box-Rahmenwerke definieren, lediglich ihre eigene Interpretation vorstellen, stellt Meyer beide einander gegenüber und lehnt dabei Interpretation (1) explizit ab.

che Weise aus den vorgegebenen Anpassungsmöglichkeiten auswählen können, müssen Parametrisierungsanwendungsrahmenwerke mit speziellen Konfigurationswerkzeugen ausgeliefert werden (vgl. [Pre97, S. 71]), die ich in Kapitel 5 noch ausführlich diskutiere.

Technisch gesehen ergeben sich für Parametrisierungsanwendungsrahmenwerke zwei Auslieferungsoptionen: Entweder als statisch gebundener Monolith oder aber mit automatisch zur Laufzeit hinzugebundenen dynamischen Bibliotheken. Aus Anwendersicht spielt diese Unterscheidung jedoch keine Rolle, da die Entscheidung darüber, welche dynamischen Bibliotheken hinzugebunden werden, bereits zum Zeitpunkt der Übersetzung festgelegt wird und nicht mehr von den Anwendern beeinflusst werden kann. Im Folgenden unterscheide ich daher nicht zwischen diesen beiden verschiedenen Arten der Auslieferung und diskutiere die Eigenschaften von Parametrisierungsanwendungsrahmenwerken für den Fall einer einzigen statisch gebundenen Auslieferungseinheit. In Abschnitt 4.3 diskutiere ich die Möglichkeiten und Voraussetzungen, unter denen die Anwender selbst auswählen können, welche dynamischen Bibliotheken verwendet werden soll.

Die Entwicklung und Auslieferung von Anwendungsfamilien als reine Parametrisierungsanwendungsrahmenwerke bringt die folgenden *Vorteile* mit sich:

- Dadurch, dass die Anwendungsfamilien wie ein herkömmliches Anwendungssystem entwickelt werden, können die umfangreichen Ergebnisse der Softwaretechnik genutzt und Strukturbrüche vermieden werden. Zudem kann die Anwendungsfamilie komplett in allen Konfigurationen vom Hersteller getestet werden.

- Die Anwender erhalten die Anwendungsfamilie wie ein reguläres Anwendungssystem als eine binäre Einheit ausgeliefert, das nicht erst kompliziert und fehlerträchtig aus getrennten Komponenten zusammengestellt werden muss. Statt dessen können die gewünschten, bereits im System vorhandenen Dienstleister und Komponenten in einer speziellen Konfigurationsumgebung auf eine solche Weise ausgewählt werden, dass technische Fehler der Anwender bei der Konfiguration ausgeschlossen werden können.

- Da Parametrisierungsanwendungsrahmenwerke in einem Stück ausgeliefert werden, sind keine besonderen Sicherungsmaßnahmen vonnöten, um Schutz vor dynamisch hinzugebundenen Systembestandteilen zu gewähren.

Die Entscheidung, eine Anwendungsfamilie als ein einziges Parametrisierungsanwendungsrahmenwerk zu realisieren, hat aber auch *Nachteile*:

- In einem Stück ausgelieferte Anwendungsfamilien werden zu sogenannten „Monolithen“ oder „fatware“[Szy98, S. 126], da sie alle Anpassungsmöglichkeiten bereits bei der Auslieferung beinhalten müssen, auch wenn der individuelle Anwender nur einen Bruchteil davon benötigt.

  Ein als Parametrisierungsanwendungsrahmenwerk ausgeliefertes Simulationssystem muss z.B. sämtliche vorhandenen Simulationselemente für unterschiedliche Simulationsarten enthalten (elektrische Schaltkreise; Bediensysteme; Petrinetze; etc.), auch wenn ein bestimmter Anwender stets nur Bediensysteme simulieren will.

- Die meisten Anpassungsstellen sind so geartet, dass sich beliebig viele dazu passenden Komponenten denken lassen. Wie in Kapitel 3 dargelegt, ist es z.B. ein wesentlicher Bestandteil musikwissenschaftlichen Forschens, neue, aussagekräftige analytische Abstraktionen für musikalische Elemente zu finden (siehe S. 71). Ein brauchbares, anwendungsorientiertes musikwissenschaftliches Analysesystem sollte dementsprechend mit Notizbestandteilen für alle bekannten und experimentellen Abstraktionen für sämtliche Analysearten ausgeliefert werden.

  Aufgrund der unbegrenzten Anzahl von Anpassungsarten ist es aber praktisch unmöglich, dass ein Hersteller alle denkbaren Komponenten auch implementieren und in die Auslieferungseinheit aufnehmen kann. Er muss daher zwischen Anwendungsorientierung und Praktikabilität abwägen: Je mehr Komponenten er einschließt, desto interessanter ist das System für Anwender aber um so größer und unhandlicher wird die Auslieferungseinheit. Je mehr Komponenten er auslässt, desto unproblematischer gestaltet sich die Auslieferung, aber um so weniger interessant wird das Produkt für die Anwender.

- Die Anwendungsfamilie kann nur zentralisiert durch den Hersteller weiterentwickelt werden. Selbst wenn einige Anwender in der Lage wären, in der Anwendungsfamilie fehlende Komponenten selbst zu erstellen bzw. bei dritten in Auftrag zu geben, besteht keine Möglichkeit, diese auch in das monolithische Parametrisierungsanwendungsrahmenwerk einzubinden.

Die Auslieferung einer Anwendungsfamilie als Parametrisierungsanwendungsrahmenwerk kommt daher nur dann in Frage, wenn die Anwendungsfamilie nur wenige Anpassungsoptionen enthält, die sich nicht häufig ändern. Von den in Kapitel 2 und 3 diskutierten Beispielen erfüllt allerdings keines diese Voraussetzungen: Bildbearbeitungssysteme müssen laufend um weitere Import- und Export-Komponenten für neuartige Graphikformate erweitert werden. Web-Browser benötigen neue Komponenten zur Darstellung von hinzugekommenen Audio-, Video- und Textdokumenttypen. Und in der Musikwissenschaft werden kontinuierlich neue Abstraktionen vorgeschlagen, die als Notizbestandteil in Frage kommen.

Für komplexe, durch die Anwender beliebig anpassbare Anwendungsfamilien müssen andere Auslieferungsarten gefunden werden, die primär die Beschränkung der Anpassungsoptionen auf die bereits fest eingebundenen Komponenten vermeiden und wenn möglich auch nicht zur Bildung von „fatware“ führen. Genau auf diese gewünschten Eigenschaften zielt der bereits in den späten sechziger Jahren geforderte (vgl. [McI68]) und nun seit einiger Zeit durch konkrete Technologien untermauerte Komponenten-Ansatz. Anwendungen sollen dabei aus eigenen und von dritten hergestellten Komponenten maßgeschneidert und ohne überflüssigen Ballast flexibel zusammengestellt werden:

> „Building new solutions by combining bought and made components improves quality and supports rapid development, leading to a shorter time to market. At the same time, nimble adaptation to changing requirements can be achieved by investing only in key changes of a component-based solution, rather than undertaking a major release change.“[Szy98, S. xiii]

Wird der Komponenten-Ansatz in seiner reinsten Form realisiert, dann bezieht ein Anwendungssystem seine gesamte Funktionalität aus ungefähr gleich großen, möglichst kontextfreien und daher in mehreren Projekten einsetzbaren Komponenten (vgl. [Szy98, S. 11]). Lediglich diejenigen Komponenten, die für die Marktführerschaft relevant sind, werden dabei noch herstellerintern entwickelt, während die restliche Funktionalität kostengünstig aus erworbenen Komponenten anderer Hersteller stammt (vgl. [Szy98, S. 5f]). Um ein konkretes Anwendungssystem zusammenzustellen, müssen die einzelnen Komponenten nur noch miteinander verbunden und dabei der anwendungsspezifische Kontrollfluss festgelegt werden. In Anlehnung an Modellbausätze, bei denen Gegenstände aus vorgefertigten Einzelteilen zusammengeklebt werden, wird dieser Teil des Anwendungssystems oft „Klebstoff“ genannt (engl. *glue*; vgl. [Szy98, S. 146]).

In den folgenden beiden Abschnitten 4.2 und 4.3 untersuche ich, inwiefern Komponenten geeignet sind, um die Probleme zu vermeiden, die sich bei der Auslieferung einer Anwendungsfamilie als ein monolithisches Rahmenwerk ergeben. Während ich in Abschnitt 4.3 eine gemischte Form der Auslieferung als Rahmenwerk mit dazu passenden Komponenten diskutiere, wende ich in Abschnitt 4.2 den Komponenten-Ansatz zunächst in seiner oben beschriebenen Reinform an. Das bedeutet:

(1) Auch das die Ähnlichkeiten verkörpernde Kernsystem mit seinen zahlreichen internen Dienstleistern und Bausteinen wird in mehrere ungefähr gleich große Komponenten aufgeteilt, die unabhängig voneinander erstellt bzw. von getrennten Herstellern bezogen werden können.

(2) Da die Komponenten des Kernsystems möglichst kontextfrei gestaltet sind und auch aus anderen Projekten stammen können, wird der anwendungsspezifische Kontrollfluss jedes Mitglieds einer Anwendungsfamilie ausschließlich im „glue code“ festgelegt.

Bei bisherigen Versuchen, im Sinne des Ideals unabhängig voneinander entwickelte Komponenten miteinander zu kombinieren, hat sich gezeigt, dass sich aufgrund der zahlreichen Möglichkeiten, Komponenten zuzuschneiden,[3] funktionale Überlappungen der Komponenten innerhalb eines Systems praktisch nicht vermeiden lassen und somit aufwendiger „glue code“ notwendig ist, um diese Konflikte aufzulösen (vgl. [Szy98, S. 139] und [Szy98, S. 280])[4]. Da sich noch nicht abzeichnet, ob diese Probleme lediglich die aktuell verfügbaren Umsetzungen des Komponenten-Ansatzes betreffen oder aber auf prinzipielle Schwierigkeiten des Ansatzes hindeuten, gehe ich im folgenden Abschnitt 4.2 von dem günstigen Idealfall aus, dass alle Komponenten überlappungsfrei zueinander passen und sich mit relativ wenig „glue code“ problemlos zu einem konkreten Mitglied einer Anwendungsfamilie kombinieren lassen.

[3] Siehe Diskussion auf S. 101.

[4] Siehe auch die Diskussion um die Probleme des auf bestehende Komponenten ausgerichteten Konzepts der Erweiterungsdimensionen auf Seite 21.

4.2 Auslieferung nur als Komponenten

Wie ich im vorangegangenen Abschnitt zeigen konnte, ist die Begriffsbildung im Bereich der Rahmenwerke bis auf wenige Ausnahmen weitgehend abgeschlossen. Für Komponenten ist dies nicht der Fall. Seit Oscar Nierstrasz und Dennis Tsichritzis die aktuelle Diskussion um den Einsatz von Komponenten begonnen haben (vgl. [GNT92] und [NT95]), werden teilweise stark voneinander abweichende Komponenten-Konzepte erörtert, die jedoch verwirrenderweise allesamt ausschließlich mit dem Begriff „Komponente“ bezeichnet werden. Um unzweideutig klar zu machen, auf welche Art von Komponenten ich mich beziehe, wenn ich im Folgenden untersuche, inwiefern Komponenten geeignet sind um Anwendungsfamilien auszuliefern, identifiziere ich zunächst die verschiedenen unter dem Begriff subsumierten Konzepte und führe eindeutige Begriffe für sie ein:

- *Komponenten als allgemeine Bestandteile von Softwaresystemen* (vgl. [ND95], [Sam97] und [Bäu98]). In dieser Interpretation werden Komponenten gemäß ihrer alltagssprachlichen Bedeutung als „Bestandteile eines Ganzen“[Dud90] angesehen und können konzeptionell unterschiedliche Teile von Softwaresystemen sein:

  - Bäumer fasst sowohl Architekturschichten als auch die wiederum darin enthaltenen technischen Rahmenwerke als Komponenten auf (vgl. [Bäu98, S. 30] und [Bäu98, S. 112]).

  - Die Definition von Nierstrasz und Dami lässt sich auf Prozeduren und Makros ebenso anwenden wie auf Klassen und Methoden (vgl. [ND95, S. 5]).

  - Das von Sametinger beschriebene Konzept geht noch über diese beiden Konzepte hinaus und umfasst zusätzlich auch rein textuelle Software-Bestandteile wie die Dokumentation (vgl. [Sam97]).

  Die Interpretationen von Komponenten als Architekturschichten oder Dokumentation sind so allgemein, dass sie sich nur unter dem alltagssprachlichen Komponentenbegriff zusammenfassen lassen. Für Rahmenwerke, Klassen, Methoden, Prozeduren, Makros und andere Konzepte, die funktionale Teile der Konstruktion eines softwaretechnischen Modells beschreiben, verwende ich im Folgenden den Begriff *Konstruktionskomponente*.

  Begriff 4.3 Konstruktionskomponente *(construction component)*

  Eine Konstruktionskomponente ist eine im Quelltext vorliegende Lösung eines softwaretechnischen Problems, das zur Wiederverwendung in einem ähnlichen Kontext geeignet ist. Sofern die einbettende Sprache die Übersetzung der Konstruktionskomponente durch einen Compiler erfordert, verliert sie innerhalb des Kompilats ihre Identität und geht nahtlos in ihm auf.

- *Komponenten als binär vorliegende Implementationsbausteine* (vgl. [FZ99, S. 786], [Ham97] und [CJJO92]). In dieser Interpretation sind Komponenten bereits übersetzte binäre Einheiten mit definierten Schnittstellen, die von den Entwicklern nutzbar sind, um die darin enthaltene Funktionalität nicht selbst implementieren zu müssen.

Beispiele für diese Art von Komponenten sind statisch ins Kompilat eingebundene Prozedur- und Klassenbibliotheken sowie *JavaBeans*[Ham97]. Von hier ab verwende ich den Begriff *Implementationskomponente*, um dieses Komponentenkonzept zu identifizieren.

Begriff 4.4 Implementationskomponente *(implementation component)*

Implementationskomponenten sind in Binärform vorliegende Lösungen softwaretechnischer Probleme, die unabhängig voneinander hergestellt und erworben werden können. Entwickler können während der Implementation auf sie zurückgreifen und sie statisch in das Kompilat von Anwendungssystemen einbinden.

- *Komponenten als zur Laufzeit kombinierbare Systembestandteile* (vgl. [Szy98, S. 315], [FZ99, S. 786], [OMG97a] und [MD95]). Eine dritte Gruppe von Autoren sieht in Komponenten binär vorliegende Systembestandteile, die nicht nur von den Entwicklern, sondern auch von qualifizierten Anwendern zu lauffähigen Anwendungssystemen zusammenstellbar sind. Um zur Laufzeit herausfinden zu können, welche Eigenschaften die zu kombinierenden Komponenten besitzen und sie dynamisch miteinander verbinden zu können, ist eine minimale Laufzeitumgebung vonnöten, die die dementsprechenden Operationen zur Verfügung stellt und die vor dem eigentlichen Anwendungssystem gestartet werden muss.

  Die am weitesten verbreiteten und einfachsten Komponenten dieser Art sind dynamische Bibliotheken, die durch den betriebssystemeigenen Lader unmittelbar beim Start einer Anwendung zu dieser hinzugebunden werden (vgl. [Mös99, S. 729]). Neuere Beispiele sind selbstregistrierende COM-Server und CORBAs Object-Server.[5]

  Um ein Mitglied einer Anwendungsfamilie im Sinne der in diesem Abschnitt untersuchten Interpretation des Komponenten-Ansatzes maximal aus Komponenten zusammenzusetzen, muss die Umgebung eine Scripting-Sprache zur Verfügung stellen, mit deren Hilfe die einzelnen Komponenten miteinander verbunden und der Kontrollfluss der Anwendung festgelegt werden kann (vgl. [Szy98, S. 145f]). Einige Autoren schlagen für diese Laufzeitumgebungen den Begriff *Komponentenrahmenwerk* vor (engl. *component framework*; vgl. [Szy98, S. 26], [Obe97] und [Wec97]). Da diese Umgebungen jedoch im Gegensatz zu herkömmlichen Rahmenwerken nicht aus Klassenhierarchien bestehen, die eine softwaretechnische Lösungen für einander ähnliche Probleme verkörpern (siehe Begriff 1.1 auf S. 6), sondern lediglich eine Laufzeitinfrastruktur zur Kombination von Komponenten zur Verfügung stellen (vgl. [Szy98, S. 280f)], bezeichne ich sie statt dessen, um Fehlinterpretationen vorzubeugen, als *Laufzeitkomponentenumgebungen*.

[5] Auf beide Technologien gehe ich in Abschnitt 4.3.1 noch genauer ein. Eine ausführlichere Erklärung befindet sich in den Anhängen B.1 und B.2.

Begriff 4.5 Laufzeitkomponentenumgebung *(runtime component environment)*

Eine Laufzeitkomponentenumgebung ist ein ausführbares Softwaresystem, das elementare Dienste anbietet, um passende Komponenten aufzunehmen und miteinander in Verbindung treten zu lassen.

Für die damit verbundene Art von Komponenten verwende ich den Begriff *Laufzeitkomponente*.

Begriff 4.6 Laufzeitkomponente *(runtime component)*

Laufzeitkomponenten sind in Binärform vorliegende Lösungen softwaretechnischer Probleme, die unabhängig voneinander hergestellt, erworben und eingesetzt werden können. Sie können in eine Laufzeitkomponentenumgebung eingegliedert und dort miteinander in Verbindung treten, um ein Anwendungssystem zu bilden.

Von den drei soeben identifizierten Komponentenarten stehen Konstruktions- und Implementationskomponenten ausschließlich während der Entwicklung zur Verfügung. Nachdem die Entwickler sie ausgewählt und mit „glue code“ untereinander zu einem konkreten Mitglied einer Anwendungsfamilie verbunden haben, verschmelzen sie jedoch miteinander zu einer einzigen Auslieferungseinheit. Konstruktions- und Implementationskomponenten können daher zwar bei der Konstruktion von Anwendungsfamilien eine deutliche Beschleunigung des Entwicklungsprozesses herbeiführen, aber zum Zeitpunkt der Auslieferung sind sie nicht mehr als separate Systembestandteile vorhanden, so dass sie als Auslieferungseinheiten für Anwendungsfamilien nicht in Frage kommen. Der Einsatz dieser Komponentenarten ist daher weder geeignet, um die geforderte sitzungsweise Zusammenstellung von Anwendungsfamilienmitgliedern zur Laufzeit durch die Anwender zu gewährleisten, noch um die in Abschnitt 4.1 aufgezeigten Probleme einer monolithischen Auslieferung zu überwinden.

Die einzige Komponentenart, mit der es möglich ist, eine monolithische Auslieferung zu verhindern und die damit verbundenen Beschränkungen und Schwierigkeiten zu vermeiden, sind Laufzeitkomponenten. Da ein konkretes Mitglied einer Anwendungsfamilie mit ihrer Hilfe tatsächlich von Sitzung zu Sitzung zusammengestellt werden kann, ergeben sich für sie die folgenden *Vorteile*, die das genaue Komplement der Nachteile einer reinen Auslieferung als Rahmenwerk darstellen:

- Da Mitglieder einer Anwendungsfamilie sitzungsweise zusammenstellbar sind, besteht keine Notwendigkeit mehr, „fatware“ auszuliefern, die bereits alle vorhandenen Anpassungsmöglichkeiten enthält, obwohl diese eventuell gar nicht benötigt werden. Dadurch, dass überflüssige Laufzeitkomponenten gar nicht erst eingebunden werden müssen, ist es möglich, stets Anwendungssysteme mit minimaler Größe zu erzeugen.

- Die Anwendungsfamilie kann flexibel weiterentwickelt werden, ohne dass die Anwender vom ursprünglichen Hersteller abhängen. Während die Anwender im Fall der monolithischen Auslieferung keine individuellen Erweiterungen der Anpassungsmöglichkeiten vornehmen können, so können sie Laufzeitkomponenten für

spezielle Bildbearbeitungs-Filter, Simulationselemente oder musikwissenschaftliche Abstraktionen selbst programmieren bzw. in Auftrag geben und dann sofort in das Anwendungssystem einbinden.

Die Auslieferung einer Anwendungsfamilie in zahlreichen separaten Komponenten, die erst zur Laufzeit zusammengestellt werden, hat jedoch auch *Nachteile*:

- Da die Anwendung nicht in einem Stück ausgeliefert wird, ist kein Integrationstest (vgl. [FZ99, S. 776]) möglich. Einige Programmierfehler können somit erst während der Ausführung der Anwendung entdeckt werden, wobei die bis dahin erreichten Arbeitsergebnisse unter Umständen verloren gehen.

- Sofern die einzelnen Komponenten nicht sämtliche von ihnen benötigten Funktionen aufwendig selbständig implementieren, können komplexe Abhängigkeitsgeflechte zwischen Komponenten entstehen, die den Konfigurationsprozess erheblich erschweren.

Wenn ein Anwendungssystem gemäß der in diesem Abschnitt untersuchten Reinform des Komponenten-Ansatzes ausschließlich aus Komponenten und „glue code“ bestehen soll, bringt die Auslieferung einer Anwendungsfamilie als Laufzeitkomponenten einen zusätzlichen gravierenden Nachteil mit sich:

- Da auch Scripting eine Form der Programmierung darstellt, müssen die Anwender über Programmierkenntnisse verfügen, um Laufzeitkomponenten per „glue code“ miteinander zu verbinden und den Kontrollfluss der von ihnen gewünschten Anwendung festzulegen. Das heißt, dass beispielsweise von einem Musikwissenschaftler erwartet wird, dass er selbständig in der Lage ist, einen Satz von kontextfreien Laufzeitkomponenten mit Skripten zu einem Anwendungssystem mit der von ihm gewünschten anwendungsspezifischen Funktionalität zu verbinden.

Wie ich in Abschnitt 3.2 dargelegt habe, sind Anwender zwar Experten in ihrem jeweiligen Fachgebiet, jedoch in den allermeisten Fällen nicht zugleich auch Experten im Umgang mit Rechnern. Nur sehr wenige Anwender von Bildbearbeitungsprogrammen, Web-Browsern und musikwissenschaftlichen Analysesystemen sind daher fähig, den zur Verwendung einer als Laufzeitkomponenten ausgelieferten Anwendungsfamilie benötigten „glue code“ selbst zu programmieren. Von den in Kapitel 2 eingeführten Beispielen sind die Chancen, dass die Anwender programmieren können, lediglich bei Simulationssystemen höher, da der technische Anwendungsgegenstand eine eingehendere Vertrautheit mit Rechnern erwarten lässt.

Aufgrund der Notwendigkeit, maximal aus Komponenten bestehende Anwendungssysteme mit Skripten verbinden zu müssen, sind Laufzeitkomponenten daher nur als Auslieferungseinheiten für solche Anwendungsfamilien geeignet, deren Anwender Programmierer sind oder über vergleichbare Fähigkeiten verfügen. Problemlos realisierbar wäre z.B. eine Anwendungsfamilie von Software-Entwicklungsumgebungen, die an die Erfordernisse unterschiedlicher Programmiersprachen, Compiler und Projekte angepasst werden kann. Da der größte Teil von Anwendern jedoch nicht in diese Kategorie fällt, stellt die in diesem Abschnitt diskutierte Form des Komponenten-Ansatzes nur einen in

Ausnahmefällen gangbaren Weg dar, um Anwendungsfamilien mit Dienstleistern und Bausteinen auszuliefern, die durch die Anwender austauschbar sind.

Damit kommt weder die Reinform des Rahmenwerk-Ansatzes noch die Reinform des Komponenten-Ansatzes als allgemein anwendbares Auslieferungsverfahren von durch die Anwender sitzungsweise anpassbaren Anwendungsfamilien in Frage:

- Reine *Parametrisierungsanwendungsrahmenwerke* sind zwar gut geeignet, um das Kernsystem und den Kontrollfluss von Anwendungsfamilien zu modellieren, aber durch die monolithische Auslieferung kann stets nur eine durch die Hersteller fest vorgegebene Anzahl von Anpassungsmöglichkeiten realisiert werden.

- Komplementär dazu ist es zwar möglich, mit einer ausschließlich als ungefähr gleich große *Laufzeitkomponenten* ausgelieferten Anwendungsfamilie beliebig viele Anpassungsmöglichkeiten zur Verfügung zu stellen, aber nur die wenigsten Anwender sind in der Lage, den nicht in den Komponenten enthaltenen Kontrollfluss der Anwendungen selbständig zu programmieren.

Um die in Kapitel 2 und Kapitel 3 diskutierten Anwendungsfamilien auf der Grundlage von Rahmenwerken und Komponenten sowohl flexibel als auch anwendungsorientiert ausliefern zu können, muss daher eine Kombination beider Ansätze erwogen werden. Das zu lösende Entwurfsproblem besteht hierbei darin, lediglich die Stärken der beiden Ansätze zu kombinieren aber gleichzeitig die Schwächen zu vermeiden. Das heißt,

(1) der gemischte Ansatz soll den fest vorgegebenen Kontrollfluss eines Rahmenwerk-Ansatzes mit der Flexibilität eines Komponenten-Ansatzes vereinen.

Er soll dabei aber so ausgestaltet sein, dass

(2) die gewünschten Mitglieder von Anwendungsfamilien im Gegensatz zum reinen Komponenten-Ansatz nicht nur durch des Programmierens kundige Anwender zusammengestellt werden können.

4.3 Gemischte Auslieferung als Rahmenwerk mit Komponenten

Am Beispiel dynamischer Bibliotheken habe ich bereits gezeigt, dass es für herkömmliche Anwendungssysteme eine gebräuchliche Vorgehensweise ist, einen den Kontrollfluss steuernden Teil einer Anwendung erst zur Laufzeit mit Funktionalität verkörpernden Laufzeitkomponenten zusammenzubinden (siehe S. 97). Da die zu bindenden Laufzeitkomponenten hierbei schon bei der Übersetzung festgelegt werden und der Prozess des Bindens automatisch ohne Zutun der Anwender erfolgt, können die Auslieferungseinheiten nach rein technischen Gesichtspunkten unter dem alleinigen Einfluss des Kontexts *verwendete Technik* zugeschnitten werden. Bäumer empfiehlt zum Beispiel, das softwaretechnische Modell des Anwendungssystems primär so in Auslieferungseinheiten zu partitionieren, dass nur ein minimaler Übersetzungsaufwand

entsteht und Benutzt-Beziehungen zwischen Klassen nicht getrennt werden (vgl. [Bäu98, S. 122ff]). Szyperski rät zu einer umfassenden Abwägung von dreizehn verschiedenen Faktoren, zu denen neben dem minimalen Übersetzungsaufwand unter anderem auch die bestmögliche Isolierbarkeit von Fehlerquellen sowie die optimale Wartbarkeit gehören (vgl. [Szy98, S. 123ff]). Die Struktur des ursprünglich auch unter dem Einfluss der Kontexte *Anwendungsbereich* und *Handhabung & Präsentation* gestalteten softwaretechnischen Modells bleibt dabei in den meisten Fällen nicht mehr erkennbar, so dass zwischen den Auslieferungseinheiten und den Kontexten *Anwendungsbereich* und *Handhabung & Präsentation* Strukturbrüche entstehen.

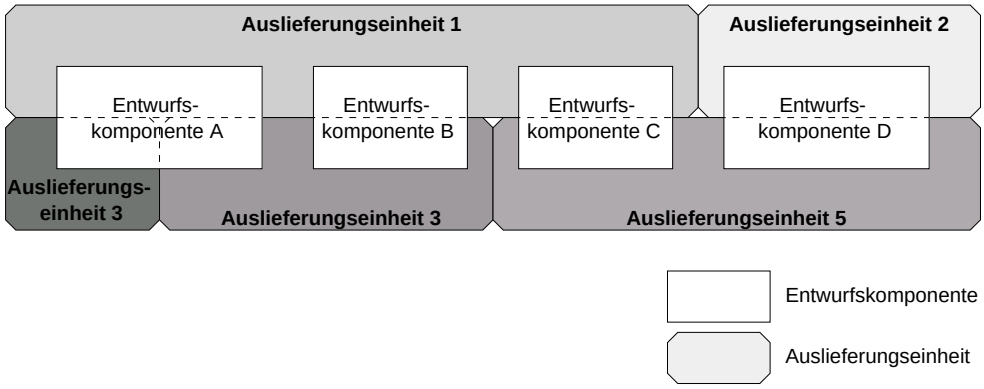


Abb. 4.3-1 Partitionierung eines softwaretechnischen Modells in Auslieferungseinheiten nach rein technischen Gesichtspunkten gemäß [Bäu98, S. 125].

Die vier Konstruktionskomponenten A bis D werden auf die fünf Auslieferungseinheiten 1 bis 5 abgebildet, die die Struktur der Konstruktionskomponenten nicht mehr erkennen lassen. So wird z.B. die Konstruktionskomponente A auf drei verschiedene Auslieferungseinheiten *aufgeteilt*, während die Auslieferungseinheit 1 Bestandteile der Konstruktionskomponenten A, B und C *vereint*.

Während diese Strukturbrüche für herkömmliche Anwendungssysteme, die automatisch gebunden werden, problemlos in Kauf genommen werden können, so können sie je nach gewähltem Leitbild zu Schwierigkeiten führen, wenn die Auslieferungseinheiten wie in Kapitel 3 gefordert durch die Anwender selbst zu einem Mitglied einer Anwendungsfamilie zusammenstellbar sein sollen:

(1) Wenn Anwendungsfamilien nach dem Leitbild des *Arbeitsplatzes für eigenverantwortliche Expertentätigkeit* (siehe S. 48) oder vergleichbaren anwendungsorientierten Leitbildern erstellt werden, dann sind rein technisch zugeschnittene Auslieferungseinheiten für die Anwender unverständlich und somit nicht handhabbar, um die gewünschten Anpassungen in Eigenregie vorzunehmen.

(2) Dementgegen führen die Strukturbrüche zu keinen Problemen, wenn Anwendungsfamilien gemäß dem Leitbild der *Objektwelten* oder ähnlichen, technisch motivierten Leitbildern erstellt werden, da dort von den Anwendern erwartet wird, dass sie sich eine ihnen vormals unbekannte technische Sichtweise zu Eigen machen (siehe S. 48).

Da ich mich in dieser Arbeit ausschließlich mit anwendungsorientierter Anwendungsentwicklung befasse, werde ich den unter (2) genannten Fall, für den keine besondere Modellierung der Auslieferungseinheiten erforderlich ist, nicht weiter betrachten, und im Folgenden ausschließlich Maßnahmen diskutieren, mit denen die aufgezeigten Struk-

turbrüche zwischen den durch die Anwender zur Anpassung verwendeten Auslieferungseinheiten und den Kontexten *Anwendungsbereich* sowie *Handhabung & Präsentation* im Interesse der Anwendungsorientierung vermieden werden können.

Die Mittel, um die Einflüsse aus diesen beiden Kontexten in einem softwaretechnischen Modell zu berücksichtigen, habe ich bereits beschrieben:

- Für eine bruchlos zum Kontext *Anwendungsbereich* passende Modellierung ist es erforderlich, dass die Strukturen des zu erstellenden Modells den Strukturen des fachlichen Modells ähneln (siehe S. 85).

- Für eine bruchlos zum Kontext *Handhabung & Präsentation* passende Modellierung ist es erforderlich, dass die Gegenstände des zu erstellenden Modells sich als Verkörperungen der zum gewählten Leitbild passenden Entwurfsmetaphern auffassen lassen (siehe S. 50).

Bisher habe ich diese Mittel ausschließlich eingesetzt, um die Mikroelemente des softwaretechnischen Modells wie Typen, Klassen und deren Operationen als Werkzeuge, Materialien, Automaten, Behälter und Arbeitsumgebungen zu gestalten. Im Folgenden zeige ich, wie sie eingesetzt werden können, um die für die Auslieferung von Anwendungsfamilien relevanten Makroelemente zu modellieren: (1) Das die Gemeinsamkeiten verkörpernde Kernsystem und (2) die die Anpassungsmöglichkeiten verkörpernden Komponenten.

- **Modellierung gemäß Einflüssen aus dem Kontext *Anwendungsbereich*.** Da in anwendungsorientierten Softwareentwicklungsprozessen der *Anwendungsbereich* der primäre Entwicklungskontext ist, muss sich der Zuschnitt der Auslieferungseinheiten nach den für die Anwender verständlichen Makroelementen Kernsystem und Komponenten richten. Anstatt die Auslieferungseinheiten daher wie in Abb. 4.3-1 gezeigt vollkommen unabhängig von diesen Elementen zuzuschneiden, muss eine direkte Korrespondenz zwischen je einem Makroelement und je einer Auslieferungseinheit etabliert werden, damit die Anwender eine Assoziation zwischen ihnen bekannten Funktionalitäten und den Auslieferungseinheiten herstellen können. Für eine aus einem Kernsystem und 11 Komponenten bestehende Anwendungsfamilie bedeutet dies, dass es nur eine einzige Auslieferungseinheit für das gesamte Kernsystem gibt und jede der 11 Komponenten genau einer Auslieferungseinheit entspricht.

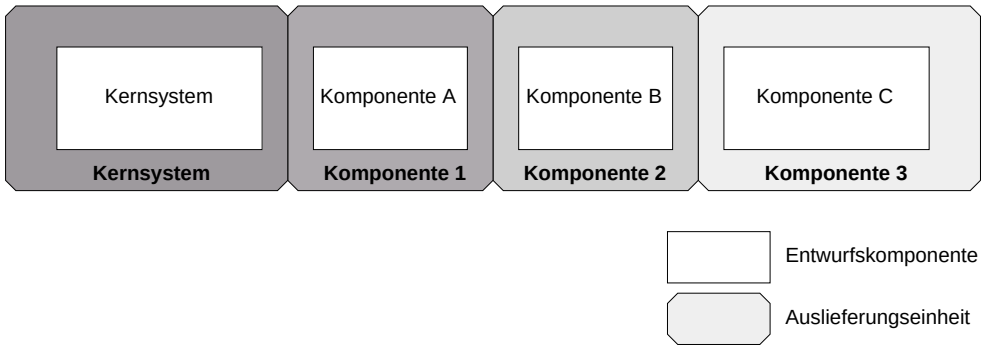


Abb. 4.3-2 Partitionierung des softwaretechnischen Modells einer Anwendungsfamilie in Auslieferungseinheiten nach anwendungsorientierten Gesichtspunkten.

Diese Dominanz der Einflüsse aus dem Kontext *Anwendungsbereich* bedeutet natürlich nicht, dass Einflüsse aus den Kontexten *Handhabung & Präsentation* und *verwendete Technik* beim Zuschnitt der Auslieferungseinheiten vollkommen außer Acht gelassen werden müssen. Es muss lediglich sichergestellt werden, dass sie einem anwendungsorientierten Zuschnitt nicht entgegenstehen. Beispielsweise können sämtliche Komponenten zwecks Verringerung ihrer Größe auf dynamische Mathematik-Bibliotheken zurückgreifen, da diese automatisch vom Betriebssystemlader hinzugebunden werden und somit nicht der Handhabung durch die Anwender bedürfen.

- **Modellierung gemäß Einflüssen aus dem Kontext *Handhabung & Präsentation*.** Damit die Anwender ein Mitglied einer Anwendungsfamilie selbständig zusammenstellen können, reicht es nicht aus, dass die Auslieferungseinheiten einer Anwendungsfamilie anwendungsorientiert zugeschnitten sind. Solange die Auslieferungseinheiten aus Anwendersicht gestaltlose technisch-binäre Einheiten sind, von denen nicht klar ist, auf welche Art und Weise sie zu einem lauffähigen System kombiniert werden können, kann die geforderte verständliche sitzungsweise Anpassung des Kernsystems durch die Anwender nicht erfolgen.

  Um den technischen Vorgang der Zusammenstellung eines Mitglieds einer Anwendungsfamilie für die Anwender verständlich zu machen, muss das in Abschnitt 3.2 vorgestellte Konzept der Entwurfsmetapher von Gegenständen der Anwendung wie Werkzeugen, Materialien und Behältern auf die Auslieferungseinheiten des Anwendungssystems ausgedehnt werden. Diese sich auf Auslieferungseinheiten beziehenden Entwurfsmetaphern, die genauso wie herkömmliche Entwurfsmetaphern zu einem bestimmten Leitbild passen müssen, nenne ich im Folgenden *Systemmetaphern*.

  Begriff 4.7 Systemmetapher *(system metaphor)*

  Eine Systemmetapher ist eine spezielle Entwurfsmetapher, die die Funktionalität und Verbindung der Bestandteile eines mehrteilig ausgelieferten Anwendungssystems vergegenständlicht.

  Ebenso wie viele Leitbilder werden Systemmetaphern in den meisten Fällen rein informal eingeführt. Entweder liegen sie der in einem Werk vertretenen Sichtweise implizit zugrunde und werden überhaupt nicht benannt (vgl. [WWW90] und [Jac92]) oder sie werden ausschließlich anhand ihres Namens eingeführt und nicht weiter erläutert (vgl. [Mey97] und [Szy98]). Zwei sehr einfache auf Auslieferungseinheiten von Anwendungssystemen angewendete Metaphern kommen aus dem Bereich des Kinderspielzeugs: *Legosteine* und *Steckbretter*.

  - *Legosteine* (vgl. [Mey97, S. 505] und [Szy98, S. 8]) werden in einer großen Vielfalt von Formen und Farben ausgeliefert und besitzen mindestens eine Stelle, an der sie mit anderen Legosteinen verbunden und auf diese Weise zu beliebigen größeren Einheiten zusammengesteckt werden können. Bezogen auf mehrteilig ausgelieferte Anwendungssysteme passt diese Metapher zu dem in Abschnitt 4.2 diskutierten reinen Komponenten-Ansatz, bei dem es allein den Anwendern obliegt, ungefähr gleich große, möglichst kontextfrei verwendbare Komponenten dergestalt auf anwendungsspezifische Weise zusammenzustellen, dass daraus ein Anwendungssystem entsteht.

- *Steckbretter* (vgl. [Mey97, S. 505]) geben dementgegen eine bestimmte Anzahl von verschieden geformten Löchern vor (z.B. in Gestalt von Tieren, Zahlen oder Buchstaben), in die jeweils nur ein bestimmter Steckstein hineinpasst. Diese Metapher korrespondiert im Gegensatz zu *Legosteinen* besser mit der in diesem Abschnitt diskutierten Auslieferungsform von Anwendungsfamilien, bei der das Kernsystem (gleich einem Steckbrett) vorgegeben ist und nur an dessen unterschiedlichen Anpassungsstellen (den verschieden geformten Löchern) mit dazu passenden Komponenten (den passenden Stecksteinen) angepasst werden kann.

Trotz ihrer oberflächlichen Anschaulichkeit sind beide Metaphern als Systemmetaphern problematisch, da sie rein passive Elemente zum Gegenstand haben, anhand derer es nicht möglich ist, die Funktionalität und die dynamische Interaktion von Bestandteilen von Anwendungssystemen verständlicher zu machen.[6] Die *Legostein*-Metapher hilft zum Beispiel wenig, um die gegenseitige Nutzung und die damit verbundene Kommunikation von Kernsystem und Komponenten zu versinnbildlichen, da Legosteine nicht miteinander kommunizieren. Ebensowenig hilft es, sich vorzustellen, dass ein *Steckstein* einen Dienst verkörpert, der durch das *Steckbrett* nutzbar ist. Es ist zwar möglich, beide Metaphern zu erweitern und z.B. *elektrische Legosteine* oder *intelligente Steckbretter* vorzuschlagen, doch solche erweiterten Metaphern, die künstliche Gegenstände zur Versinnbildlichung heranziehen, die in der Realität gar nicht existieren, verlieren genau durch diesen Realitätsverlust ihren veranschaulichenden Wert, da sich die Anwender im Gegensatz zu Metaphern wie *Werkzeug*, *Material*, *Legostein* und *Steckbrett* unter diesen fiktiven Gegenständen nichts Eindeutiges und ihnen Vertrautes vorstellen können.

Um die Handhabung von Auslieferungseinheiten einer Anwendungsfamilie zu veranschaulichen, muss eine Systemmetapher daher so beschaffen sein, dass sie die dynamischen Eigenschaften des Kernsystems und der dazu passenden Komponenten auf natürliche Weise ohne künstliche Zusätze veranschaulichen kann. Darüber hinaus muss sie alle Aspekte von Anwendungsfamilien und deren Bestandteilen in geeigneter Weise verkörpern können:

(1) Die Kontrollflusssteuerung ist im Kernsystem angesiedelt.

(2) Komponenten sind entweder als Bausteine oder als Dienstleister auffassbar.

(3) Die Verbindungen zwischen dem Kernsystem und den Komponenten sind wie folgt gestaltet:

  (3a) Sie sind nicht permanent und lassen sich sitzungsweise auch wieder lösen.

  (3b) Das Kernsystem hat voneinander unabhängige Anpassungsstellen.

  (3c) Nur genau eine Art von Komponenten passt zu einer bestimmten Art von Anpassungsstelle.

[6] Die Existenz von Legomotoren beeinflusst die Qualität der Metapher nicht, da es sich bei der Motorisierung neben Form und Farbe nur um eine weitere von vielen optionalen Ausprägungen von Legosteinen handelt. Da Metaphern aber auf den allgemeinen Eigenschaften einer Klasse von Gegenständen fußen, ergäbe sich eine Änderung der Qualität der Lego-Metapher nur dann, wenn sämtliche Legosteine motorisiert wären.

(3d) Anpassungsstellen sind entweder Einfach-, Mehrfach- oder Zwangsanpassungsstellen.

(3e) Kernsystem und Komponenten haben jeweils eine eingehende als auch eine ausgehende Schnittstelle (engl. *incoming* und *outgoing interface*; vgl. [Szy98, S. 149]).

Im Rest dieses Abschnitts untersuche ich drei der am häufigsten in der Literatur auf Auslieferungseinheiten angewandten Metaphern (Server, aktive Dokumente sowie Einsteck-Erweiterungen) und prüfe, inwiefern sie gemäß den genannten Kriterien als Systemmetaphern für Anwendungsfamilien geeignet sind. Dabei gehe ich davon aus, dass die Auslieferungseinheiten entsprechend den bisherigen Ergebnissen anwendungsorientiert zugeschnitten sind, und das Kernsystem als erweiterbares Parametrisierungsanwendungsrahmenwerk ausgeliefert wird sowie jede eine Anpassungsmöglichkeit verkörpernde Komponente genau einer Laufzeitkomponente entspricht.

Da sich alle drei Metaphern als problematisch erweisen, entwickle ich in Kapitel 5 die Systemmetapher *Einschub und Einschubrahmen*, die sämtliche der obigen Kriterien erfüllt und somit die Schwierigkeiten der hier diskutierten Metaphern vermeidet.

4.3.1 Komponenten als Server

Der Begriff des Servers geht auf das Client-Server-Prinzip aus dem Bereich der verteilten Systeme und somit auf den Kontext *verwendete Technik* zurück. In seiner einfachsten Form beschreibt es das Verhältnis zwischen zwei über eine Leitung miteinander verbundener Rechner. Der eine Rechner bietet Dienste an (Server), die der andere Rechner als Kunde in Anspruch nimmt (Client). Nachdem die Rechner miteinander verbunden worden sind, kann der Client beim Server erfragen, welche Dienste dort zur Verfügung stehen. Sobald er eine Antwort erhalten hat, kann der Client die vom Server angebotenen Dienste beliebig häufig in Anspruch nehmen. Dazu sendet er zunächst eine Anfrage an den Server, aus der die Art des gewünschten Dienstes und die zur Bearbeitung notwendigen Parameter hervorgehen. Der Server erbringt daraufhin die erbetene Dienstleistung und übermittelt sie an den Client (vgl. [Bor99, S. 665ff]). Aufbauend auf diesem minimalen Grundprinzip sind mehrere Verallgemeinerungen möglich, die üblicherweise ebenfalls mit dem Client-Server-Prinzip in Verbindung gebracht werden:

1. *Beliebig viele Clients und Server*. Die Verbindung besteht nicht nur zwischen genau einem Client und genau einem Server, sondern kann mehrere Clients und / oder Server umfassen. Das heißt, dass ein Client für verschiedene Arten von Diensten auf verschiedene Server zugreifen kann und dass umgekehrt ein Server mehr als einen Client haben kann, für den er Dienste erbringt (vgl. [FZ99, S. 786]).

2. *Flexible Rollenverteilung*. Wenn ein Rechner sowohl als Client als auch als Server fungieren kann, dann ergibt sich die Aufteilung, welcher Rechner der Client und welcher Rechner der Server ist, nicht statisch, sondern erfolgt für jeden Aufruf aufs Neue. Nachdem beispielsweise Rechner A den Dienst b_3 von Rechner B in Anspruch genommen hat, kann unmittelbar danach Rechner B von Rechner A die Dienstleistung a_2 erbitten (vgl. [Bor99, S. 665ff]).

Die Systemmetapher *Client und Server* überträgt diese Eigenschaften des Client-Server-Prinzips aus dem Kontext *verwendete Technik* in den Kontext *Handhabung & Präsentation* und versucht, die Bestandteile eines mehrteilig ausgelieferten Anwendungssystems als *Client* oder als *Server* zu veranschaulichen.

Im hier betrachten Fall einer Anwendungsfamilie ist es nicht möglich, eine feste Zuordnung anzugeben, bei der beispielsweise das Kernsystem dauerhaft dem Client entspricht und die Komponenten dauerhaft als Server fungieren. Zwar greift das Kernsystem als *Client* auf die Dienste der Komponenten zu und macht sie dadurch zu *Servern*, aber umgekehrt können auch die Komponenten auf die internen Dienstleister des Kernsystems zugreifen, wodurch die Rollenverteilung umgekehrt wird (siehe Begriff 2.3 auf S. 20 und die Diskussion des softwaretechnischen Modells zu Beginn dieses Kapitels auf S. 87).

Entsprechend ihrer technischen Natur wird die Metapher von *Client und Server* hauptsächlich im Zusammenhang mit dem Kontext *verwendete Technik* diskutiert (vgl. [FZ99], [Müh99], [Gri98], [Szy98]). Am häufigsten besteht dabei eine Verbindung zu den konkreten Technologien CORBA und COM, die das Client-Server-Prinzip zur Erreichung unterschiedlicher Ziele einsetzen (vgl. [OMG97] und [MD95]):[7]

- *CORBA* (Object Management Group, Version 1.1 1991 vorgestellt) zielt darauf ab, die Dienste objektorientierter Systeme über die Grenzen von Programmiersprachen, Implementationen, Betriebssystemen und Rechnern mit den Mitteln des Operationsfernaufrufs (*remote method invocation*, *RMI*) verfügbar zu machen. Zu diesem Zweck werden die plattformabhängigen Dienstanbieter als Server realisiert, an die sich interessierte Aufrufer in Form von Clients wenden können, um deren Dienste von einer beliebigen anderen (oder auch der gleichen) Plattform aus zu nutzen. Um die Clients und Server nur möglichst lose miteinander zu koppeln und die notwendigen Konvertierungen zur Überbrückung der plattformbedingten binären Inkompatibilitäten vorzunehmen, kommunizieren Clients und Server ausschließlich über einen zentralen *Object Request Broker* (ORB), bei dem sich alle Dienstleister als *object server* anmelden müssen.

- *COM* (Microsoft Corporation und Digital Equipment, Version 0.9 1995 vorgestellt) setzt zwar ebenso wie CORBA Clients und Server ein, doch geht es dabei nicht um die Überbrückung des Zwischenraums zwischen Rechnern, auf denen objektorientierte Systeme laufen, sondern darum, Systeme auf einem einzigen Rechner miteinander zu verbinden, die mit Hilfe unterschiedlicher – auch nicht-objektorientierter Sprachen – erstellt worden sind. Während CORBA primär ein Protokoll vorgibt, über das CORBA-Server und -Clients via ORB indirekt miteinander kommunizieren können, so definiert COM einen Binärstandard für Server, dank dessen Clients auch ohne eine vermittelnde und übersetzende Instanz direkt auf Server zugreifen können. Zur Koordinierung von Clients und Servern gibt es lediglich eine sogenannte *COM library*, mit deren Hilfe interessierte Clients die Adressen derjenigen Server

[7] In den Anhängen B.1 und B.2 erkläre ich die technischen Details von CORBA und COM ausführlich, wobei ich auch auf Erweiterung wie die OMA, COM+ und DCOM eingehe. Während diese Details zum Verständnis der Diskussion dieses Abschnitts nicht notwendig sind, so sind sie hilfreich, um die in den grauen Kästen angeführten Realisierungsbeispiele nachvollziehen zu können.

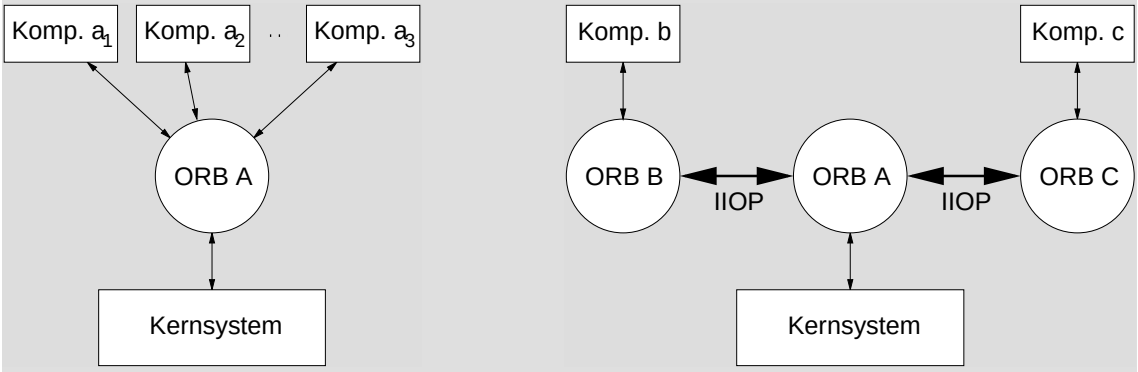
Realisierung einer Anwendungsfamilie mit CORBA

Um das Kernsystem und die Komponenten als Server zu realisieren, müssen deren Schnittstellen in der OMG IDL beschrieben werden. Da sich Schnittstellenbeschreibungen in CORBA auf Klassen beziehen, muss dementsprechend die Schnittstelle jeder Komponente durch genau eine Klasse repräsentiert werden. Um das Kernsystem als eine Einheit zu modellieren, empfiehlt es sich, die Schnittstelle des Kernsystems als eine einzige Fassade-Klasse zu gestalten (vgl. [GHJV98, S. 189]), über die alle internen Dienstleister erreichbar sind, die den Komponenten zum Zugriff dargeboten werden. Auf diese Weise liegt nach der Übersetzung genau ein *object server* für das Kernsystem und je ein *object server* für jede Komponente vor.

Die Art und Weise, wie die *object server* einzelnen ORBs zugeordnet werden, hängt davon ab, ob die verfügbare ORB-Implementation dazu in der Lage ist, *object server* zur Laufzeit einzugliedern und somit Anwendungsfamilien sitzungsweise um neue Komponenten zu erweitern.

(1) Da jeder ORB ihm bekannte *object server* dynamisch laden und starten kann, sofern dessen Dienste benötigt werden, können alle *object server* einem einzigen ORB zugeordnet werden, falls der ORB *object server* auch dynamisch eingliedern kann. Neue Komponenten können in diesem Fall auf unproblematische Weise jederzeit hinzugefügt werden. *(linke Graphik*)

(2) Falls der ORB jedoch schon beim Start eine feste, nicht mehr erweiterbare Liste der ihm zugänglichen *object server* benötigt, dann müssen die *object server* der Komponenten auf separate ORBs aufgeteilt werden, um die dynamische Erweiterbarkeit zu erreichen. Um eine neue Komponente einzugliedern, muss in diesem Fall deren „privater“ ORB gestartet werden, der die Dienste der neuen Komponente per IIOP allen anderen ORBs (und damit auch dem Kernsystem) bekannt macht. *(rechte Graphik)*



Realisierung einer Anwendungsfamilie mit COM

Die Aufteilung der Komponenten und des Kernsystems auf einzelne Server kann in COM genauso wie in CORBA erfolgen, so dass es für das gesamte Kernsystem genau einen COM-Server und für jede Komponente je einen separaten COM-Server gibt. Da COM jedoch ein reiner Binärstandard ist, ist es bei COM im Gegensatz zu CORBA nicht notwendig, die Schnittstellen des Kernsystems und der Komponenten mit einer IDL zu beschreiben. Statt dessen reicht es aus, wenn der Übersetzer jedes Kompilat automatisch so erweitert, dass dessen Binärform dem COM-Standard genügt und als selbstregistrierender COM-Server fungieren kann. Solche Direct-to-COM-Übersetzer (1) erweitern ohne weiteres Zutun der Entwickler die Schnittstellen um die notwendigen Operationen `QueryInterface`, `AddRef` und `Release`, (2) generieren IIDs sowie eine CLSID und (3) fügen an der vorgegebenen Stelle einen Zeiger auf die Zeigertabelle ein, die die Schnittstelle des COM-Servers repräsentiert (vgl. [Szy98, s. 266]).

Da jeder der so erzeugten COM-Server damit als separate ausführbare Datei vorliegt (`.EXE`), kann die Anwendungsfamilie jederzeit problemlos um neue Komponenten erweitert werden. Dazu ist es lediglich notwendig, die ausführbare Datei, die den COM-Server mit der neuen Komponente enthält, mit der Option `/REGSERVER` zu starten, wodurch die Komponente für das Kernsystem zugreifbar gemacht wird (vgl. [MD95]).

erfahren können, die die gewünschten Dienste anbieten. Aufgrund der Sprachunabhängigkeit und des Fokus' auf rechnerlokale Kommunikation ist COM in seiner Grundform im Gegensatz zu CORBA jedoch nicht dazu geeignet, objektorientierte Prinzipien zu modellieren oder Server auf anderen Rechnern zu nutzen.

Die Systemmetapher *Client und Server* weist Vor- und Nachteile auf, um Funktionalität und Handhabung von Kernsystem und Komponenten von Anwendungsfamilien im Sinne der in Abschnitt 4.3 genannten Kriterien zu veranschaulichen. Dies ist unabhängig davon, ob CORBA, COM oder eine andere auf dem Client-Server-Prinzip beruhende Technologie eingesetzt wird, um diese Anwendungsfamilien technisch zu realisieren. Zunächst die *Vorteile*:

- Da *Server* Dienste verkörpern, sind sie ideal, um die Funktionalität von Dienstleisterkomponenten zu veranschaulichen

- Die Eigenschaft von Anwendungsfamilien, sitzungsweise neu zusammenstellbar zu sein, wird dadurch verdeutlicht, dass *Client* und *Server* lediglich über Leitungen miteinander verbunden und nicht zu einer monolithischen Einheit verschmolzen sind.

- Die Tatsache, dass ein *Client* optional mit beliebig vielen *Servern* in Verbindung treten kann, ist geeignet, um Mehrfachanpassungsstellen zu versinnbildlichen, an denen das Kernsystem mit beliebig vielen gleichartigen Komponenten anpassbar ist.

Diesen Vorteilen stehen allerdings zahlreiche *Nachteile* entgegen, die die Verwendung der Systemmetapher problematisch erscheinen lassen:

- Die Metapher ist hinsichtlich der Frage unterspezifiziert, ob jeder *Client* mit jedem *Server* in Verbindung treten kann oder nicht. Daher ist sie nicht hilfreich, um drei Eigenschaften von Anwendungsfamilien zu verdeutlichen:

  1. Sie vermag nicht zu veranschaulichen, dass nicht jede Art von Komponente zu jeder Anpassungsstelle des Kernsystems passt.

  2. Sie kann nicht versinnbildlichen, dass die Anpassungsstellen des Kernsystems den verschiedenen Arten von Komponenten unterschiedliche interne Dienstleister zur Verfügung stellen.

  3. Zwischen Auslieferungseinheiten von Anwendungsfamilien gibt es nur zwei Arten von Aufrufen: (1) das Kernsystem nutzt eine Komponente oder (2) eine Komponente nutzt einen internen Dienstleister des Kernsystems. Da die Metapher nur *Clients* und *Server* kennt und Komponenten beide Rollen einnehmen können, ist sie nicht geeignet, um zu veranschaulichen, dass keine Aufrufe zwischen Komponenten stattfinden.

- Da *Clients* stets die Dienste beliebig vieler *Server* in Anspruch nehmen können, ist die Metapher ausschließlich geeignet, um Mehrfachanpassungsstellen zu veranschaulichen. Bei der Vergegenständlichung der Eigenschaften von Einfach- und Zwangs-

anpassungsstellen, die maximal eine bzw. genau eine Komponente benötigen, hilft die Metapher dementgegen nicht.

- Das Verhältnis zwischen *Clients* und *Server* lässt sich nur schwer mit dem Verhältnis eines Systems zu dessen Bausteinen in Übereinstimmung bringen, da *Clients* und *Server* nicht solide zusammengefügt, sondern lediglich über Leitungen miteinander verbundenen sind. Zur Verdeutlichung der Funktionalität und Handhabung von Bausteinkomponenten ist die Metapher daher nur wenig hilfreich.

- Dadurch, dass das Kernsystem und die Komponenten laufend ihre Rolle wechseln und alternierend als *Client* oder *Server* fungieren, schafft die Metapher kein klares Bild darüber, dass Kernsystem und Komponenten Systembestandteile mit grundverschiedenen Eigenschaften sind.

- Die Metapher ist aufgrund ihrer technischen Herkunft für technisch nicht versierte Anwender nur wenig hilfreich. Die meisten dieser Anwender können mit den Begriffen „Client“ und „Server“ keine klare Vorstellung verbinden, da sie lediglich Experten in ihrem Fachgebiet, nicht jedoch im Umgang mit Rechnern sind.

Nur wenn eine Anwendungsfamilie sämtliche der folgenden Spezialfälle erfüllt, erweisen sich die vorgenannten Nachteile der Client-Server-Metapher als unproblematisch:

(1) Die Anwendung muss für Anwender gedacht sein, die mit den Begriffen „Client“ und „Server“ etwas anfangen können.

(2) Die Anwendungsfamilie darf ausschließlich Dienstleisterkomponenten besitzen.

(3) Das Kernsystem darf nur eine einzige Mehrfachanpassungstelle aufweisen, so dass Probleme bei der Veranschaulichung verschiedener Arten von Komponenten und deren Kommunikation untereinander nicht auftreten können.

Ein Beispiel, das diese restriktiven Eigenschaften erfüllt, ist eine Familie von Programmeditoren, die verschiedene Parser-Komponenten für das Syntax-Highlighting unterschiedlicher Programmiersprachen besitzt. Im Gegensatz dazu gehen aber die in Kapitel 2 und 3 diskutierten Familien von Web-Browsern, Simulations- und musikwissenschaftlichen Analysesystemen in allen drei Punkten über die Beschränkungen hinaus, wodurch dementsprechend alle der oben genannten Nachteile der Systemmetapher *Client und Server* zum Tragen kommen.

Die Systemmetapher *Client und Server* ist daher nur sehr begrenzt geeignet, um die Handhabung und die Funktionalität von Kernsystemen und Komponenten im Rahmen von Anwendungsfamilien zu vergegenständlichen. Neben den Problemen, die aus den Eigenschaften des Client-Server-Prinzips selbst resultieren, zeigt es sich wie im Fall des Leitbilds *Objektwelten* auch hier, dass es der Verständlichkeit abträglich ist, ein Prinzip aus dem Kontext *verwendete Technik* (Objektorientierung bzw. das Client-Server-Prinzip) unmittelbar in den Kontext *Handhabung & Präsentation* zu übertragen, wenn die Anwender mit diesen Prinzipien nicht vertraut sind.

Im Sinne der Anwendungsorientierung sollten daher, wie in Abschnitt 3.2 dargelegt, Leitbilder und Metaphern nicht dem Kontext *verwendete Technik*, sondern dem Kontext *Anwendungsbereich* entstammen. Nachdem eine für die Anwender verständliche Metapher gefunden worden ist, spielt es keine Rolle mehr, auf welchen Prinzipien die zur Umsetzung der Anwendungsfamilie gewählte Technologie beruht, solange deren Eigenschaften vollkommen im Kontext *verwendete Technik* gekapselt und die den Einflüssen des Kontexts *Handhabung & Präsentation* entgegenstehen Eigenschaften kompensiert werden können (siehe grauer Kasten auf S. 111).

Die in den folgenden Abschnitten 4.3.2 und 4.3.3 diskutierten Systemmetaphern *aktives Dokument* und *Einsteck-Erweiterung* kommen diesen Anforderungen bereits von vornherein entgegen, da weder Dokumente noch Stecker einem Teilbereich der Informatik entstammen.

4.3.2 Komponenten als aktive Dokumente

Im Gegensatz zu einem Server ist ein Dokument ein Gegenstand, der jedem Anwender aus seinem täglichen Arbeitsalltag bekannt ist und der damit gut zum Leitbild *Werkzeugunterstützung für eigenverantwortlichen Expertentätigkeit* passt. Je nach Anwendungsbereich kann es sich bei einem Dokument z.B. um eine Rechnung, Akte, Broschüre, Buch, Bild, Zeichnung, Partitur oder einen Schaltplan handeln. Dokumente lassen sich auf anschauliche Weise durch Einkleben oder Zusammenheften kombinieren, wobei sie ggf. zunächst zurechtgeschnitten oder mit Hilfe eines Kopierers vergrößert werden müssen.

Aufgrund ihrer großen Anschaulichkeit ist die Metapher *Dokument* untrennbar mit dem Konzept der direkten Manipulation und Schreibtischoberflächen verbunden (vgl. [Shn83] und [Bla99b]). Dabei werden die erwähnten Operationen Ausschneiden, Kopieren und Einkleben unmittelbar auf elektronische Dokumente übertragen (engl. *cut, copy and paste*).

Trotz dieser Eignung als Metapher innerhalb von Anwendungssystemen, ist die Metapher *Dokument* als *System*metapher problematisch, da Dokumente stets passiv sind und damit wenig hilfreich erscheinen, um die Funktionalität der Bestandteile eines mehrteilig ausgelieferten Anwendungssystems wie z.B. eines Kernsystems sowie Dienstleister- oder Bausteinkomponenten zu veranschaulichen. Um sich die versinnbildlichende Kraft von Dokumenten trotzdem auch im Bereich der Auslieferungseinheiten nutzbar zu machen, schlagen einige Autoren daher die Metapher des *aktiven Dokuments* vor (vgl. [Obe97], [MD95] und [Nel95]).

Zusätzlich zu einer graphischen oder textuellen Repräsentation besitzt ein aktives Dokument eine bestimmte Funktionalität, die in den meisten Fällen durch direkte Manipulation an seiner Darstellung – die damit zur Bedienschnittstelle wird – beeinflussbar ist. So kann an einem aktiven Dokument, das eine Uhr verkörpert, beispielsweise am Zifferblatt die Uhrzeit eingestellt werden. Aktive Dokumente können ebenso wie herkömmliche Dokumente miteinander oder mit passiven Dokumenten kombiniert werden, wodurch dann zusammengesetzte aktive Dokumente entstehen. Zusätzlich werden auch spezielle Dokumente vorgeschlagen, die wie z.B. Sortieralgorithmen aus reiner Funktio-

Technische Besonderheiten bei der Realisierung von Anwendungsfamilien

Um die Eigenschaften einer Anwendungsfamilie bestmöglich mit Hilfe einer bestimmten Technologie umzusetzen, ist es vorteilhaft, wenn diese bereits die notwendigen Mittel zur Verfügung stellt, um die Eigenschaften des Kernsystems und der Komponenten zu modellieren:

- Das Kernsystem einer Anwendungsfamilie steht den Komponenten nicht uneingeschränkt zur Verfügung, sondern bietet nur denjenigen Komponenten Zugriff auf unterschiedliche interne Dienstleister, die zu einer seiner Anpassungsstellen passen. Technologisch gesehen erfordert das zwei Eigenschaften:
  1. **Black Boxes** (siehe S. 91) müssen gebildet werden können, um das Kernsystem zu kapseln. Da Komponenten von beliebigen dritten auch über das potentiell unsichere Internet beziehbar sein sollen, gewährt dies zugleich auch Schutz vor den Manipulationsversuchen bösartiger Komponenten, wenn die Black Box eine **herstellerabhängige Rechtevergabe** erlaubt.
  2. **Typisierung** (vgl. [GZ99, S. 486]) ist notwendig, um modellieren zu können, dass nur eine bestimmte Art von Komponenten zu einer Anpassungsstelle des Kernsystems passt. Um die Passform möglichst schon beim Laden und nicht erst zur Laufzeit überprüfen zu können, empfiehlt sich dabei die strikte Form der **statischen Typisierung**.
- Da das Kernsystem und die Komponenten ein einziges System bilden, sollte die Kommunikation zwischen ihnen nur **minimale Zeitverzögerungen** mit sich bringen.

CORBA und COM erfüllen diese Eigenschaften nur teilweise, da sie als Systeme zum Operationsfernaufruf bzw. zur Nutzung der Funktionalität anderer Programme auf dem gleichen Rechner entwickelt worden sind. Da die Anforderungen an solche Technologien von denen an Technologien zur Realisierung von Anwendungsfamilien teilweise stark abweichen, müssen Entwickler die nicht vorhanden Eigenschaften in Eigenarbeit kompensieren.

	CORBA		COM	
Black Boxes	-	Sowohl bei CORBA als auch bei COM werden die Dienste von Servern jedem beliebigen Client in gleicher Weise angeboten. Eine Kapselung der internen Dienstleister des Kernsystems im Sinne einer Anpassungsstelle ist mit keiner der beiden Technologien möglich.		
Herstellerabhängige Rechtevergabe	-	Das CORBA-Sicherheitskonzept beruht auf dem *Principal*-Verfahren, das Rechte nicht in Abhängigkeit vom Urheber eines Clients vergibt, sondern in Abhängigkeit von der Person, die den Aufruf veranlasst hat. Vor bösartigen Komponenten gewährt dies keinen Schutz, da Komponenten von Anwendungsfamilien stets vom Kernsystem aus aufgerufen werden.	+	In COM werden Rechte auf der Basis eines herstellerabhängigen *Authenticodes* vergeben. Diese Rechte können aber nicht abgestuft werden, so dass eine einmal akzeptierte Komponente nicht nur Zugriff auf ihre Anpassungsstelle, sondern auf das gesamte über die *COM library* erreichbare Rechnersystem erhält. Da auch MS-*Windows* auf COM basiert, kann eine Komponente dort auch Shutdowns veranlassen.
Statische Typisierung	+	Wenn die Schnittstellen an den Anpassungsstellen des Kernsystems nicht den Typ `any` enthalten, der für jeden primitiven oder Referenzdatentyp stehen kann, dann ist eine statische Passformprüfung von Komponenten möglich.	-	Typisierung ist in COM nur schwach ausgeprägt: Erstens können einer COM-Komponente Operationen fehlen, obwohl sie gemäß der IID der Schnittstelle vorhanden sein müssten. Dies kann aber erst zur Laufzeit durch einen Aufruf herausgefunden werden, der zur Fehlermeldung `E_NOTIMPL` führt. Zweitens ist die Verwendung einer *type library*, die die Signaturen der Operationen enthält, optional.
Minimale Zeitverzögerungen	-	Daten werden zwischen Clients und Servern grundsätzlich zunächst in der CORBA-spezifischen, zeichenkettenbasierten Transportform ausgetauscht, auch wenn beide auf dem selben Rechner ausgeführt werden.	+	Da COM nur zwischen Client und Server vermittelt, ohne in den eigentlichen Transport einzugreifen, entsteht kaum zusätzlicher Zeitbedarf.

nalität bestehen und keinerlei Darstellung besitzen (vgl. [MD95]). Die Zusammenstellung eines Anwendungssystems aus Auslieferungseinheiten soll gemäß der Metapher *aktives Dokument* ebenso erfolgen, wie das Zusammenkleben eines Hauptdokuments aus zahlreichen Teildokumenten.

Der von Apple und IBM entwickelte *OpenDoc*-Standard und das von oberon microsystems entwickelte *BlackBox*-System betrachten ihre Komponenten explizit als aktive Dokumente.

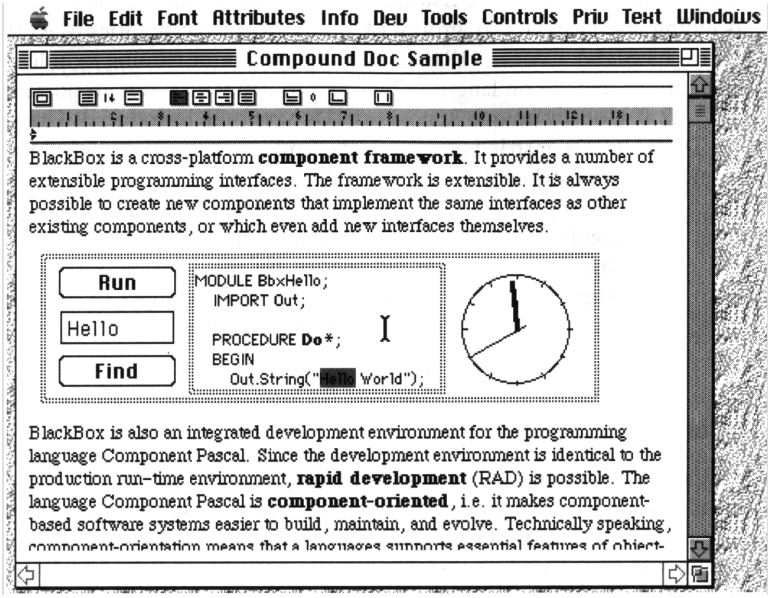


Abb. 4.3-3 Ein aus einem Text, einer Uhr, einem Programmeditor und einem Programminterpretierer bestehendes aktives Dokument (vgl. [Szy98, S. 176]).

Die Metapher *aktives Dokument* ähnelt der Metapher *elektrischer Legostein* (siehe S. 104), da sie ebenfalls auf einem wohlbekannten Gegenstand fußt, den sie dann aber um Attribute erweitert, die dieser in der sinnlich erfahrbaren Welt nicht besitzt. Wenn Anwendern gesagt wird, sie sollen sich das Kernsystem bzw. die Komponenten einer Anwendungsfamilie als *aktive Dokumente* oder *elektrische Legosteine* vorstellen, dann hilft ihnen das nur wenig, um zu verstehen, welche Art von Funktionalität diese Systembestandteile verkörpern und auf welche Weise diese miteinander kommunizieren können. Da aber genau diese Eigenschaften durch eine Systemmetapher auf verständliche Weise vergegenständlicht werden sollen (siehe Begriff 4.7 auf S. 103), sind aktive Dokumente nicht als Systemmetaphern geeignet. Natürlich können die fehlenden Eigenschaften und Sonderfälle (wie z.B. der darstellungslose Sortieralgorithmus) im Rahmen der Dokumentation des jeweiligen Systems definierend eingeführt werden, aber genau dadurch, dass eine solche Erklärung überhaupt notwendig ist, verliert die Metapher ihren eigentlichen Wert. Im Endeffekt führt die Metapher *aktives Dokument* daher dazu, dass die Anwender sich genauso in ihnen vormals unbekannte technische Sachverhalte einarbeiten müssen, als ob überhaupt keine die Vorstellung leitenden Metapher vorhanden wäre, um aus Kernsystem und Komponenten ein konkretes Mitglied einer Anwendungsfamilie zusammenzustellen.

4.3.3 Komponenten als Einsteck-Erweiterungen

Die Metapher *Einsteck-Erweiterung* fußt auf Steckern und Steckdosen, um die Funktionalität und die Handhabung von mehrteilig ausgelieferten Anwendungssystemen zu vergegenständlichen (vgl. [Mey97, S. 506]). Durch diesen Bezug auf Alltagstechnik vermeidet die Metapher von vorneherein die Probleme, die ich in den Abschnitten 4.3.1 und 4.3.2 bezüglich anderer Metaphern herausgearbeitet habe:

1. Im Gegensatz zu einem Dokument besteht bei Steckern und Steckdosen ein unmittelbarer Bezug zu den dazugehörigen technischen Geräten, so dass das Vorhandensein von Funktionalität der Bestandteile nicht erst wie bei *aktiven Dokumenten* oder *elektrischen Legosteinen* künstlich hinzudefiniert werden muss.

2. Im Gegensatz zu Clients und Servern sind Stecker und Steckdosen auch technisch ungeschulten Anwendern ein Begriff und somit sofort verständlich.

Durch diesen Bezug auf alltägliche Technik sind Stecker und Steckdosen eine gute Grundlage für Systemmetaphern für alle Arten von mehrteilig ausgelieferten Anwendungssystemen. Hinsichtlich der hier diskutierten Anwendungsfamilien besitzen sie darüber hinaus noch drei weitere Eigenschaften, die sie speziell geeignet erscheinen lassen, um das Verhältnis zwischen einem Kernsystem und den sitzungsweise zur Anpassung verwendbaren Komponenten zu veranschaulichen:

3. Die Steckverbindung versinnbildlicht in zutreffender Weise, dass die Komponenten dem Kernsystem nicht permanent hinzugefügt werden, sondern dass das Anwendungssystem vor jeder Inbetriebnahme neu konfigurierbar ist.

4. Steckdosen können als Vergegenständlichungen der voneinander unabhängigen Anpassungsstellen angesehen werden: Für jede weitere Anpassungsstellen wird einfach eine weitere Steckdose hinzugefügt.

5. Dadurch, dass Stecker und Steckdosen nicht nur funktionale, sondern auch gegenständliche Unterschiede aufweisen, wird unmittelbar deutlich, dass das Kernsystem und die Komponenten unterschiedliche Arten von Funktionalität verkörpern. Im Gegensatz zu dem Verhältnis zwischen Client und Server alterniert die Rollenverteilung nicht aufrufsweise, sondern besteht ebenso lange wie die Steckverbindung selbst.

Aufbauend auf diesen Vorteilen der Anschaulichkeit der Verbindung zwischen Stecker und Steckdose werden drei verschiedene Arten von Systemmetaphern diskutiert, die sich allesamt unter dem Begriff *Einsteck-Erweiterung* subsumieren lassen:

(1) *Elektrogerät mit Stromstecker und Netzsteckdose.* Diese Metapher fußt auf derjenigen Art von Steckern, die den meisten Anwendern als erstes einfällt, wenn sie mit dem Begriff „Stecker“ konfrontiert werden. Sie ist aufgrund ihrer Eigenschaften jedoch nahezu ungeeignet, um die Funktionalität und die Handhabung eines Kernsystems und seiner Komponenten zu veranschaulichen:

Im Gegensatz zu Anwendungsfamilien, bei denen Funktionalität besitzende Auslieferungseinheiten miteinander verbunden werden, verbindet ein Stromstecker lediglich ein Gerät mit einer Energiequelle, die selbst keinerlei Funktionalität verkörpert. Es ist daher nicht klar, in welcher Weise entweder das Stromnetz oder das Elektrogerät für das anpassbare Kernsystem oder die Komponenten einer Anwendungsfamilie stehen kann:

- Durch die Gleichsetzung des Stromnetzes mit dem Kernsystem entsteht zwangsweise eine Analogie der Komponenten mit kompletten Elektrogeräten. Dadurch wird jedoch nicht deutlich, dass die Komponenten einer Anwendungsfamilie für sich genommen keine funktional vollständigen Systeme sind.

- Wird umgekehrt das Elektrogerät selbst mit dem Kernsystem gleichsetzt, dann vermag die Metapher nicht zu erklären, wie dieses Kernsystem sitzungsweise mit Komponenten angepasst werden kann, da der Stecker ja nur die Verbindung zum Stromnetz, nicht aber zu anderen Geräteteilen versinnbildlicht.

Abgesehen von diesem Hauptproblem sprechen auch die einseitige Richtung des Stromflusses, die Verbindung über eine Leitung sowie die Gleichförmigkeit aller Stecker dagegen, die Metapher *Elektrogerät mit Stromstecker und Netzsteckdose* auf Anwendungsfamilien zu übertragen, da es durch sie schwerfällt, sich beiderseitig aufeinander zugreifende Kernsysteme und Komponenten, Bausteinkomponenten sowie verschiedene Arten von Anpassungsstellen vorzustellen.

(2) *Gerät einer Stereoanlage* (vgl. [Szy98, S. 7] und [Hei97]). Diese Variante der Metapher bezieht die über Stecker und Steckdosen miteinander verbundenen Systembestandteile auf die Bestandteile einer aus mehreren Einzelgeräten bestehenden Stereoanlage. Das Kernsystem entspricht dabei dem Verstärker mit den Lautsprechern. Die Komponenten entsprechen komplementär dazu den Kassetten-, Schallplatten- und CD-Abspielgeräten, die über Steckverbindungen mit den Steckdosen auf der Rückseite des Verstärkers in das System eingebunden werden können. Die Metapher *Gerät einer Stereoanlage* hat gegenüber den bisher diskutierten Metaphern zusätzlich zu den zu Beginn des Abschnitts genannten Eigenschaften die folgenden *Vorteile*:

- Die Analogie zwischen dem Verstärker und dem Kernsystem verdeutlicht, dass das Kernsystem tatsächlich über Funktionalität verfügt und nicht lediglich ein rein versorgendes Netz wie im Fall von *Elektrogerät mit Stromstecker und Netzsteckdose* ist.

- Die Einzelgeräte sind gut geeignet, um Dienstleisterkomponenten zu veranschaulichen.

- Sowohl seitens des Verstärkers als auch seitens der Einzelgeräte wird bei einer Stereoanlage explizit zwischen Eingängen und Ausgängen unterschieden, was sich mit den Eigenschaften einzelner Anpassungsstellen und der Komponenten deckt, sowohl ein *incoming interface* als auch ein *outgoing interface* zu besitzen. Eine Komponente, die dem Kernsystem einerseits Dienste anbietet und anderer-

seits dazu auf Dienste des Kernsystems zugreift, ähnelt einem Tape Deck, das über zwei separate Verbindungen gleichermaßen zur Wiedergabe wie auch zur Aufzeichnung von Audiosignalen geeignet ist.

Trotz dieser Vorteile weist die Metapher auch *Nachteile* auf, wenn es darum geht, die Funktionalität und die Handhabung der Bestandteile einer Anwendungsfamilie zu vergegenständlichen:

- Die über Leitungen miteinander verbunden Geräte sind schlecht geeignet, um Bausteinkomponenten zu veranschaulichen, aus denen ein Arbeitsgegenstand wie z.B. ein HTML-Dokument, ein Simulationssystem oder ein Notizzettel eines Musikwissenschaftlers aufgebaut werden kann.

- Die Steckdosen auf der Rückseite des Verstärkers weisen unabhängig von ihrer Beschriftung einheitliche Schnittstellen auf, an die jede Art von Audiogerät angeschlossen werden kann. Dies entspricht nicht den Eigenschaften von Anpassungsstellen, die nur für jeweils eine bestimmte Komponentenart geeignet sind. Dementsprechend lassen sich auch Zwangs- und Einfachanpassungsstellen, die das Vorhandensein einer bestimmten Art von Komponente erzwingen bzw. ermöglichen, durch die Steckdosen des Verstärkers nicht versinnbildlichen.

  Problematisch an diesen Steckdosen ist weiterhin, dass trotz der funktionalen Unterschiede zwischen Eingängen und Ausgängen die physikalische Schnittstelle der Stecker diese Unterscheidung nicht zum Ausdruck bringt, so dass erst beim Einschalten der Anlage feststellbar ist, ob die Steckverbindungen richtig geschlossen wurden.

- Die Funktionalität eines Verstärkers entspricht nicht der eines Kernsystems einer Anwendungsfamilie: Während bei einer Stereoanlage die komplexe Funktionalität in den Einzelgeräten steckt und der Verstärker die Audiosignale lediglich aufbereitet und an die Lautsprecher weiterleitet, so enthält das Kernsystem einer Anwendungsfamilie bereits den größten Teil der Funktionalität und steuert den Kontrollfluss der gesamten Anwendung.

(3) *Plug-In* (vgl. [Net98a] und [Tur99]). Diese letzte Variante der Metapher *Einsteck-Erweiterung* nimmt ebenso wie die Metapher *Client und Server* Gegenstände aus dem Bereich der Rechensysteme zum Vorbild. Im Gegensatz zu Clients und Servern sind Plug-Ins jedoch nahezu allen Anwendern bekannt, die schon einmal einen Web-Browser (beispielsweise Netscapes *Navigator* oder Microsofts *Internet Explorer*) oder komplexe Bildbearbeitungsprogramme benutzt haben (*GIMP*, Adobes *Photoshop* oder Corels *Photopaint*).

Plug-Ins werden in diesen Systemen eingesetzt, um ein bereits für sich genommen selbständig lauffähiges Anwendungssystem um optionale Zusatzfunktionalität zu erweitern, die zum Zeitpunkt der Herstellung noch nicht vorhanden war bzw. die nicht von allen Anwendern benötigt wird. Im Gegensatz zu den bisher erwähnten Komponententechnologien CORBA und COM sowie *OpenDoc* und *BlackBox* gibt es jedoch keine vorherrschende Technologie, mit deren Hilfe Einsteck-Erweiterun-

gen anwendungsneutral realisiert werden. Statt dessen definieren die meisten Anwendungssysteme einen eigenen Standard, den die Hersteller von Einsteck-Erweiterungen einhalten müssen, um das jeweilige Anwendungssystem ergänzen zu können.[8] Bezüglich der Handhabung ist allen Plug-Ins lediglich gemein, dass sie vor dem Start der Anwendung in einem bestimmten Verzeichnis abgelegt werden müssen, wo sie dann von der Anwendung erkannt und eingegliedert werden.

Die Metapher *Plug-In* kann auf Anwendungsfamilien übertragen werden, indem das Grundanwendungssystem mit dem Kernsystem und ein Plug-In mit einer Komponente gleichgesetzt wird. Dies bringt zunächst folgenden *Vorteile* mit sich:

- Da das Grundanwendungssystem bereits ein für sich lauffähiges System darstellt, wird im Gegensatz zu einem Verstärker deutlich, dass das Kernsystem den Großteil der Funktionalität enthält und den Kontrollfluss steuert.

- Dadurch, dass die Plug-Ins direkt in das Grundsystem eingegliedert werden, sind auch Bausteinkomponenten als Plug-Ins denkbar.

Der große *Nachteil* der Verwendung eines softwaretechnischen Konstrukts als Metapher für ein anderes softwaretechnisches Konstrukt besteht wie bei *Client und Server* darin, dass die Metapher keine unmittelbare Vorstellung über die physikalische Gestalt und damit die Handhabbarkeit der Komponenten und des Kernsystems fördert. Bezüglich sämtlicher Eigenschaften von Anwendungsfamilien, die sich auf die Verbindung zwischen Kernsystem und Komponenten beziehen, ist die Systemmetapher *Plug-In* daher nicht hilfreich. Dies betrifft die Frage, wie die Anpassungsstellen beschaffen sind (voneinander unabhängig? *Incoming* und *outgoing interface*? Einfach-, Mehrfach- und Zwangsanpassungsstellen?) und ob nur eine bestimmte Art Komponenten zu einer bestimmten Anpassungsstelle passt.

4.4 Zusammenfassung und Ausblick

Zu Beginn von Kapitel 3 habe ich gezeigt, dass sich die in der Literatur diskutierten Ansätze zur Modellierung von Anwendungsfamilien ausschließlich auf die Analyse und die Entwicklung von herstellerseitig automatisch generierbaren Anwendungsfamilienmitgliedern beschränken. Nachdem ich im weiteren Verlauf von Kapitel 3 dargelegt habe, dass es aber für zahlreiche Anwendungsfamilien notwendig ist, dass die Anwender selbst die konkreten Anwendungssysteme aus Kernsystem und Komponenten zusammenstellen können, habe ich in diesem Kapitel untersucht, wie sich solche Anwendungsfamilien softwaretechnisch modellieren lassen. Dabei habe ich herausgearbeitet:

(1) Komponenten lassen sich im softwaretechnischen Modell durch eine Kombination herkömmlicher Entwurfsmuster ausdrücken.

[8] Um einen Eindruck der jeweiligen Technologien zu vermitteln, beschreibe ich in Anhang B.3 Plug-Ins für Netscapes *Navigator* und *GIMP*.

(2) Für die Auslieferung von Anwendungsfamilien, deren Mitglieder durch die Anwender zusammenstellbar sein sollen, kommt nur eine Kombination von Rahmenwerk- und Komponenten-Ansätzen in Frage.

(3) Außerdem muss eine geeignete Systemmetapher vorhanden sein, damit Anwender die separat ausgelieferten Kernsysteme und Komponente auf für sie verständliche Weise zusammenfügen zu können.

Schließlich konnte ich zeigen, dass die in der Literatur implizit oder explizit vorgeschlagenen Systemmetaphern die Eigenschaften von Kernsystemen und Komponenten nur unzureichend vergegenständlichen können, so dass die zwei Hauptprobleme von auf herkömmliche Weise als monolithische Rahmenwerke ausgelieferten Anwendungsfamilien bestehen bleiben:

1. Eine problemlose Anpassung von Software je nach Anwendungssituation gemäß den Prinzipien der Anwendungsorientierung (siehe Abschnitt 1.2) lässt sich nur in begrenztem Maße umsetzen, da der Monolith nach der Auslieferung nicht mehr modifizierbar ist.

2. Ein Rahmenwerk, das möglichst viele – und somit auch von den meisten Anwendern nicht benötigte – Komponenten fest enthält, wird zu inflexibler, platzraubender „fatware“ (siehe Abschnitt 4.1).

Um diese Probleme zu lösen, entwickle ich in Kapitel 5 den auf der Systemmetapher *Einschub und Einschubrahmen* beruhenden Einschub-Ansatz, mit dem Anwendungsfamilien ohne die aufgezeigten Schwierigkeiten softwaretechnisch modelliert, implementiert, ausgeliefert und von den Anwendern selbst zusammengestellt werden können. Zuvor fasse ich im Rest dieses Abschnitts die weiteren Ergebnisse dieses Kapitels zusammen:

- Prees Vorschlag, austauschbare Dienstleister als Strategie-Klassen zu modellieren, habe ich verallgemeinert und auf austauschbare Bausteine ausgedehnt. Für Dienstleisterkomponenten von Anwendungsfamilien habe ich gezeigt, dass das Brückenmuster am besten geeignet ist. Für Bausteinkomponenten sind Fliegengewichtfabriken die beste Lösung. Besondere Modellierungsmaßnahmen auf der Ebene von Makrostrukturen wie Architekturschichten sind nicht notwendig.

- Aus den in der Literatur unscharf verwendeten Komponenten-Begriff habe ich drei klar abgegrenzte Konzepte herausgearbeitet und griffig benannt. Unklarheiten in Komponentendiskussionen können dadurch nunmehr weitgehend vermieden werden.

    1. *Konstruktionskomponenten* beziehen sich auf im Quelltext vorliegende Lösungen wie Makros, einzelne Klassen oder auch Spezialisierungsrahmenwerke und werden von den Entwicklern kombiniert.

    2. *Implementationskomponenten* sind bereits übersetzte Einheiten wie Prozedurbibliotheken oder JavaBeans und werden von den Entwicklern statisch ins Kompilat eingebunden.

3. *Laufzeitkomponenten* sind binäre Einheiten, die einem System erst zur Laufzeit hinzugefügt werden. Da diese Komponenten selbst nicht mehr modifiziert werden können, müssen die Verwender dafür sorgen, dass sie auf sinnvolle Weise mit den anderen Systemteilen verbunden werden.

- Die konfligierenden Definitionen von Black-Box- und White-Box-Rahmenwerken habe ich gemäß ihres Fokus' entweder auf die Sichtbarkeit der Implementation oder aber auf die Art der Verwendung neu gefasst und das Begriffspaar *Parametrisierungs- und Spezialisierungsrahmenwerke* eingeführt:

  1. *Parametrisierungs- und Spezialisierungsrahmenwerke* beziehen sich auf die durch deren Entwickler vorgesehene Verwendung des Rahmenwerks. Während Spezialisierungsrahmenwerke durch Unterklassenbildung von den Verwendern an deren Anforderungen angepasst werden müssen, stellen Parametrisierungsrahmenwerke bereits fertige Parametrisierungsklassen zur Verfügung, mit denen das Rahmenwerk über reine Benutzt-Beziehungen eingesetzt werden kann. Während Black-Box-Rahmenwerke ausschließlich parametrisierend genutzt werden können, sind White-Box-Rahmenwerke oft sowohl als Parametrisierungs- als auch als Spezialisierungsrahmenwerke verwendbar.

  2. Die Bezeichnungen *Black-Box- und White-Box-Rahmenwerke* nehmen ausschließlich Bezug darauf, ob das Innere eines Rahmenwerks von außen sichtbar ist oder nicht. Black-Box-Rahmenwerke schirmen sowohl ihre Implementation als auch ihre internen Daten während der Entwicklung und zur Laufzeit vor ihren Verwendern ab, während White-Box-Rahmenwerke zu jedem Zeitpunkt voll einsehbar sind.

- Die Einflüsse aus den Kontexten *Anwendungsbereich* und *Handhabung & Präsentation* müssen auch bei der Modellierung der binären Auslieferungseinheiten eines Anwendungssystems berücksichtigt werden, damit diese von den Anwendern zusammenstellbar sind. Für den Kontext *Anwendungsbereich* lässt sich dies durch eine Partitionierung des softwaretechnischen Modells entlang den funktionalen Grenzen zwischen dem Kernsystem und den einzelnen Komponenten erreichen. Für den Kontext *Handhabung & Präsentation* muss das Konzept der bisher nur zur Laufzeit verwendeten Entwurfsmetaphern in Form von *Systemmetaphern* auf die Konfigurationszeit und die Auslieferungseinheiten erweitert werden.

  Die in der Literatur diskutierten, als Systemmetaphern auffassbare Vergegenständlichungen und Analogien von austauschbaren Bestandteilen von Software lassen sich in drei Gruppen mit unterschiedlichen Vor- und Nachteilen einteilen:

  (1) Metaphern wie *elektrischer Legostein* oder *aktives Dokument*, die einen allgemein bekannten Gegenstand zur Grundlage nehmen und diesen dann um Eigenschaften erweitern, die er in seiner physikalischen Form nicht besitzt, sind wenig hilfreich, da die Anwender die neuen Eigenschaften zusätzlich erlernen müssen und durch diesen Lernvorgang der Sinn einer Systemmetapher – die Übertragung der Handhabung eines bekannten Gegenstandes auf eine Auslieferungseinheit – ad absurdum geführt wird.

(2) Metaphern aus dem Bereich des Kontexts *verwendete Technik* wie *Client und Server* oder *Plug-In* können zwar durch den Bezug auf Softwaresysteme auch komplexe funktionale Zusammenhänge erklären, leiden aber daran, dass sie zum einen technisch nicht versierten Anwendern unbekannt sind und dass sie zum anderen keine wirklich gegenständliche Vorstellung unterstützen: Immaterielle Software kann keine klare Orientierung anbieten, wie Auslieferungseinheiten handhabbar sind.

(3) Am günstigsten sind Metaphern, die wie *Stecker und Steckdose* oder *Gerät einer Stereoanlage* auf Alltagstechnik aufbauen und somit einerseits ohne unnatürliche technische Erweiterungen von passiven Gegenständen wie Legosteinen oder Dokumenten auskommen und andererseits nicht so speziell und abstrakt sind, dass sie den meisten Anwendern unbekannt sind bzw. keine klare Vorstellung von der Gestalt und Handhabung der Systembestandteile vermitteln. Doch selbst die günstigste Systemmetapher dieser Gruppe – *Gerät einer Stereoanlage* – reicht nicht aus, um alle Bestandteile und Verbindungen der Bestandteile einer Anwendungsfamilie zutreffend zu charakterisieren. Neben der Tatsache, dass ein Verstärker sich nicht mit einem den Kontrollfluss steuerndem Kernsystem gleichsetzen lässt, bestehen die Probleme hauptsächlich darin, dass die Anschlüsse eines Verstärkers keine komponentenspezifischen Unterscheidungen zulassen und somit lediglich Mehrfachanpassungsstellen versinnbildlicht werden können.

	Client und Server	*Aktives Dokument*[9]	*Gerät einer Stereoanlage*
Für Nicht-Techniker verständlich	-	-	+
Kontrollflusssteuerung durch das Kernsystem	-	-	o
Dienstleisterkomponenten	+	-	+
Bausteinkomponenten	-	o	+
Verbindung zwischen Kernsystem und Komponenten • Die Verbindungen lassen sich sitzungsweise auch wieder lösen	+		+
• voneinander unabhängige Anpassungsstellen	-		+
• nur genau eine Art von Komponenten passt zu einer bestimmten Anpassungsstelle	-		-
• Einfachanpassungsstellen	-		-
• Mehrfachanpassungsstellen	+		+
• Zwangsanpassungsstellen	-		-
• Beidseitig eingehende und ausgehende Schnittstellen	o		+

[9] Im ausgegrauten Bereich befinden sich keine Eintragungen, da die Systemmetapher *aktives Dokument* ungeeignet ist, um Unterschiede zwischen Kernsysteme und Komponenten zu veranschaulichen.

5 Der Einschub-Ansatz

Wie ich in Kapitel 4 gezeigt habe, ist die Systemmetapher *Gerät einer Stereoanlage* hinsichtlich vier ihrer Eigenschaften problematisch, wenn sie auf mehrteilig ausgelieferte Anwendungsfamilien angewendet wird. Um Anwendungsfamilien problemlos mehrteilig auszuliefern und durch die Anwender sitzungsweise zusammenstellbar zu machen, muss eine Systemmetapher gefunden werden, die sich wie *Gerät einer Stereoanlage* auf Gegenstände aus dem Bereich der Alltagstechnik stützt, aber die vier genannten Anschaulichkeitsdefizite vermeidet. Dies leistet die Systemmetapher *Einschub und Einschubrahmen*:

(1) *Nur genau eine Art von Komponenten passt zu einer bestimmten Anpassungsstelle*. Den Zusammenhang zwischen Einschüben und Einschubrahmen beschreiben enzyklopädische und technische Nachschlagewerke wie folgt (vgl. [Bro99] und [JM94]):

> Ein Einschub ist eine technische Konstruktionseinheit mit geschlossenem oder offenem Gehäuse zum Einschieben in den Einschubrahmen einer größeren Einheit, wobei die elektrischen Verbindungen unmittelbar, meist über Steckverbindungen[1], hergestellt werden.

Aus dieser Definition geht zunächst hervor, dass die Frage, ob ein Einschub in den Einschubrahmen einer größeren Einheit passt, nicht nur wie bei einem Gerät einer Stereoanlage von der Form der Steckverbindung abhängt, sondern zusätzlich von der Gestalt des gesamten Einschubrahmens. Ob diese Eigenschaft auch als zusätzliches Differenzierungsmerkmal unterschiedlicher Anpassungsstellen genutzt wird, hängt davon ab, ob Schrankeinschübe oder Geräteeinschübe betrachtet werden:

- *Schrankeinschübe* passen in industrieweit genormte Einschubschränke, die über zahlreiche identische (meist 19 Zoll breite) Einschubrahmen verfügen. Sie bilden oft keine geschlossene technische Einheit, sondern sind offene Baugruppeneinschübe, die lediglich als Gefäße zur Aufnahme kleinerer Baugruppen wie Platinen und Karten dienen. Schrankeinschübe werden primär im technischen Bereich eingesetzt und verkörpern keine Funktionalität, die für technisch nicht versierte Fachleute wie Musikwissenschaftler oder Graphiker verständlich wäre (siehe Abb. 5-1).

[1] Einsteckverbinder werden z.B. in DIN 41618 und DIN 41622 beschrieben.(vgl. [Rin81])

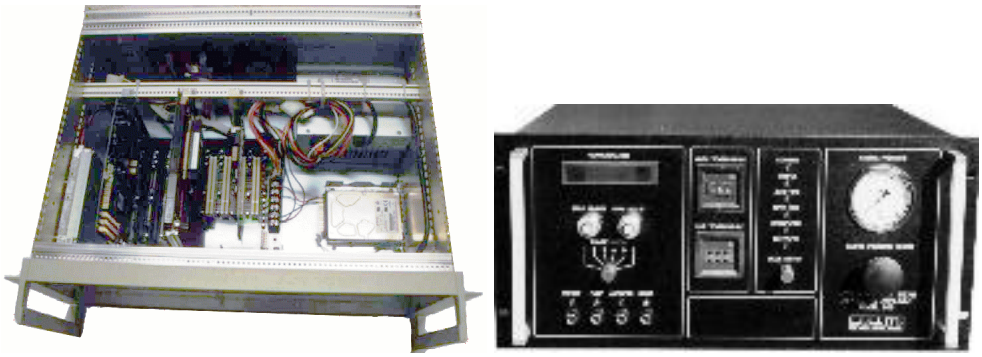


Abb. 5-1 19-Zoll-Schrankeinschübe

Links ein offener Baugruppeneinschub in der Draufsicht, der bereits einige Steckkarten enthält. Rechts ein geschlossener Einschub, der die Funktionalität eines Messgeräts zur Verfügung stellt.

- *Geräteeinschübe* sind stets geschlossene Einheiten, deren Form durch die gerätespezifischen physikalischen Beschränkungen der aufnehmenden Einheit vorgegeben ist. Im Gegensatz zu industrieweit genormten Schrankeinschüben oder den Steckern des Geräts einer Stereoanlage gibt die Form eines Geräteeinschubs bzw. eines Geräteeinschubrahmens daher unmittelbar Auskunft über die Art der anpassbaren Funktionalität.

Abb. 5-2 Geräteeinschübe

Links stabförmige Einschubmotoren mit unterschiedlichen Leistungsmerkmalen. Rechts ein GPS-Navigationssystem-Einschub für Flugzeugcockpits.

Da nur bei Geräteeinschüben die Form des Einschubrahmens ein funktionales Differenzierungsmerkmal darstellt, betrachte ich im Folgenden nur Geräteeinschübe als Basis für die Systemmetapher von *Einschub und Einschubrahmen*.

(2) *Kontrollflusssteuerung durch das Kernsystem*. Da Einschübe direkt vom Einschubrahmen eines Gerät aufgenommen werden und über keine eigene Stromversorgung verfügen, verdeutlichen sie anders als Clients und Server oder Geräte einer Stereoanlage auf den ersten Blick, bei welchem Systemteil es sich um das Kernsystem und bei welchem es sich um die austauschbare Komponente handelt.

(3) *Zwangsanpassungsstellen*. Einschübe unterscheiden sich von den in Kapitel 4 zur Vergegenständlichung von austauschbaren Komponenten herangezogenenen Gegenständen, da es bei ihnen nicht unüblich ist, dass die im Einschub enthaltene Funktionalität wie im Beispiel der Einschubmotoren aus Abb. 5-2 für die Lauffähigkeit des Systems essentiell ist. Ein Einschubrahmen, der für solche essentiellen Einschübe vorgesehen ist, besitzt somit die Charakteristika einer Zwangsanpassungsstelle.

(4) *Einfachanpassungsstellen*. Da jedem Einschubrahmen unmittelbar angesehen werden kann, wievielen Einschüben er Platz bietet, können Einfacheinpassungsstellen auf einfache Weise als Einschubrahmen mit Platz für genau einen Einschub realisiert werden.

Zusammengefasst und ohne Bezug auf die Systemmetapher *Gerät einer Stereoanlage* lassen sich damit die Systemmetaphern *Einschub* und *Einschubrahmen* wie folgt charakterisieren:

Begriff 5.1 Einschubrahmen *(slide-in frame)*

Ein Einschubrahmen ist eine Stelle eines technischen Systems, an der es im ausgeschalteten Zustand auf einfache Weise mit Einschüben versehen werden kann. Ein Einschubrahmen verkörpert eine individuelle, von anderen unabhängige, fachlich motivierte Anpassungsstelle des Systems.

Ein Einschubrahmen besitzt eine genaue Schnittstellenspezifikation, aus der hervorgeht, welche Dienste das System in diesem Einschubrahmen für Einschübe verfügbar macht und welche Leistungen es umgekehrt von Einschüben erwartet.

Außerhalb der Einschubrahmen besitzt das technische System keine zur Verbindung mit Einschüben vorgesehenen Stellen.

Begriff 5.2 Einschub *(slide-in unit)*

Ein Einschub ist eine austauschbare technische Konstruktionseinheit, die fachlich motivierte Funktionalität verkörpert und zu genau einem Einschubrahmen eines technischen Systems passt.

Obwohl Einschübe für das System essentiell Dienste erbringen können, sind sie für sich genommen nicht sinnvoll verwendbar.

Ein Einschub passt genau dann in einen bestimmten Einschubrahmen, wenn er dessen Schnittstellenspezifikation erfüllt. Dazu muss er die im Einschubrahmen geforderten Leistungen zur Verfügung stellen und darf nur diejenigen Dienste des Systems beanspruchen, welche dieses ihm im Einschubrahmen anbietet.

Einschübe besitzen außerhalb ihrer einschubrahmenspezifischen Schnittstelle keine weiteren zur Verbindung mit dem technischen System vorgesehenen Stellen und können daher allein anhand der Schnittstellenspezifikation unabhängig vom Hersteller des Systems von dritten hergestellt sowie von Anwendern bezogen und eingesetzt werden.

Auf der Grundlage dieser beiden Systemmetaphern ist es anders als mit den in der Literatur diskutierten Systemmetaphern möglich, drei der in Kapitel 1 aufgeworfenen und in Kapitel 4 verfeinerten softwaretechnischen Entwurfs- und Konstruktionsprobleme zu lösen:

1. *Die Eigenschaften des Konzepts der Anwendungsfamilie können bruchlos auf binäre Auslieferungseinheiten übertragen werden*. Wie ich in Abschnitt 4.3 herausgearbeitet habe, entstehen bei jeder der in der Literatur vorgeschlagenen Systemmetaphern Strukturbrüche, wenn sie auf die Bestandteile von Anwendungsfamilien angewendet werden. Dadurch ist es mit ihnen nicht möglich, die Anpassung eines Kernsystems durch geeignete Komponenten vollkommen verständlich zu veranschaulichen. Bei *Servern* lassen sich beispielsweise keine Unterschiede zwischen Einfach-, Mehrfach- und Zwangsanpassungsstellen vergegenständlichen und *aktive Dokumente* sind ungeeignet, um die kontrollflusssteuernde Rolle des Kernsystems begreiflich zu machen. Mit *Einschüben* und *Einschubrahmen* lassen sich dementgegen alle Bestandteile einer Anwendungsfamilie samt ihren Eigenschaften auf die Gegenstände übertragen, die ein Einschubsystem ausmachen.

   Begriff 5.3 Einschubsystem *(slide-in system)*

   Eine mehrteilig ausgelieferte Anwendungsfamilie ist ein Einschubsystem, wenn deren Auslieferungseinheiten sich gemäß der Systemmetapher *Einschub und Einschubrahmen* sitzungsweise zusammenstellen lassen.

Anwendungsfamilie	**Einschubsystem**
Kernsystem	Kernsystem
Anpassungsstelle	Einschubrahmen
Einfachanpassungsstelle	Einfacheinschubrahmen
Mehrfachanpassungsstelle	Mehrfacheinschubrahmen
Zwangsanpassungsstelle	Zwangseinschubrahmen
Komponente	Einschub
Dienstleisterkomponente	Dienstleistereinschub
Bausteinkomponente	Bausteineinschub

   Da keine Strukturbrüche zwischen den Elementen des Konzepts der Anwendungsfamilie und denen der Systemmetapher *Einschub und Einschubrahmen* bestehen, ist die Metapher uneingeschränkt dazu geeignet, die Vorstellung der Anwender darüber zu leiten, auf welche Weise Mitglieder einer Anwendungsfamilie sitzungsweise zusammenstellbar sind. Einschubsysteme mit ihren Einschubrahmen und Einschüben können aufgrund ihrer Anschaulichkeit ebenso wie Schreibtische mit Werkzeugen, Materialien und Behältern die Grundlage für anwendungsorientierte Arbeitsumgebungen bilden. Während Arbeitsumgebungen, die sich auf Entwurfsmetaphern wie *Werkzeug* und *Material* stützen, die Arbeit mit dem Anwendungssystem erleichtern (siehe Abschnitt 3.2), können sich auf die Systemmetaphern *Einschub* und *Einschubrahmen* stützende Konfigurationsumgebungen die Zusammenstellung des gewünschten Anwendungssystems vereinfachen.

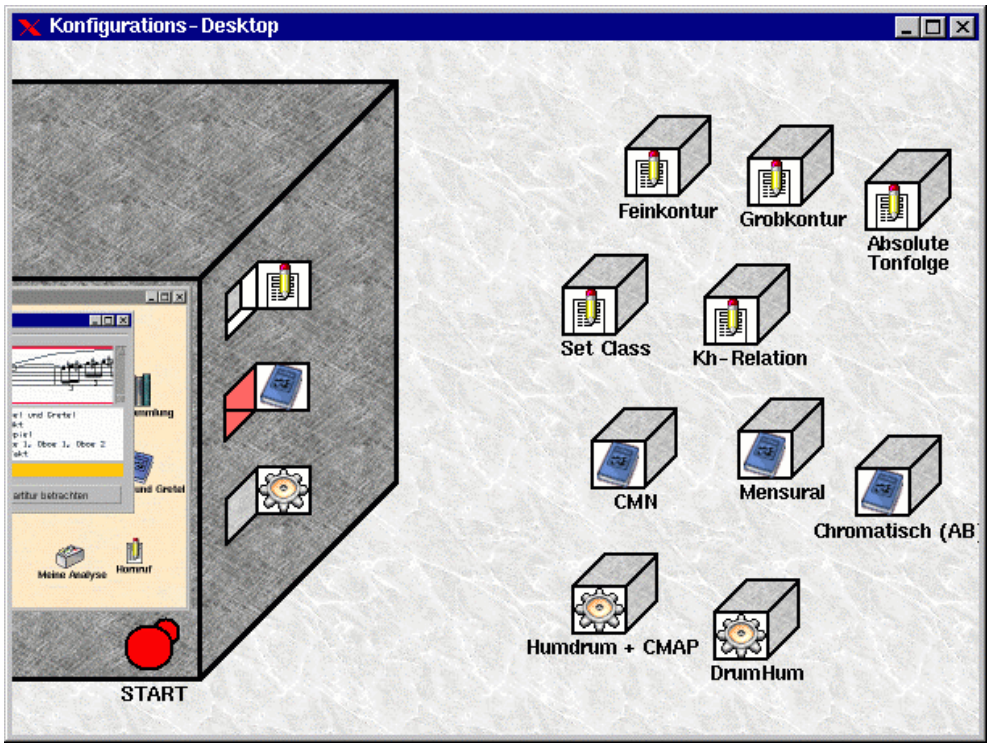


Abb. 5-3 Konfigurationsdesktop des *JRing*-Analysesystems

Auf der linken Seite des Desktops befindet sich das anpassbare Kernsystem, das auf seiner rechten Seite die mit Piktogrammen versehenen Einschubrahmen für Notizbestandteile, Partitur-Engine und MDV-Subsystem enthält. Rechts zu sehen sind die zur Anpassung zur Verfügung stehenden Einschübe, aus deren Piktogrammen ersichtlich wird, in welche Einschubrahmen sie passen.

Der Einschubrahmen für MDV-Subsysteme ist innen rot ausgekleidet, um anzuzeigen, dass es sich um einen Zwangseinschubrahmen handelt. Der Einschubrahmen für Notizbestandteile ist als Mehrfacheinschubrahmen identifizierbar, da eingeschobene Einschübe nicht mit der Oberfläche des Kernsystems abschließen und somit Platz für weitere Einschübe signalisieren. Zusätzlich zu diesen graphischen Unterscheidungsmerkmalen können die weiteren Eigenschaften von Einschüben und Einschubrahmen mit Sondierungswerkzeugen angezeigt werden.

Einschübe können per *drag and drop* (vgl. [Bor99, S. 663f], [Lip99, S. 36] und [WAM98, S. 190f]) in passende Einschubrahmen verbracht werden. Das System kann über den Start-Knopf auf der Frontseite des Kernsystems in Betrieb gesetzt werden, sobald der Zwangseinschubrahmen *MDV-Subsystem* einen Einschub enthält.

2. *Anwendungsorientierung*. Anwendungsorientierte Software zeichnet sich unter anderem dadurch aus, dass sie an die Anforderungen der Anwender anpassbar ist (siehe Begriff 1.1). Wie ich in Kapitel 4 gezeigt habe, lässt sich diese Flexibilität mit herkömmlichen Ansätzen nur bis hin zu einer geeigneten anwendungsorientierten Handhabung und Präsentation zur Laufzeit des Anwendungssystems realisieren. Die sich zur Laufzeit bietenden funktionalen Möglichkeiten des Anwendungssystems lassen sich hingegen mit den in der Literatur diskutierten Ansätzen durch die Anwender nicht beliebig anpassen, da entweder nicht alle möglichen Anpassungen ins Kompilat einbindbar sind oder sich die Konfiguration für Nicht-Techniker als zu komplex erweist.

Mit der Systemmetapher *Einschub und Einschubrahmen* können diese beiden Beschränkungen überwunden und das Prinzip der Anwendungsorientierung somit vollständig umgesetzt werden: Einerseits ist der Anpassungsprozess hier so gestaltet, dass er für die Anwender verständlich ist und somit auf einfache Weise jedes Mal ausgeführt werden kann, wenn sich die fachlichen Anforderungen ändern. Andererseits können Einschübe losgelöst vom Kernsystem entwickelt werden, so dass eine beliebige Anpassung der *Funktionalität* von Anwendungssystemen durch die Anwender ermöglicht wird.

3. *Vermeidung von „fatware“ bei der Auslieferung von Anwendungsfamilien*. Da mit der Systemmetapher *Einschub und Einschubrahmen* verständlich handhabbare Komponenten als Einschübe getrennt vom Kernsystem ausgeliefert und je nach Bedarf erst von den Anwendern zu einem konkreten Anwendungssystem zusammengestellt werden können, ist eine Auslieferung als platzraubende, inflexible „fatware“ nicht mehr notwendig (siehe Abschnitt 4.1).

Die Realisierung aller drei Vorzüge von Einschubsystemen hängt allerdings davon ab, dass sich die Auslieferungseinheiten von Anwendungsfamilien auch tatsächlich als Einschübe und Kernsysteme mit Einschubrahmen modellieren und konstruieren lassen. Diese konstruktive Validierung der bisherigen Ergebnisse in Form des Einschub-Ansatzes ist der Gegenstand des Rests dieses Kapitels. In Abschnitt 5.1 fokussiere ich dabei zunächst auf den Entwurf und zeige, wie das zu Beginn von Kapitel 4 skizzierte allgemeine softwaretechnische Modell einer Anwendungsfamilie verfeinert werden muss, um ein Einschubsystem zu modellieren. Anschließend demonstriere ich in Abschnitt 5.2 am Beispiel des musikwissenschaftlichen Analysesystems *JRing*, wie sich die zur Eingliederung von Einschüben notwendige Laufzeitinfrastruktur konkret in Java implementieren lässt.

5.1 Softwaretechnische Modellierung

Als eine konkrete Ausprägung von Anwendungsfamilien unterliegen Einschubsysteme den Modellierungsrichtlinien, die ich in Abschnitt 4.3 für den Zuschnitt und die Gestaltung von Auslieferungseinheiten einer Anwendungsfamilie diskutiert habe.

1. Das softwaretechnische Modell der Anwendungsfamilie muss so in Auslieferungseinheiten partitioniert werden, dass jeder den Anwendern verständlicher Systembestandteil (Kernsystem oder Komponente) genau einer Auslieferungseinheit entspricht (siehe Abb. 4.3-2 auf S. 102).

2. Die Auslieferungseinheiten der Anwendungsfamilie müssen für die Anwender auf verständliche Weise handhabbar sein.

Mit der Systemmetapher *Einschub und Einschubrahmen* habe ich bereits eine Lösung gefunden, mit der sich die zweite Richtlinie umsetzen lässt. In diesem Abschnitt disku-

tiere daher Konstruktionsansätze, mit denen sich die in Kapitel 4 nur auf allgemeine Anwendungsfamilien bezogene erste Richtlinie auf das softwaretechnische Modell eines Einschubsystems anwenden lässt. Das grundlegende Konstruktionsproblem besteht dabei darin, bereits im softwaretechnischen Modell zu berücksichtigen, dass Kernsystem und Einschübe erst zur Laufzeit in beliebigen Kombinationen zusammengebunden werden. Die Verbindungspunkte müssen daher einerseits flexibel genug sein, um für verschiedene Einschübe geeignet zu sein und andererseits nur diejenigen Kombinationen zulassen, die mit der Systemmetapher *Einschub und Einschubrahmen* vereinbar sind.

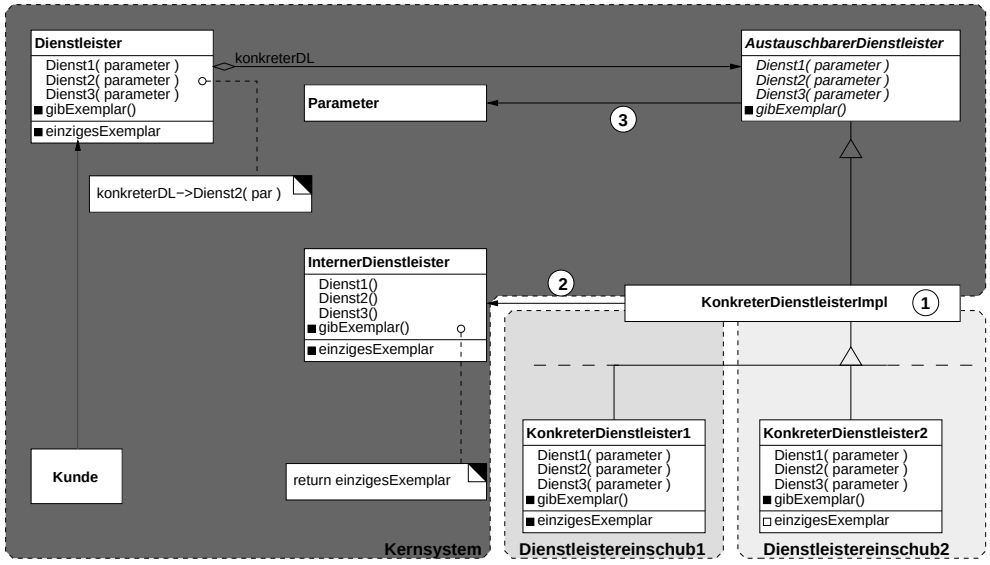


Abb. 5.1-1 Softwaretechnisches Modell eines austauschbaren Dienstleisters (siehe Abb. 4-3 auf S. 88) und dessen Abbildung auf die Auslieferungseinheiten „Kernsystem“ sowie „Dienstleistereinschub1“ und „Dienstleistereinschub2“.

An denjenigen Stellen, an denen Klassen oder die Beziehungen zwischen Klassen sich mit Grenzen von Auslieferungseinheiten kreuzen, muss das in Abb. 5.1-1 dargestellte Modell verfeinert werden, damit die Einschübe von Einschubsystemen gemäß der Motivation von Komponentenansätzen unabhängig vom Kernsystem und voneinander entwickelt und ausgeliefert werden können. Die an diesen Stellen auftreten Konstruktionsfragen sind:

(1) Sollte die Klasse `KonkreterDienstleisterImpl` dem Kernsystem oder den Einschüben zugeordnet werden, oder muss sie auf mehrere Auslieferungseinheiten aufgeteilt werden?

(2) Wie können Einschübe auf kernsysteminterne Dienstleister zugreifen, ohne gleichzeitig anfällig für Änderungen des Kernsystems zu werden?

(3) Auf welche Weise können komplexe anwendungsspezifische Parameter zwischen dem Kernsystem und den Einschüben ausgetauscht werden, ohne die Einschübe von kernsysteminternen Parameterklassen und deren Änderungen abhängig zu machen?

(4) Sofern ein Einschubsystem mehr als einen Einschubrahmen besitzt, stellt sich die Frage, wie Einschübe sich gegenseitig nutzen können, ohne die in Abschnitt 4.2 aufgezeigten komplexen Abhängigkeitsgeflechte bei der Konfiguration von Komponentensystemen zu erzeugen.

Zusätzlich zu diesen allgemeinen Konstruktionsfragen, hinsichtlich derer sich keine Unterschiede zwischen Dienstleister- und Bausteineinschüben ergeben, eröffnen sich noch eine zusätzliche Fragen, die von der Art des jeweiligen Einschubs bzw. des Einschubrahmens abhängt:

(5) Inwieweit muss die technologische Infrastruktur, mit der Einschübe zum Konfigurationszeitpunkt in die Einschubrahmen eingefügt werden können, bereits beim Entwurf des softwaretechnischen Modells berücksichtigt werden? Das heißt

- *für die verschiedenen Arten von Einschubrahmen*: Welche Funktionalität wird an Einschubrahmen benötigt, um gewährleisten zu können, dass auch nur die jeweils passenden Einschübe in der erforderlichen Anzahl eingefügt werden?

- *Für Dienstleistereinschübe*: Wie erhalten Exemplare von `Abstrakter-Dienstleister` Zugriff auf konkrete Dienstleister, die zum Zeitpunkt der Übersetzung noch nicht feststanden?

- *Für Bausteineinschübe*: Wie erhalten Exemplare von `Bausteinfabrik` die Verweise auf die in Einschüben befindlichen Baustein-Baupläne, um ihren Produktvorrat um Bausteine zu erweitern, die bei der Übersetzung noch unbekannt waren?

Jede dieser Konstruktionsfragen diskutiere ich im Folgenden in einem separaten Abschnitt und entwickle dabei durch schrittweise Verfeinerung ein softwaretechnisches Modell eines Einschubsystems, für das ich in Abschnitt 5.2 am Beispiel *JRings* demonstriere, wie dieses sich in Java konkret implementieren lässt.

5.1.1 Einschübe

In einem monolithisch entwickelten und ausgelieferten musikwissenschaftlichen Analysesystem kann es vorteilhaft sein, beispielsweise verschiedene austauschbare Notizbestandteil-Bausteine als Spezialisierungen einer gemeinsamen Oberklasse `NotizbestandteilImpl` zu modellieren, um ähnliche Funktionalität nicht redundant und fehleranfällig an zwei getrennten Stellen zu duplizieren. In einem mehrteilig ausgelieferten und potentiell durch mehrere Hersteller entwickeltem Einschubsystem stellt sich die Frage, welcher Auslieferungseinheit `NotizbestandteilImpl` am besten zugeordnet werden sollte bzw. ob sich eine gemeinsame Implementation überhaupt weiterhin vertreten lässt.

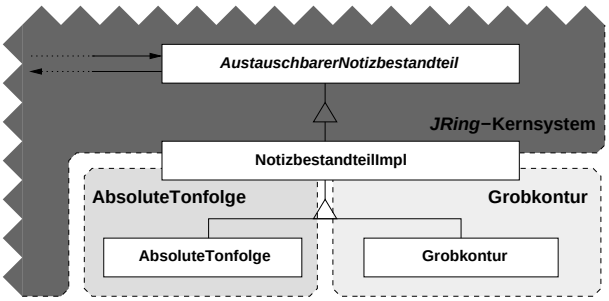
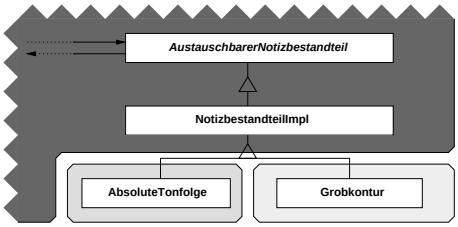
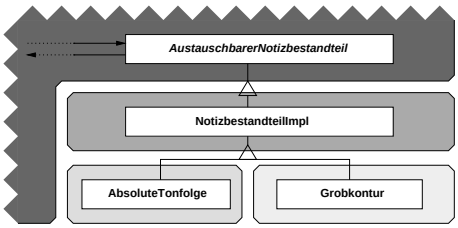


Abb. 5.1-2 Die Klasse `NotizbestandteilImpl` im Spannungsfeld der drei Auslieferungseinheiten *JRing-Kernsystem*, Einschub *AbsoluteTonfolge* und Einschub *Grobkontur*.

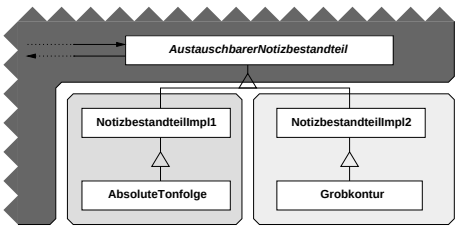
Da eine einzelne Klasse nicht auf mehrere Auslieferungseinheiten aufteilbar ist, ergeben sich drei Modellierungsmöglichkeiten mit unterschiedlichen *Vorteilen*:



(1) *`NotizbestandteilImpl` als Bestandteil des Kernsystems*. Hierbei können beide Einschübe wie bei einer monolithischen Auslieferung eine gemeinsame Implementation teilen. Da beispielsweise sowohl die in *AbsoluteTonfolge* als auch die in *Grobkontur* verkörperte musikwissenschaftliche Abstraktion sich auf die Melodie bezieht, kann melodiebezogene Funktionalität zentral in `NotizbestandteilImpl` implementiert werden, was zu kleineren, schlankeren Einschüben führt.



(2) *`NotizbestandteilImpl` als separate Auslieferungseinheit*. Notizbestandteile lassen sich nach der in ihnen verkörperten Funktionalität in verschiedene Gruppen einteilen, die unterschiedliche Eigenschaften und somit auch Elemente ihrer Implementation teilen. Eine Möglichkeit bestünde darin, gemeinsame Implementationsklassen gemäß der Art der enthaltenen musikwissenschaftlichen Abstraktion zu modellieren (melodisch – wie in (1) –, harmonisch oder rhythmisch). Eine andere Option wäre es, nach der Art der Präsentation des Notizbestandteils zu unterteilen und die gemeinsame Darstellungsfunktionalität auszulagern (graphisch oder textuell). Der Vorteil verschiedener spezialisierter `NotizbestandteilImpl`-Klassen als separate Auslieferungseinheiten zusätzlich zum Kernsystem und zu den Einschüben besteht darin, dass keine eventuell nicht benötigte Funktionalität im Kernsystem vorgehalten werden muss und die Einschübe trotzdem klein und schlank sein können.



(3) *Separate `NotizbestandteilImpl`-Klassen in jedem Einschub*. Hierbei wird kein Teil der Implementation zwischen Einschüben per Implementationsvererbung geteilt. Die einzige Vererbungsbeziehung besteht in der Übernahme der verhaltensbestimmenden Schnittstelle `AustauschbarerNotizbestandteil` aus dem Kernsystem. Die Funktiona-

lität der Einschübe kann aber trotzdem schlank implementiert werden, wenn im Einschubrahmen des Kernsystems geeignete interne Dienstleister bereitstehen, die den Einschüben die benötigte Funktionalität über Benutzt-Beziehungen zur Verfügung stellen.

Trotz ihrer Vorteile sind die Lösungen (1) und (2) problematisch, da sie das Ausdrucksmittel der Implementationsvererbung über Systembestandteilgrenzen hinweg verwenden, was folgende *Nachteile* mit sich bringt:

- Um die Implementation einer Oberklasse (1) aus dem Kernsystem oder (2) aus einer anderen Auslieferungseinheit erben zu können, muss deren Implementation vollkommen einsehbar sein. Black-Box-Auslieferungseinheiten und die damit verbundenen Modellierungs- und Sicherheitsvorteile lassen sich daher nicht realisieren (siehe Abschnitt 4.1).

- Implementationsvererbung über Systembestandteilgrenzen hinweg eröffnet das Problem der änderungsempfindlichen Oberklasse (engl. *fragile base class problem*; vgl. [Szy98, S. 102ff] und [Pre97, S. 9]). Dabei können Schwierigkeiten auftreten, wenn die Entwickler von Unterklassen nicht lediglich die Schnittstelle, sondern auch die Implementation der Oberklasse heranziehen, um deren Verhalten zu ermitteln. Durch diese Abhängigkeit von der internen Implementation anstatt von der öffentlichen Schnittstelle kann jede Implementationsänderung der Oberklasse auch bei gleichbleibender öffentlicher Schnittstelle zu unvorhersehbaren Verhaltensänderungen der Unterklasse führen.[2]

- Da Implementationsvererbung sich nicht der öffentlichen Schnittstelle bedient, um Funktionalität einer anderen Auslieferungseinheit zu nutzen, sondern unmittelbar auf deren interne Implementation zugreift, widerspricht sie der Systemmetapher *Einschub und Einschubrahmen*, in der geschlossene Konstruktionseinheiten sich ausschließlich über ihre Schnittstellen gegenseitig nutzen.

Darüber hinaus bringt Variante (2) noch weitere Schwierigkeiten mit sich, da sie Hilfskomponenten einführt, die jeweils eine verschiedene `NotizbestandteilImpl`-Klasse enthalten. Dadurch entsteht neben Kernsystem und Einschub noch eine dritte Art von Auslieferungseinheiten.

- Aus Sicht der *Handhabung & Präsentation* sind Hilfskomponenten problematisch, da sie rein technisch motiviert und somit für die Anwender nicht verständlich sind (siehe Abschnitt 4.3).

- Aus technischer Sicht sind sie ungünstig, weil sie sich wie unter (2) gezeigt nach zahlreichen verschiedenen Gesichtspunkten zuschneiden lassen und komplexe funktionale Überlappungen zwischen Produkten verschiedener Hersteller somit wahrscheinlich werden. Die durch den Einschub-Ansatz vermiedenen Probleme der

[2] Szyperski arbeitet in [Szy98, S. 103] heraus, dass es zwei Varianten des Problems der änderungsempfindlichen Oberklasse gibt. Neben der hier diskutierten *semantischen* Variante, bei der es um Verhaltensveränderungen geht, existiert auch eine *syntaktische* Variante, die zu Binärinkompatibilitäten bei umgestellten Vererbungshierarchien führt und die sich mit rein technologischen Mitteln lösen lässt (vgl. [CDFR95]).

komplexen Abhängigkeitsgeflechte zwischen Laufzeitkomponenten würden durch Hilfskomponenten wieder auftreten (siehe Abschnitt 4.2).

Aufgrund dieser Nachteile der auf Implementationsvererbung beruhenden Konstruktionsvarianten (1) und (2) empfiehlt es sich, die Funktionalität von Einschüben wie in (3) geschildert vollkommen gekapselt in jedem Einschub separat zu implementieren und Funktionalität des Kernsystems ausschließlich über systeminterne Dienstleister zu nutzen.

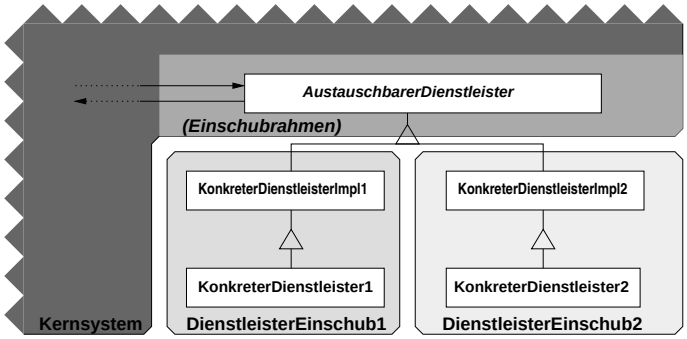


Abb. 5.1-3 Unproblematische Modellierung von in mehreren Einschüben genutzter Funktionalität.

5.1.2 Systeminterne Dienstleister

Wie ich in Kapitel 2 gezeigt habe, sind Dienstleister unabhängig vom Einschub-Ansatz ein gebräuchliches Mittel, um mehrfach im System benötigte Funktionalität nicht redundant implementieren zu müssen. In Verbindung mit Anwendungsfamilien können diese Dienste als interne Dienstleister auch den austauschbaren Komponenten zur Verfügung gestellt werden (siehe Begriff 2.3 auf S. 20). Zum Beispiel rechnet der interne Dienstleister `KoordinatenKonvertierer` eines Simulations-Einschubsystems geräteunabhängige graphische Repräsentationen von Simulationselementen in konkrete Viewport-Koordinaten um. Seine Dienste sind sowohl zur Darstellung von in Einschüben realisierten Simulationsbausteinen als auch zu deren im Kernsystem festgelegter Verdrahtungen nützlich. Klienten von `KoordinatenKonvertierer` befinden sich daher im Kernsystem und auch in Einschüben.

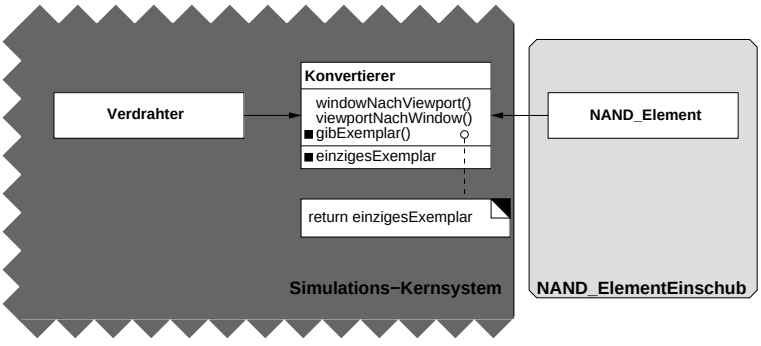


Abb. 5.1-4 Der internen Dienstleister `Koordinaten-Konvertierer` eines Simulations-Einschubsystems und dessen Klienten `Verdrahter` (kernsystemintern) und `NAND_Element` (Einschub).

Für Einschubsysteme ergibt sich aus der Tatsache, dass das Kernsystem und die Einschübe von getrennten Entwicklergruppen erstellt und ausgeliefert werden können, das folgende Modellierungsproblem, bei dem Flexibilität für die Kernsystementwickler gegen Stabilität für die Einschubentwickler abgewogen werden muss:

- *Flexibilität für Kernsystementwickler*. Unabhängig von funktionalen Änderungen sollen die Dienstleister innerhalb des Kernsystems nach rein technischen Gesichtspunkten beliebig modifizierbar sein. Wenn es notwendig erscheint, sollen Dienstleister umbenannt, in andere Teile des Kernsystems verschoben oder aufgrund anderer Rahmenwerke oder Bibliotheken realisiert werden können.

- *Stabilität für die Einschubentwickler*. Einschubentwickler sollen vor diesen Änderungen abgeschirmt werden, so dass sie nicht gezwungen sind, ihre Einschübe aus rein technischen Gründen neu zu übersetzen und auszuliefern, obwohl die Funktionalität der internen Dienstleister unverändert geblieben ist.

Die konstruktive Lösung dieses Problems besteht darin, Einschubrahmen explizit zu modellieren und als Schnittstelle des Kernsystems gegenüber den Einschüben auszugestalten. Das heißt, ein Einschubrahmen macht die Funktionalität der verfügbaren internen Dienstleister zwar nutzbar, verbirgt aber deren Implementation. Einschubentwickler müssen ihre Einschübe daher nur dann neu übersetzen und ausliefern, wenn die internen Dienstleister sich funktional ändern. Vor rein technischen Änderungen werden sie durch den Einschubrahmen abgeschirmt.

Als Ausgangspunkt für eine technische Realisierung bietet sich das Proxy-Entwurfsmuster in der Form des Schutz-Proxies an (vgl. [GHJV98, S. 227ff]). Dabei wenden sich Klienten einer abzuschirmenden Klasse nicht mehr direkt an diese, sondern an einen Stellvertreter mit identischer Schnittstelle. Dieser leitet alle Anfragen an die abzuschirmende Klasse weiter und übermittelt die Ergebnisse an die Klienten zurück, ohne dass diese ersehen können, wer den Dienst tatsächlich erbracht hat.

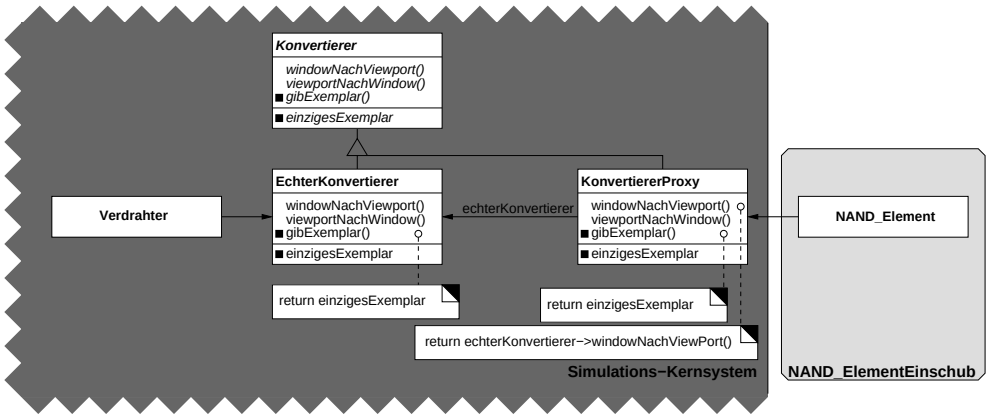


Abb. 5.1-5 Der interne Dienstleister `KoordinatenKonvertierer` verborgen hinter dem Schutz-Proxy `KoordinatenKonvertiererProxy` des Einschubrahmens.

Während der interne Klient `Verdrahter` weiterhin unmittelbar auf den Dienstleister zugreift und dessen Modifikationen nachvollziehen muss, ist der Bausteineinschub `NAND_Element` vor technischen Änderungen abgeschirmt.

Das Proxy-Entwurfsmuster ist nicht speziell auf Dienstleister ausgelegt und kann – wie ich im nächsten Abschnitt zeige – verwendet werden, um eine Vielzahl von Stellvertreterarten zu modellieren. Die Standardform des Entwurfsmusters, bei der jede Operation von `Proxy` die gleichnamige Operation von `RealXY` aufruft, muss daher in jedem konkreten Anwendungsfall daraufhin überprüft werden, ob sich individuelle Verbesserungen realisieren lassen.

Im Fall der als Singleton modellierten internen Dienstleister ist es nicht notwendig, für jede Operation von `Proxy` eine Umleitung auf `RealXY` zu implementieren. Statt dessen reicht es aus, wenn die Operation `gibExemplar()` von `Proxy` anstelle eines Verweises auf die eigene Variable `einzigesExemplar` die Operation `gibExemplar()` von `RealXY` aufruft. Auf diese Weise werden alle Aufrufe an `Proxy` automatisch durch `RealXY` ausgeführt, ohne dass dies für die Klienten ersichtlich wird.

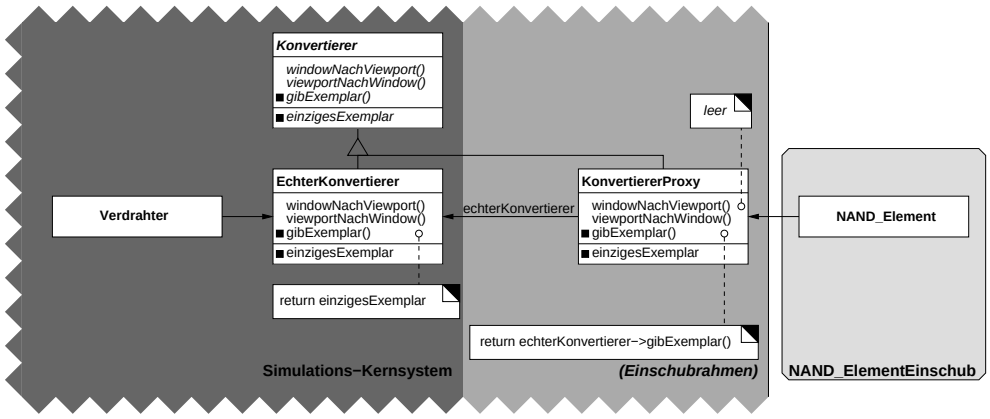


Abb. 5.1-6 Modellierung von internen Dienstleistern mit minimalem Implementationsaufwand.

5.1.3 Parameterklassen

Parameterklassen sind in Anwendungsfamilien notwendig, wenn Werte oder Exemplare eingebauter Datentypen (beispielsweise `bool`, `int`, `float` in C++ oder `java.util.LinkedList` in Java) nicht ausreichen, um Operationen über die Grenze zwischen Kernsystem und austauschbaren Komponenten hinweg aufzurufen. Die Partitur-Engine eines musikwissenschaftlichen Analysesystems benötigt beispielsweise für ihre Operation `gibDarstellung()` die Parameterexemplare des Fachwerttyps `fwPartiturposition` und liefert als Ergebnis eine Liste von `PartiturObjekten` zurück.[3]

[3] Den umgekehrten Fall, in dem Komponenten auf interne Dienstleister des Kernsystems zugreifen, diskutiere ich am Ende dieses Abschnitts.

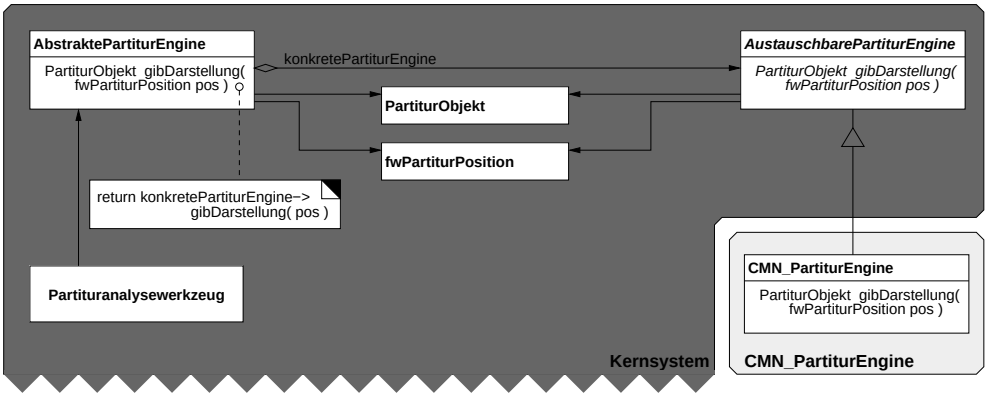


Abb. 5.1-7 Die von der Komponente `PartiturEngine` benötigten Parameterklassen `fwPartiturposition` und `PartiturObjekt`.

Im Gegensatz zu monolithisch ausgelieferten Anwendungsfamilien ergibt sich bei Einschubsystemen das Entwurfsproblem, dass die Parameterklassen sowohl im Kernsystem als auch in den separat entwickelten und ausgelieferten Einschüben bekannt sein müssen, damit Kernsystem und Einschübe Parameterexemplare erzeugen und gegenseitig austauschen können. Wie bei der Modellierung interner Dienstleister sollen Einschübe dabei aber vor rein technischen, kernsysteminternen Modifikationen abgeschirmt werden, solange sich die Schnittstelle der Parameterklasse nicht ändert. Aufgrund der ähnlichen Problemlage bietet sich die bereits in Abschnitt 5.1.2 für interne Dienstleister gefundene Lösung mit Proxies als Ausgangsbasis an. Am Beispiel des Einschubs `PartiturEngine` heißt dies, dass an seiner Schnittstelle nicht Parameter des kernsysteminternen Typs `fwPartiturposition`, sondern des im Einschubrahmen bekanntgemachten Typs `fwPartiturpositionProxy` erwartet werden. Umgekehrt liefert `gibDarstellung()` eine Liste von Exemplaren des Typs `PartiturObjektProxy` zurück.

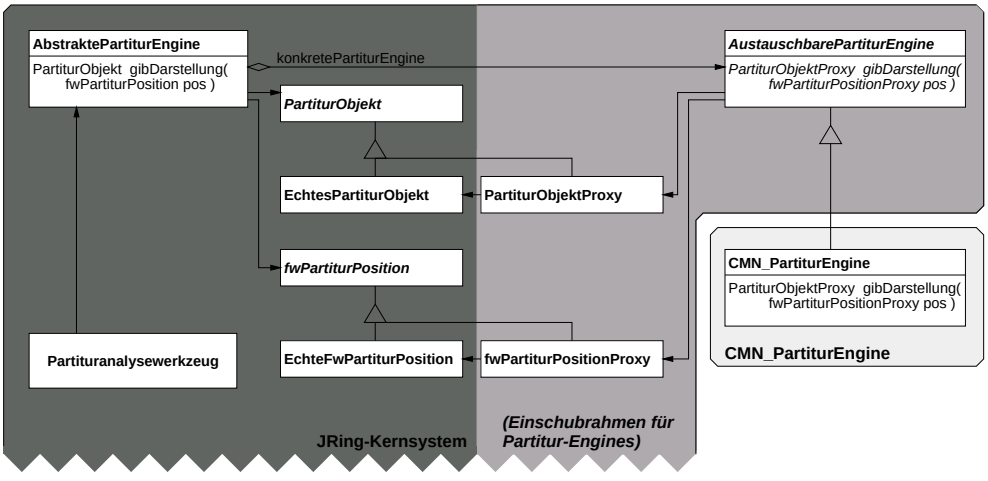


Abb. 5.1-8 Die Parameterklassen `fwPartiturposition` und `PartiturObjekt` mit ihren Proxies im Einschubrahmen.

Diese rein Proxy-Lösung reicht aber im Gegensatz zur Modellierung interner Dienstleister nicht aus, da es sich um einen Kommunikationsprozess zwischen Auslieferungseinheiten handelt. Wenn auf Seiten des Kernsystems Parameter des Typs `EchterXY` und auf Seiten der Einschübe Parameter des Typs `XYProxy` verwendet werden, dann kann kein Austausch stattfinden. Die Parameter müssen daher kernsystemintern in die im

Einschubrahmen deklarierte Form umgewandelt werden, bevor Operationen über die Grenze zwischen Kernsystem und Einschüben hinweg aufgerufen werden können. Dies betrifft alle auslieferungseinheitenüberschreitenden Operationsaufrufe:

(1) *Aufrufe von Operationen von Einschüben*. Für nach dem Brücken-Entwurfsmuster modellierte Dienstleistereinschübe (siehe Kapitel 4) lässt sich die Typumwandlung in die Brücke zwischen Abstraktion und Implementor integrieren. Da der Implementor bereits die Schnittstelle für den Einschub vorgibt und somit `ParameterProxy` verwendet, muss die Umwandlung innerhalb der Abstraktion erfolgen. Alle `Dienst`-Operationen von `AbstrakterDienstleister` erzeugen dazu für jedes Exemplar des Typs `Parameter` ein neues Exemplar des Typs `ParameterProxy`. Dies kann beispielsweise über einen Copy-Konstruktor von `ParameterProxy` geschehen.

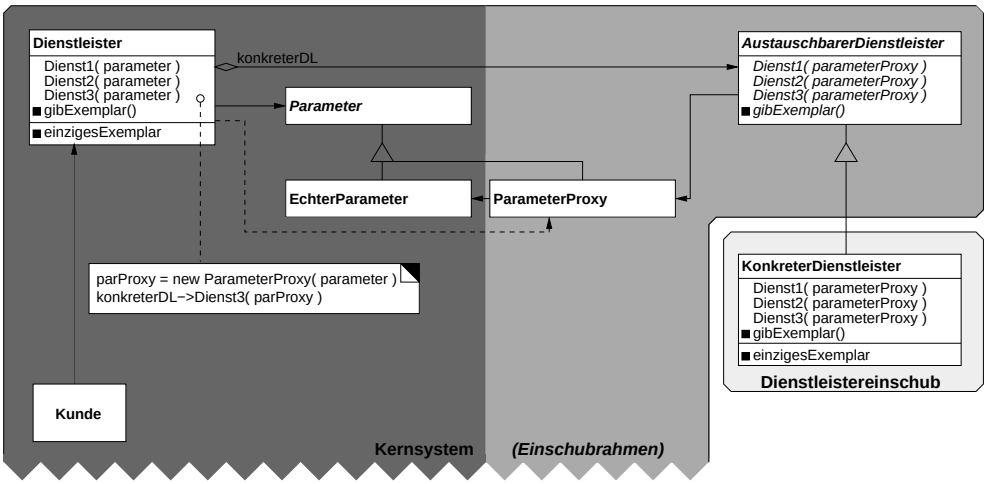


Abb. 5.1-9 Typumwandlung zum Aufruf von Operationen von Dienstleistereinschüben.

Da Bausteineinschübe direkt von individuellen kernsysteminternen Klienten aufgerufen werden, kommt eine zentrale Umwandlung wie durch die Abstraktion eines Brücken-Musters nicht in Frage. Stattdessen muss die Umwandlung hier für jeden Aufruf einzeln erfolgen.

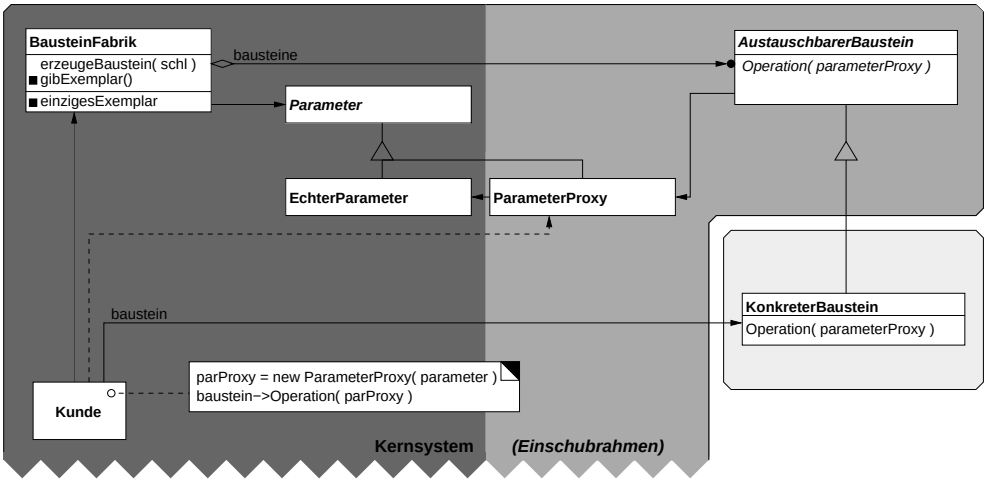


Abb. 5.1-10 Typumwandlung zum Aufruf von Operationen von Bausteineinschüben.

(2) *Aufrufe von Operationen von internen Dienstleistern*. Da die Klasse des Parameter-Proxies stets eine Unterklasse des Typs des kernsystemintern verwendeten Parameters ist, müssen interne Dienstleister, deren Operationen diese Parameter benötigen, nicht modifiziert werden, da diese sichere polymorphe Typumwandlung vom Sub- zum Supertyp in jeder objektorientierten Sprache automatisch durchgeführt wird.

Damit habe ich Lösungen für alle drei zu Beginn von Abschnitt 5.1 aufgeworfenen Probleme aufgezeigt, die sich ergeben, wenn Bezüge zwischen Klassen im Kernsystem und den Einschüben bestehen. Im folgenden Abschnitt 5.1.4 zeige ich nun zunächst, wie sich Einschübe in verschiedenen Einschubrahmen gegenseitig nutzen können und in den Abschnitten 5.1.5 und 5.1.6 diskutiere ich, wie das bisher entwickelte Modell eines Einschubsystems erweitert werden muss, damit Einschübe zum Konfigurationszeitpunkt tatsächlich dynamisch in die Einschubrahmen des laufenden Kernsystems eingefügt werden können.

5.1.4 Unabhängigkeit der Einschubrahmen

In Kapitel 4 habe ich dargelegt, dass eine Auslieferung einer Anwendungsfamilie ausschließlich in ungefähr gleichgroßen Komponenten zwar potentiell eine maximale Wiederverwendung erlaubt, aber andererseits ein komplexes Abhängigkeitsgeflecht zwischen den einzelnen Komponenten erzeugt, das die gewünschte Unabhängigkeit der einzelnen Komponenten praktisch verhindert (siehe S. 99).

Mit einer Kombination der bereits vorgestellten Modellierungstechniken ist es im Einschub-Ansatz möglich, dass sich Einschübe zwar gegenseitig nutzen, ohne jedoch von einer bestimmten Konfiguration anderer Einschübe abhängig zu sein oder unmittelbar miteinander in Verbindung treten zu müssen. Beispielsweise können Notizbestandteil-Einschübe damit zur Ausführung musikwissenschaftlicher Operationen auf eines MDV-System-Einschubs zurückgreifen, ohne diese eigenständig implementieren zu müssen.

Jeder Einschub hat innerhalb des Kernsystems einen dazugehörigen Dienstleister, über den kernsysteminterne Klienten auf den Einschub bzw. die Einschübe innerhalb des jeweiligen Einschubrahmens zugreifen können. Bei Dienstleistereinschüben handelt es sich um den Abstraktionsteil des Brückenmuster, bei Bausteineinschüben um eine Bausteinfabrik (siehe Abb. 4-3 auf S. 88 und Abb. 4-5 auf S. 89). Einschübe können sich nun auf einfache Weise gegenseitig nutzen, indem diese Dienstleister als interne Dienstleister in anderen Einschubrahmen bekanntgemacht werden. Auf diese Weise wird durch die Entkopplung und abschirmende Wirkung des internen Dienstleisters einerseits verhindert, dass ein Einschub explizit von einem anderen abhängig wird und andererseits doch die Möglichkeit eröffnet, dass die Funktionalität anderer Einschübe genutzt wird. Die einzige Voraussetzung für die gegenseitige Nutzung ist, dass der genutzte Einschubrahmen ein Zwangseinschubrahmen ist, damit die verwendete Funktionalität auch garantiert verfügbar ist.

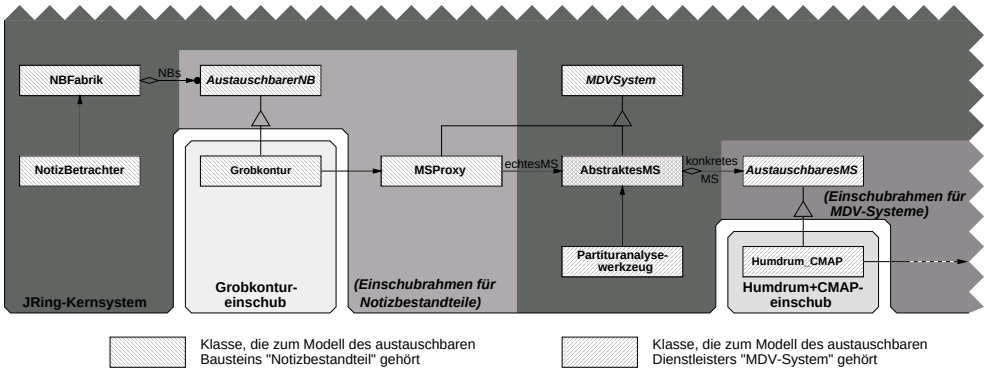


Abb. 5.1-11 Gegenseitige Nutzung von zwei voneinander unabhängigen Einschüben am Beispiel eines Notizbestandteil-Einschubs, der einen MDV-System-Einschub nutzt.

Der Dienstleister `AbstraktesMDVSystem` dient als Verbindungsglied zwischen den beiden Einschubrahmen und fungiert als interner Dienstleister für den Notizbestandteil-Einschub `Grobkontur`.

5.1.5 Eingliederungsinfrastruktur

In den vorangegangenen Abschnitten habe ich Einschubrahmen bisher nur bezüglich derjenigen Aspekte modelliert, die sie als besondere Stellen des Kernsystems charakterisieren, an denen interne Dienstleister und Parameterklassen für die Einschübe sichtbar und nutzbar sind. Dagegen steht eine Diskussion derjenigen Aspekte, die sich auf den eigentlichen Vorgang der Eingliederung beziehen, noch aus. Die Ergebnisse der Abschnitte 5.1.1 bis 5.1.4 genügen damit nicht, um zu gewährleisten, dass beispielsweise der Einschubrahmen „*Notizbestandteile*“ eines Einschubsystems zur musikwissenschaftlichen Analyse auch tatsächlich nur Notizbestandteil-Einschübe aufnehmen kann und dass diese Notizbestandteile als Produkte der kernsysteminternen Notizbausteinfabrik zur Verfügung stehen.

Erst wenn auch sämtliche die auf die Eingliederung bezogenen Aspekte von Einschubrahmen (siehe Begriff 5.1 und 5.2) auf geeignete Weise im softwaretechnischen Modell von Einschubsystemen realisiert werden können, kann die Systemmetapher *Einschub und Einschubrahmen* vollkommen umgesetzt werden. Die daraus resultierenden Anforderungen beziehen sich auf (1) die Passform von Einschüben, (2) die funktionale Eingliederung von Einschüben und (3) Zwangseinschubrahmen. Mit Ausdrucksmitteln des softwaretechnischen Modells lassen sie sich wie folgt interpretieren:

(1) *Einschubrahmen können nur passende Einschübe aufnehmen*. Die Operation des Einschubrahmens zum Aufnehmen von Einschüben muss diejenigen Einschübe zurückweisen, die kein Subtyp von `AustauschbarerDienstleister` bzw. `AustauschbarerBaustein` sind.

    Am Beispiel des Einschubrahmens *„Notizbestandteil“* muss also der Einschub `GrobKontur` eine Unterklasse von `Notizbestandteil` sein, damit er eingefügt werden kann.

```
class EinschubrahmenFürNotizbestandteile
 {
  variable eingefügte Einschübe
  variable Klasse Notizbestandteil

  operation einfügen( einschubKlasse )
   {
    if( einschubKlasse ist Subtyp von Notizbestandteil )
     {
      füge einschubKlasse den eingefügten Einschüben hinzu
     }
   }
 }
```

(2) *Mit dem Hochfahren des Einschubsystems am Ende der Konfigurationszeit müssen die technischen Verbindungen zwischen dem Kernsystem und den Einschüben hergestellt werden*. Dazu müssen die Dienstleister innerhalb des Kernsystems, über die Klienten auf die von Exemplaren der Einschübe zur Verfügung gestellte Funktionalität zugreifen können (Bausteinfabriken bei Bausteineinschüben und abstrakte Dienstleister bei Dienstleistereinschüben), Referenzen auf die Klassen der Einschübe erhalten. Als Oberbegriff für diese Dienstleister verwende ich im Folgenden den Begriff *Einschubexemplaranbieter*.

Begriff 5.4 Einschubexemplaranbieter (***slide-in instance provider)***

Einschubexemplaranbieter sind interne Stellen des Kernsystems eines Einschubsystems, über die kernsysteminterne Klienten zur Laufzeit Zugang zu der von Exemplaren von Einschüben erbrachten Funktionalität erhalten.

Für Dienstleistereinschübe sind Einschubexemplaranbieter abstrakte Dienstleister. Für Bausteineinschübe sind es Bausteinfabriken.

Um Einschubexemplaranbietern die von ihnen benötigten Einschübe zuzuführen, müssen die Einschubrahmen als Vermittler agieren:

- Zu *Einschüben* stehen Einschubrahmen im bereits unter (1) dargestellten Verhältnis der Aggregators. Das heißt, ein Einschubrahmen kann – je nach Art – entweder einen oder mehrere Einschübe enthalten

- Zu *Einschubexemplaranbietern* verhalten sich Einschubrahmen wie Subjekte zu einem Beobachter im Sinne des Beobachter-Entwurfsmusters (vgl. [GHJV98, S. 257ff]). Einschubexemplaranbieter melden sich als Beobachter bei Einschubrahmen an und werden über ihre Operationen `setzeKonkretenDienstleister()` bzw. `fügeProduktHinzu()` mit Einschubklassen versorgt, sobald diese in den Einschubrahmen eingefügt werden.

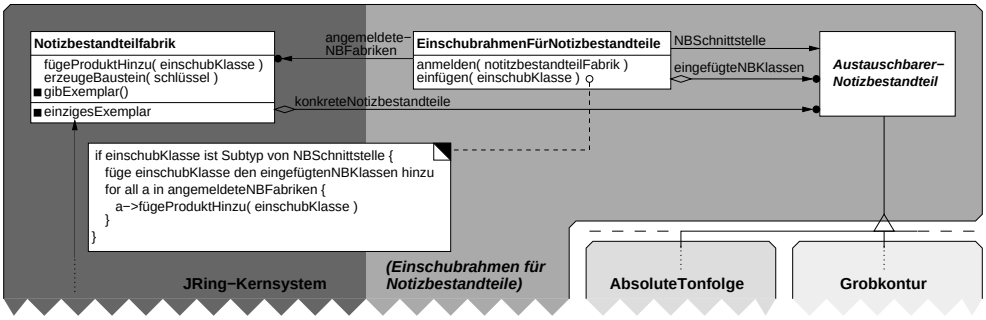


Abb. 5.1-12 Die Eingliederungsinfrastruktur des Einschubrahmens *„Notizbestandteile“* eines Einschubsystems zur musikwissenschaftlichen Analyse.

(3) *Zwangseinschubrahmen müssen am Ende der Konfigurationszeit einen Einschub enthalten, damit das Einschubsystem gestartet werden kann*. Bereits in Kapitel 2 habe ich dargelegt, dass Zwangsanpassungsstellen spezielle Einfachanpassungsstellen einer Anwendungsfamilie sind, die mit exakt einer Komponente angepasst werden müssen. Zusammen mit dem Konzept des Einschubexemplaranbieters ist es möglich, Zwangseinschubrahmen nicht als einen eigenen Typ von Einschubrahmen zu modellieren, sondern als normale Einfacheinschubrahmen aufzufassen. Ein Einfacheinschubrahmen wird genau dann zu einem Zwangseinschubrahmen, wenn mindestens ein Einschubexemplaranbieter bei seiner Anmeldung mitteilt, dass er unbedingt einen Einschub benötigt, um das Kernsystem funktionstüchtig zu machen. Beim Einschubrahmen *„Partitur-Engine“* ist dies beispielsweise der Einschubexemplaranbieter, über den sowohl Partituranalysewerkzeuge als auch Notizbetrachter und -editoren ihre Layoutdaten beziehen.

Damit am Ende der Konfiguration feststellbar ist, ob das Einschubsystem gestartet werden kann oder nicht, müssen Einfacheinschubrahmen eine sondierende Operation `istGefüllt()` besitzen.

```
operation istGefüllt()
 {
  if( eingefügte Einschübe vorhanden ) return true
  else return false
 }
```

Im softwaretechnischen Modell eines Einschubsystems sind Einschubrahmen damit auf zweierlei Weise repräsentiert:

- Zum einen als derjenige Bereich des Kernsystems, in dem sich die Proxy-Klassen befinden, über die Einschübe auf interne Dienstleister und die zum Datenaustausch mit dem Kernsystem notwendigen Parameter zugreifen können.

- Zum anderen durch eine explizite Klasse `Einfacheinschubrahmen` oder `Mehrfacheinschubrahmen`, welche die Funktionalität zur Verfügung stellt, die von demjenigen Subsystem eines Einschubsystems genutzt wird, das für die technische Eingliederung von Einschüben zuständig ist.

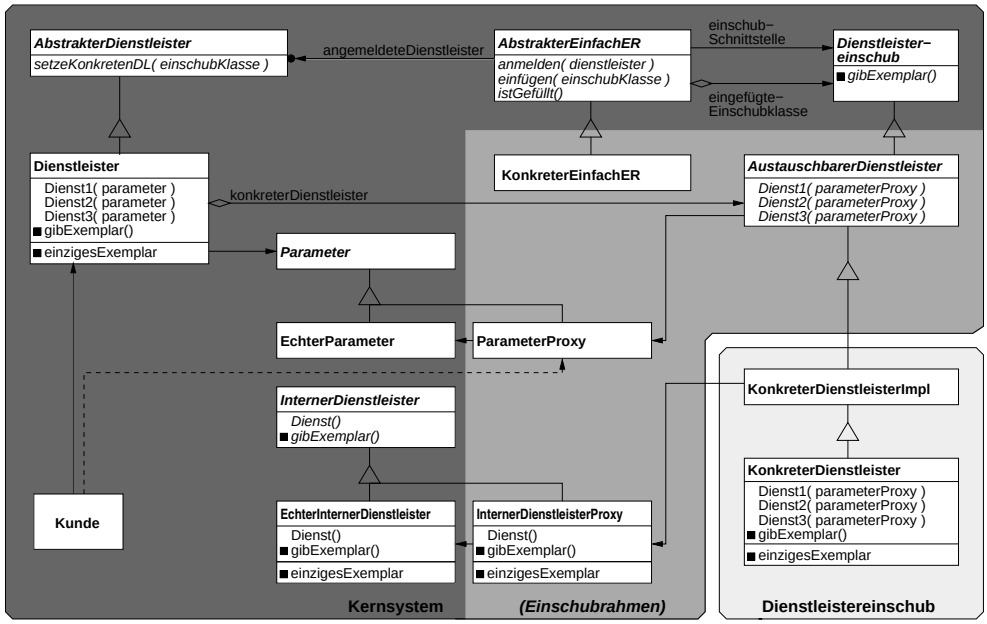


Abb. 5.1-13 Softwaretechnisches Modell eines Einfacheinschubrahmens für Dienstleistereinschübe.

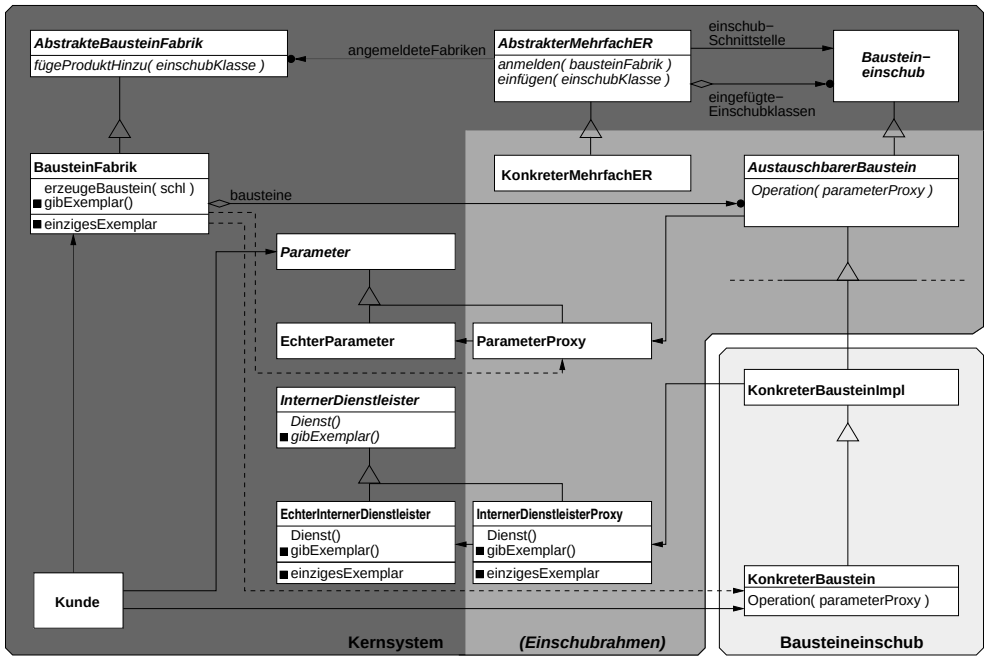


Abb. 5.1-14 Softwaretechnisches Modell eines Mehrfacheinschubrahmens für Bausteineinschübe.

Mit diesen beiden Entwurfsmodellen von Einschubrahmen eines Einschubsystems habe ich gezeigt, wie sich die Systemmetapher von *Einschub und Einschubrahmen* konkret auf das softwaretechnische Modell einer Anwendungsfamilie beziehen lässt. Im folgenden Abschnitt 5.2 komplettiere ich die Beschreibung des Einschub-Ansatzes, indem ich am Beispiel einer Implementation in Java mit Hilfe von *JWAM* demonstriere, wie sich das Modell konstruktiv in einer allgemein verfügbaren Technologie umsetzen lässt.

5.2 Technische Realisierung in Java mit *JWAM*

Bereits in Abschnitt 4.3 habe ich dargelegt, dass die Komponententechnologien CORBA und COM aus zwei Gründen nicht ideal sind, um damit Anwendungsfamilien zu implementieren:

(1) Konzeptionell folgen beide Technologien in weiten Teilen der Systemmetapher *Client und Server*, von der ich gezeigt habe, dass sie zur Veranschaulichung der Bestandteile von Anwendungsfamilien nur bedingt geeignet ist.

(2) Technisch sind sie primär auf die Unterstützung des objektorientierten Operationsfernaufrufs (CORBA) bzw. der sprachübergreifenden Nutzung von Funktionalität ausgerichtet (COM). Der für Anwendungsfamilien wichtige Mechanismus zum dynamischen Einbinden von Laufzeitkomponenten ist daher in vielen Teilen nicht auf die Anforderungen von Anwendungsfamilien ausgerichtet.

Die Programmiersprache Java (siehe Abschnitt 3.3 und in detaillierter Form Anhang B.4) bietet sich dementgegen zur Implementation von Anwendungsfamilien und somit auch von Einschubsystemen an, weil es sich nicht wie bei CORBA und COM um eine Spezialtechnologie, sondern um eine neutrale objektorientierte Programmiersprache samt Laufzeitumgebung handelt. Insbesondere kann dadurch der Mechanismus zum dynamischen Einbinden von Laufzeitkomponenten individuell flexibel an die gewählte Systemmetapher angepasst werden. Obwohl auch jedes Betriebssystem einen Mechanismus zum dynamischen Binden von Laufzeitkomponenten in Form von dynamischen Bibliotheken zur Verfügung stellt (siehe Abschnitt 4.2), empfiehlt sich Java dadurch, dass der Mechanismus nicht für jede Plattform neu entwickelt werden muss, sondern plattformübergreifend einheitlich ist, so dass in Java implementierte Einschubsysteme auf praktisch allen Plattformen auf die gleiche Weise lauffähig sind. Darüber hinaus erfüllt Java anders als CORBA und COM alle fünf in Abschnitt 4.3 aufgestellten Kriterien an Laufzeitkomponenten-Technologien, die für eine bruchlose Umsetzung von Anwendungsfamilien notwendig sind (siehe graue Kästen auf S. 107 und S. 111):

	CORBA	COM	Java
Laufzeitkomponenten	**+**	**+**	**+** Mittels *Java Archives* (JARs)
Black Boxes	**-**	**-**	**+** Durch dementsprechende Vergabe von Berechtigungen
Herstellerabhängige Rechtevergabe	**-**	**+**	**+** Abhängig von zertifizierter, digital signierter Personenangabe
Statische Typisierung	**+**	**-**	**+** Ausnahmslos
Minimale Zeitverzögerungen	**-**	**+**	**+** Aufgrund lokaler Operationsaufrufe innerhalb nur eines Prozessraums

In einer ersten Annäherung lässt sich das in Abschnitt 5.1 erarbeitete Modell eines Einschubsystems wie folgt umsetzen: Sowohl das Kernsystem als auch die Einschübe werden als JARs ausgeliefert, die zu einem konkreten Mitglied einer Anwendungsfamilie

zusammengestellt werden müssen, bevor die Anwender mit ihrer fachlichen Arbeit beginnen können. Mit dieser Minimalumsetzung ist jedoch noch nicht sichergestellt, dass das so implementierte System auch tatsächlich alle dynamischen Charakteristika eines Einschubsystems aufweist:

(1) *Im standardmäßigen Prozess des dynamischen Bindens in Java* (siehe Anhang B.4, S. XXVIII) *werden Einschubsystem-spezifische Eigenschaften nicht überprüft*. Insbesondere ist es bei der Eingliederung von JARs im Gegensatz zu Einschüben irrelevant, ob eines der im JAR enthaltenen *class files* auch tatsächlich eine Implementation der vom jeweiligen Einschubrahmen vorgegebenen Schnittstelle enthält, so dass beliebige JARs geladen werden können. Des Weiteren widerspricht das Prüfverfahren des Standardklassenladers dem Vorgehen beim Einfügen eines Einschubs, da erst nach der endgültigen Eingliederung festgestellt wird, ob dessen Hersteller überhaupt vertraut wird.[4]

(2) *Java besitzt keine anwendungsorientierte Konfigurationsumgebung, in der Einschübe wie in Abb. 5-3 dargestellt auf verständliche Weise in Einschubrahmen verbracht werden können*. Stattdessen müssen JARs beim Start der Virtual Machine auf Kommandozeilenebene in Form von Argumenten spezifiziert werden.

Daher müssen Einschubsysteme in diesen beiden Bereichen über zusätzliche Infrastruktur verfügen, die über diejenige einer Standard-Java-VM hinausgeht. Diese zusätzliche Infrastruktur beschreibe ich in den beiden folgenden Abschnitten. In Abschnitt 5.1 stelle ich einen Einschubmanager vor, der gewährleistet, dass Einschübe nur gemäß der Systemmetapher *Einschub und Einschubrahmen* einsetzbar sind. In Abschnitt 5.2 beschreibe ich anschließend, wie ein spezieller Konfigurations-Desktop auf einfache Weise auf der Grundlage des *JWAM*-Desktops implementierbar ist. In beiden Abschnitten diskutiere ich dabei Implementationsalternativen und ihre Konsequenzen.

5.2.1 Der Einschubmanager

Bei der dynamischen Eingliederung von JARs durchlaufen die darin enthaltenen Class-Files drei Prüfstufen, bevor sie durch die VM genutzt werden können: (1) Das JAR muss zugreifbar sein, (2) der *bytecode* der Class-Files muss lesbar und zur Version der VM kompatibel sein und (3) die von den Class-Files benötigten Oberklassen bzw. Oberschnittstellen müssen entweder im JAR oder bereits in der VM vorhanden sein. Für Einschubsysteme müssen diesem Standardverfahren vier weitere Arbeits- und Prüfschritte hinzugefügt werden, um sicherzustellen, dass sich ein Einschubsystem auch gemäß der Systemmetapher *Einschub und Einschubrahmen* verhält:

- Ermittlung des Einschubherstellers und dessen Vertrauenswürdigkeit vor der endgültigen Eingliederung.

- Softwaretechnische Prüfung, ob der Einschub zum Einschubrahmen passt und keine Implementationsvererbung aus dem Kernsystem heraus verwendet.

[4] Da es sich gezeigt hat, dass formale Codeanlayse nicht zur Verifikation eines gutartigen Verhaltens geeignet ist (vgl. [FZ99, S. 648f]), wird unter Sicherheitsaspekten empfohlen, auf das Vertrauen des Anwenders gegenüber dem Hersteller zurückzugreifen (vgl. [Szy98, S. 89]), der sich durch eine digitale Signatur ausweist.

- Endgültige Eingliederung des Einschubs in das Einschubsystem.
- Ganz zu Beginn muss ermittelt werden, welche Einschubrahmen das System überhaupt besitzt

Da das in der Implementation der VM festgeschriebene Standardverfahren nicht erweitert werden kann, muss eine gesonderte Stelle in einem Einschubsystem geschaffen werden, welche das Standardverfahren um die vier beschriebenen Schritte ergänzt. Diese Stelle ist der Einschubmanager[5]. Im Detail arbeitet er wie folgt:

(1) Ermittlung der Einschubrahmen und Anmeldung der Einschubexemplaranbieter. Da in den Einschubrahmen eines Einschubsystems nur solche JARs hinzugebunden werden können, die zum Einschubrahmen passende Einschübe verkörpern, müssen als erstes die Einschubrahmen ermittelt werden. Als nächstes muss festgestellt werden, welche Einschubexemplaranbieter von den einzelnen Einschubrahmen abhängen, damit klar wird, (1) an welche Stellen des Kernsystems die Klassen aus den Einschüben geliefert werden müssen und (2) bei welchen Einschubrahmen es sich um Zwangseinschubrahmen handelt.

Implementationsbesonderheiten: Eine ideale Implementation, bei der keine statische Liste der Einschubrahmen und Einschubexemplaranbieter geführt werden muss, sondern der Einschubmanager über introspektive Dienste ermittelt, welche Einschubrahmen und Einschubexemplaranbieter im Kernsystem vorhanden sind, ist mit derzeitigen Versionen von Java nicht möglich. Die introspektiven Dienste oberhalb der Ebene von einzelnen Klassen und Schnittstellen beschränken sich auf eine Operationen zur Auflistung aller Packages, nicht aber deren Inhalts (vgl. `java.lang.Package`).

(2) Sicherheitsprüfung der Einschübe. Bevor die Klassen des Einschubs eingegliedert werden, überprüft der Einschubmanager die Vertrauenswürdigkeit des Einschubherstellers anhand des X.509-Zertifikat des JARs (vgl. [Sun98e], [Sun98f] und [ITU97]).

Implementationsbesonderheiten: Die Vertrauenswürdigkeit des *principal* genannten Unterzeichners eines JARs kann in Java nicht unmittelbar durch eine API-Operation wie beispielsweise `java.security.Principal.isTrustworthy()` überprüft werden. Stattdessen kommen nur indirekte Verfahren in Frage:

Zu jeder Quelle, die der VM als Lagerort von Class-Files bekannt ist (beispielsweise ein Verzeichnis des Dateisystems oder ein JAR), gibt es ein Exemplar der Klasse `java.security.CodeSource`, anhand derer sich die Rechte ermitteln lassen, die Class-Files aus dieser Quelle zugebilligt werden. Diese Rechte weichen nur dann von dem sehr restriktiven Standardrechten ab, wenn der Anwender für den betreffenden Code-Source einen Eintrag im seinem *policy file* vorgenommen hat (vgl.

[5] In *JRing* gibt es anstelle eines Einschubmanagers einen Parametrisierungsmanager, der zusätzlich auch die Versorgung des Kernsystems mit Konfigurationsparametern (z.B. Layout-Reihenfolge der Notizbestandteile) und Datenparametern übernimmt (beispielsweise Lagerort der Partituren im Dateisystem). Da diese Form der Parametrisierung auch in gewöhnlichen, monolithisch ausgelieferten Anwendungsfamilien vorkommt (vgl. Pre97]), wird sie in diesem Abschnitt nicht diskutiert.

[Sun98d]). Diese auf den Code-Source bezogene Vertrauensbekundung kann der Einschubmanager verwenden, um über die Vertrauenswürdigkeit eines Einschubs zu entscheiden:

- *Er kann das* policy file *direkt auslesen*. Da diese Datei plattformabhängig an unterschiedlichen Orten unter verschiedenen Namen gelagert ist, muss der Einschubmanager die betreffenden Informationen für jede Plattform kennen.

- *Er vergleicht die Standardrechte mit denjenigen des Einschubs*. Da die Standardrechte nicht explizit über die API ermittelbar sind, muss zunächst ein Code-Source-Exemplar erzeugt werden, für das garantiert kein Eintrag im *policy file* vorhanden ist – beispielsweise `http://gibt.es.nicht`. Die diesem Code-Source zugebilligtem Rechte können nun mit denjenigen des Einschubs verglichen werden.

Erst nach dieser Prüfung bindet der Einschubmanager das JAR dynamisch zum Kernsystem hinzu, um es nun auch softwaretechnisch zu untersuchen.

(3) Softwaretechnische Prüfung der Einschübe. Einschübe müssen die Schnittstelle des Einschubrahmens implementieren, in den sie hineinpassen. Implementationsvererbung aus dem Kernsystem ist nicht zulässig. Beide Eigenschaften lassen sich leicht prüfen:

- Auf genau eine Klasse des JARs muss `java.lang.Class.isAssignableFrom()` für die Schnittstelle des Einschubrahmens zutreffen – z.B. für `AustauschbarerNotizbestandteil`. Andernfalls lehnt der Einschubmanager das JAR ab.

- Für keine Klassen des JARs darf die per `java.lang.Class.getSuperclass()` ermittelte Oberklasse aus einem Paket des Kernsystem stammen.

(4) Eingliederung von Einschüben. Wenn alle vorangegangen Prüfschritte erfolgreich waren, ruft der Einschubmanager `hinzufügen()` an demjenigen Einschubrahmen auf, dessen Schnittstelle der Einschub erfüllt.

Mit diesem vierten Schritt ist die Arbeit des Einschubmanagers beendet. Während der Anwender das Einschubsystem fachlich nutzt, ergeben sich keine weiteren Besonderheiten, die eine spezielle Implementation erfordern. Lediglich beim Herunterfahren des Systems muss bei in Java implementierten Einschubsystemen ein mit der Arbeitsweise des Einschubmanagers zusammenhängender Sonderfall berücksichtigt werden, wenn der Zustand der Anwendung in Form von serialisierten Laufzeitexemplaren für die nächste Ausführung konserviert werden soll.

Beim Wiedereinlesen der Laufzeitexemplare aus einem `java.io.ObjectInputStream` überprüft `resolveClass()` normalerweise lediglich anhand des Standardklassenladers, ob die Klasse des wiederherzustellenden Laufzeitexemplars vorhanden ist. Klassen, die aus einem eigenen Klassenlader (beispielsweise des Einschubsystems) stammen, werden dementsprechend nicht gefunden, so dass in diesem Fall der Zustand

der Anwendung nicht rekonstruiert werden kann. Um dies zu verhindern, bieten sich zwei Möglichkeiten an:

- *Speicherung des* bytecodes *der Klassen aus Einschüben*. Da der Standardklassenlader die Klasse des Einschubexemplars nicht kennt, wird die Klasse in Form ihres *bytecodes* mitgespeichert. Dazu ist es notwendig, sowohl `ObjectOutputStream` als auch `ObjectInputStream` zu spezialisieren: Der spezialisierte `ObjectOutputStream` schreibt mit `ObjectOutputStream.annotateClass()` anstelle nur des Namens der Klassen aus den Einschüben deren kompletten *bytecode* in die Serialisierungsdatei und die Unterklasse von `ObjectInputStream` rekonstruiert die Klasse in `ObjectInputStream.resolveClass()` nicht anhand von Informationen aus dem Standardklassenlader, sondern unmittelbar aus dem serialisierten *bytecode*.

- *Der Einschubmanager teilt dem `ObjectInputStream` die Einschubklassenlader mit*. Immer wenn der Einschubmanager einen neuen Klassenlader erzeugt, übermittelt er ihn an ein Exemplar einer Unterklasse von *`ObjectInputStream`*.. Dieses durchsucht dann in `resolveClass()` nicht nur den Standardklassenlader, sondern auch alle weiteren ihm bekannten Klassenlader nach der Klasse des wiederherzustellenden Einschubexemplars.

5.2.2 Der Konfigurations-Desktop

Damit der Einschubmanager Einschübe eingliedern kann, muss er zunächst erfahren, welche Einschübe die Anwender in der aktuellen Sitzung welchen Einschubrahmen zuordnen wollen. Je nachdem, wie vertraut die Anwender mit technischen Sachverhalten sind, gibt es hierbei zwei Möglichkeiten:

1. Technisch versierte Anwender können Einschübe direkt auf der Kommandozeilenebene einem Einschubrahmen zuordnen. Da Java-Programme nur über Properties parametrisiert werden können, muss dementsprechend für jeden Einschubrahmen eine Property geschaffen werden – beispielsweise `einschuebe.notizbestandteile`. Die Kommandozeile zum Start eines für die set-theoretische Analyse in einer chromatischen Partitur konfigurierten *JRing*-Systems könnte dementsprechend wie folgt aussehen:

```
java JRing einschuebe.partiturengine=pe_chromatisch.jar \
           einschuebe.notizbestandteile=nb_setclass.jar \
           einschuebe.notizbestandteile=nb_icvector.jar \
           einschuebe.notizbestandteile=nb_krelation.jar \
           einschuebe.notizbestandteile=nb_khrelation.jar
```

2. Da ich in Kapitel 3 am Beispiel von Musikwissenschaftlern demonstriert habe, dass die meisten Anwender lediglich Fachleute in ihrem jeweiligen Fachgebiet, nicht aber unbedingt Experten im Umgang mit Rechnern sind, muss diesen Anwendern eine anwendungsorientierte Konfigurationsumgebung angeboten werden, in der sie die Einschübe auf für sie verständliche Weise in die gewünschten Einschubrahmen ver-

bringen können. Aufgrund der intuitiven Benutzbarkeit empfiehlt sich eine Desktop-Lösung (vgl. [Bla99b, S. 796f]), wie ich sie in Abb. 5-3 auf S. 125 dargestellt habe.

Aus technischer Sicht ist die Realisierung eines Desktops wesentlich aufwendiger als die einer Kommandozeilen-Lösung. In *JWAM* kann dieser Aufwand jedoch minimiert werden, da mit dem WAM-Desktop bereits eine Desktop-Lösung vorliegt, mit der die WAM-Entwurfsmetaphern *Werkzeug*, *Material*, *Automat*, *Behälter* und *Arbeitsumgebung* bereits ausgezeichnet handhabbar sind (vgl. [Lip99]). Da sich während der Konfiguration die Systemmetaphern des Einschub-Ansatzes als Ausprägungen der Entwurfsmetaphern des WAM-Ansatzes auffassen lassen (siehe S. 124), kann zur Implementation einer Konfigurationsumgebung weitgehend auf die existierende Implementation des WAM-Desktops zurückgegriffen werden. Da Automaten und Werkzeuge zur Konfiguration eines Einschubsystems nicht benötigt werden, stellen sich die Korrespondenzen wie folgt dar:

Einschub-Systemmetaphern	**WAM-Entwurfsmetaphern**
Konfigurationsumgebung	Arbeitsumgebung
Einschubrahmen	Behälter
Einschübe	Materialien

Die zentrale Analogie besteht dabei zwischen Einschubrahmen und Behältern (siehe Abschnitt 3.2):

(1) Aufnahmefähigkeit. Genauso, wie ein Einschubrahmen Einschübe aufnehmen kann, kann ein Behälter Materialien aufnehmen.

(2) Spezialisierung. Einschubrahmen akzeptieren nur diejenigen Einschübe, die in sie hineinpassen. Dies entspricht der Eigenschaft von Behältern, nur für eine bestimmte Art von Materialien geeignet zu sein. Beispielsweise passen in einen Aktenordner nur Akten, aber keine Archivschränke.

(3) Kapazitätsbegrenzung. Einfach- und Zwangseinschubrahmen haben eine auf einen Einschub begrenzte Kapazität, so dass ein bereits eingefügter Einschub zunächst entfernt werden muss, bevor ein anderer eingeschoben werden kann. Auch bei Behältern kann es Kapazitäten geben, bei deren Erreichen sie „voll“ sind, und keine weiteren Materialien aufnehmen können.

Da die Konfiguration eines Einschubsystems ausschließlich darin besteht, die gewünschten Einschübe in die dafür geeigneten Einschubrahmen einzufügen, genügen diese Analogien, um Einschubrahmen als spezielle Behälter eines WAM-Desktops zu implementieren. Die auf Behälter bezogenen Aktionen und deren semantisches Feedback können daher unmittelbar auch in Konfigurations-Desktops genutzt werden:

(1) Aufnahmefähigkeit. Einschübe können per *drag and drop* in Einschubrahmen hinein- und herausgezogen werden (vgl. [Lip99, S. 42]).

(2) Spezialisierung. Passt ein Einschub nicht in den Einschubrahmen, in den er gezogen werden soll, so wird graphisch angezeigt, dass diese Operation unmöglich ist (vgl. [Lip99, S. 36]).

(3) Kapazitätsbegrenzung. Wenn ein Einfach- oder Zwangseinschubrahmen gefüllt ist, verändert sich dessen graphische Repräsentation. (vgl. [Lip99, S. 85]). Außerdem wird wie unter (2) angezeigt, dass es unmöglich ist, per *drag and drop* weitere Einschübe einzufügen.

Für Anwender, die am Entwicklungsprozess beteiligt waren, ist die geschilderte Desktop-Version ausreichend, um ein Einschubsystem auf verständliche Weise zu konfigurieren. Für Anwender, die das System im Sinne von Standardsoftware erworben haben, sind die konkreten Bestandteile des Einschubsystems eventuell anfangs nicht sofort klar, so dass sie zusätzliche Erklärungen – beispielsweise über die Bedeutung des Einschubrahmens *„Partitur-Engine“* und die dort hineinpassenden Einschübe – benötigen.

Um diese Informationen zur Verfügung zu stellen, sollten die Schnittstellen `DienstleisterEinschub`, `BausteinEinschub` sowie `EinfachEinschubrahmen` und `MehrfachEinschubrahmen` eine Operation `gibBeschreibung()` enthalten. Wenn die Anwender auf dem Konfigurations-Desktop auf einen Einschub oder Einschubrahmen klicken, wird ein rein sondierendes Informationswerkzeug[6] gestartet, das `gibBeschreibung()` aufruft und die jeweilige Beschreibung darstellt. Für den Einschub *„Partitur-Engine“* könnte die Beschreibung beispielsweise wie folgt lauten:

> „Eine Partitur-Engine ist notwendig, um Partituren und Exzerpte graphisch darzustellen. Ohne eine Partitur-Engine können keine Analysematerialien angezeigt und somit keine Analysen durchgeführt werden.“

Anstatt für die Darstellung der Informationen ein eigenes Fenster zu öffnen, ist es alternativ auch möglich, die Beschreibung als *tool tip* darzustellen, der erscheint, sobald der Mauszeiger länger als eine Sekunde über dem Einschubrahmen bzw. Einschub verweilt.

5.3 Zusammenfassung und Ausblick

Zu Beginn diese Kapitels habe ich die Systemmetapher *Einschub und Einschubrahmen* eingeführt, mit der sich die Eigenschaften sowohl des Kernsystems als auch der austauschbaren Bestandteile einer Anwendungsfamilie bruchlos vergegenständlichen lassen, so dass die technischen Auslieferungseinheiten einer Anwendungsfamilie auch für technisch ungeübte Anwender verständlich sind. Nachdem ich bereits in Kapitel 4 herausgearbeitet habe, wie Konstruktionskomponenten und Auslieferungseinheiten zugeschnitten werden müssen, um die anwendungsorientierten, austauschbaren Komponenten von Anwendungsfamilien zu modellieren, habe ich mit der Entwurfsmetapher *Einschub und Einschubrahmen* nun auch die Grundlage dafür

[6] Die Existenz dieser Informationswerkzeuge widerspricht nicht der Feststellung zu Beginn des Abschnitts, dass die Konfigurationsumgebung keine Werkzeuge enthält. Informationswerkzeuge sind Desktop-Kleinstwerkzeuge, die keine eigene Repräsentation auf dem Desktop besitzen und nur eine einzige, einfache Aufgabe erfüllen, die durch Klicken auf eine bestimmte Stelle der Arbeitsfläche aktiviert werden. Klickt ein Anwender eines Apple- oder Windows-Desktops beispielsweise auf den Namen einer Datei, dann aktiviert er ein Kleinstwerkzeug, das an Ort und Stelle einen Eingabekasten darstellt, in dem der Dateiname geändert werden kann. Diese Kleinstwerkzeuge sind nicht mit fachlichen Werkzeugen im Sinne der WAM-Entwurfsmetapher vergleichbar.

gelegt, dass diese Bestandteile von Anwendungsfamilien auch von den Anwendern selbst und nicht nur von den Systementwicklern oder -administratoren sitzungsweise zusammengestellt werden können.

Anschließend habe ich in den Abschnitten 5.1 und 5.2 demonstriert, wie sich die zuvor nur im Kontext der *Handhabung & Präsentation* diskutierten Einschübe und Einschubrahmen im softwaretechnischen Modell einer Anwendungsfamilie modellieren und am Beispiel von *JWAM* konkret implementieren lassen.

In Abschnitt 5.1 habe ich zunächst herausgearbeitet, dass Einschubrahmen sich auf zweierlei Weise im softwaretechnischen Modell manifestieren:

- Zum einen als konkrete Klassen, die mit denjenigen Stellen des Kernsystems in Verbindung steht, die zur Laufzeit Exemplare der in den Einschüben enthaltenen Klassen erzeugen. Diese Einschubexemplaranbieter (abstrakte Dienstleister im Fall von Dienstleistereinschüben sowie Bausteinfabriken im Fall von Bausteineinschüben) sind Beobachter des Einschubrahmens und werden mit den Klassen aus den Einschüben versorgt, sobald der Einschubmanager die Operation `hinzufügen()` am jeweiligen Einschubrahmen aufruft.

- Zum anderen sind Einschubrahmen auch derjenige Bereich des Kernsystems, den Einschübe zur Laufzeit einsehen können, um auf die dort bekannt gemachte Auswahl von internen Dienstleistern des Kernsystems zugreifen zu können. Damit keine Abhängigkeiten von einer bestimmten Implementation des Kernsystems entstehen können, werden sowohl die internen Dienstleister als auch die zur Kommunikation zwischen Kernsystem und Einschüben benötigten Parameterklassen im Einschubrahmen durch Schutz-Proxies repräsentiert.

Einschübe müssen die von ihrem Einschubrahmen vorgegebene Schnittstelle vollständig implementieren, ohne sich dabei auf Implementationsvererbung aus dem Kernsystem oder aus anderen Einschüben zu stützen. Funktionalität des Kernsystems kann ausschließlich über im jeweiligen Einschubrahmen verfügbare interne Dienstleister genutzt werden. Auf diese Weise können maximal unabhängige Einschübe im Sinne des Ideals der Komponenten-Idee (siehe Abschnitt 4.2) realisiert werden. Komplexe Abhängigkeitsgeflechte zwischen Einschüben können nicht entstehen, da Einschübe nicht direkt auf andere Einschübe zugreifen können, sondern nur auf deren Einschubexemplaranbieter, die durch gewöhnliche interne Dienstleister gekapselt werden.

In Abschnitt 5.2 habe ich demonstriert, wie sich Einschübe in Java auf der Basis von digital signierten JARs realisieren und ausliefern lassen. Dabei habe ich zunächst einen Einschubmanager diskutiert, mit dessen Hilfe das Standardverfahren einer Java-VM zum dynamischen Eingliedern von JARs so ergänzt wird, dass diejenigen JARs zurückgewiesen werden die (1) ein Sicherheitsrisiko für das Einschubsystem darstellen, (2) nicht zum spezifizierten Einschubrahmen passen oder (3) Implementationsvererbung aus dem Kernsystem verwenden. Anschließend habe ich gezeigt, dass der für die anwendungsorientierte Anpassung des Kernsystems notwendige Konfigurations-Desktop in *JWAM* nicht aufwendig von Grund auf neu implementiert werden muss, sondern als Spezialisierung des bereits existierenden WAM-Desktops realisiert werden kann. Da

Einschubrahmen sich im Rahmen der Konfiguration wie Behälter und Einschübe sich wie Materialien verhalten, können die diesbezüglichen Eigenschaften des WAM-Desktops in vollem Umfang genutzt werden.

Bereits in Kapitel 3 habe ich herausgearbeitet, (1) welche Anforderungen Anwendungssysteme zur Unterstützung musikwissenschaftlichen Analyse erfüllen müssen und (2) dass diese Anwendungssysteme sich als eine Anwendungsfamilie auffassen lassen. Mit den in den Kapiteln 4 und 5 diskutierten Konzepten zur Modellierung und Implementierung von Anwendungsfamilien habe ich nun außerdem die Fragen beantwortet, (3) welche Eigenschaften Anwendungsfamilien aufweisen müssen, um sitzungsweise durch die Anwender anpassbar zu sein und (4) wie sich zu diesem Zweck Rahmenwerk- und Komponenten-Konzepte dergestalt aufeinander beziehen lassen, dass keine Strukturbrüche zwischen den Kontexten *Anwendungsbereich*, *Handhabung & Präsentation* und *verwendete Technik* entstehen. Mit dem am Beispiel des Einschubsystems *JRing* konstruktiv validierten Einschub-Ansatz habe ich somit zwei Probleme gelöst, die mit herkömmlichen Ansätzen bei der Modellierung von Anwendungsfamilien auftraten:

- Das Prinzip der Anwendungsorientierung kann auf die Konfiguration von Anwendungssystemen ausgedehnt werden und muss nicht wie bisher auf deren Entwicklung und deren Handhabung zur Laufzeit beschränkt bleiben.

- Es ist nicht mehr notwendig, Anwendungsfamilien als monolithische *fatware* auszuliefern, da Anwender mit dem Einschub-Ansatz konkrete Anwendungssysteme flexibel sitzungsweise an ihre Anforderungen anpassen können.

Damit habe ich den Einschub-Ansatz vollständig beschrieben. In Kapitel 6 diskutiere ich die letzte offene Fragestellung dieser Arbeit. Dabei untersuche ich Konstruktionslösungen, um das MDV-Subsystem *JRings* dergestalt zu kapseln, dass das Analysesystem auch dann noch sinnvoll nutzbar ist, wenn aus technischen Gründen kein MDV-Subsystem wie beispielsweise *Humdrum* verfügbar ist. Dabei zeige ich, dass die Lösungen durch den Einschub-Ansatz begünstigt werden und zudem geeignet sind, um generelle Konstruktionsprobleme innerhalb des WAM-Ansatzes zu beheben, die sich bei der Kopplung von Materialien und Fachwerten mit komplexen Subsystemen ergeben.

6 Modellierungsmittel zur flexiblen Kapselung bestehender Subsysteme

Wie ich in Abschnitt 3.3 am Beispiel der Spezifikation eines musikwissenschaftlichen Analysesystems demonstriert habe, ist es aus ökonomischen Gründen oft wünschenswert, bei der Realisierung komplexer Funktionalität auf bestehende Systeme zurückzugreifen, die somit zu Subsystemen des zu entwickelnden Anwendungssystems werden. Da solche Systeme oft ursprünglich nicht explizit als Subsysteme konzipiert worden sind – und wie beispielsweise *Humdrum* auch nicht in einer objektorientierten Implementation vorliegen –, müssen sie zunächst auf geeignete Weise gekapselt werden, bevor sie nutzbringend einsetzbar sind. Je nachdem, wie sich die Schnittstelle des Subsystems zu derjenigen des Anwendungssystems verhält, bieten sich unterschiedliche Entwurfsmuster an, um das Subsystem zu kapseln:

- Eine *Brücke* ist geeignet, wenn die Schnittstellen funktional größtenteils identisch sind und Operationsaufrufe innerhalb des Anwendungssystems somit unmittelbar auf analoge Operationen des Subsystems abbildbar sind (vgl. [GHJV98, S. 165]).

- Wenn sich keine vollständige Analogie der Operationen etablieren lässt, bietet sich ein *Adapter* an. Hierbei werden Operationen des Anwendungssystems auf eine oder mehrere Operationen des Subsystems abgebildet, die für sich genommen nicht kompatibel wären (vgl. [GHJV98, S. 151]).

- Falls die Klassen des Subsystems nicht individuell gekapselt werden sollen bzw. nicht individuell gekapselt werden können, weil das Subsystem nicht in objektorientierter Form vorliegt, dann empfiehlt es sich, eine *Fassade* zu verwenden. Eine Fassade bündelt sämtliche Zugriffe auf ein Subsystem in einer einzigen Klasse, und schirmt das Subsystem auf diese Weise vollkommen vom Rest des Anwendungssystems ab. Hinter einer Fassade können daher auch leicht Zugriffe auf nicht objektorientierte Systemteile oder durch externe Programme realisierte Funktionalität verborgen werden (vgl. [GHJV98, S. 189]).

Im Einschub-Ansatz sind austauschbare Subsysteme auf einfache Weise durch eine Kombination des Brücken- und des Fassade-Musters realisierbar. Zunächst wird eine einheitliche Fassade-Klasse für alle austauschbaren Subsysteme etabliert, die somit zur Schnittstelle eines Dienstleistereinschubs wird. Im Kernsystem wird dann über den Abstraktionsteil einer Brücke auf die jeweils konkrete, durch den Einschub gelieferte Implementation zugegriffen (siehe S. 86 in Kapitel 4).

Im WAM-Ansatz haben sich zwei Kapselungsverfahren für Subsysteme herausgebildet:

(1) *Technische Subsysteme* wie zum Beispiel konkrete Persistenzdienstleister werden in der Systemschicht gekapselt. Sie werden nur von speziellen technischen Diensten innerhalb des Systems aufgerufen und sind den fachlichen Systembestandteilen wie Werkzeugen und Materialien unbekannt. Es ist somit in den fachlichen Schichten ir-

relevant, ob eine Registratur ihre Daten mit Hilfe einer objektorientierten oder einer relationalen Datenbank verwaltet, oder ob sie die sogar unmittelbar lesend und schreibend auf das Dateisystem zugreift. In Verbindung mit dem Einschub-Ansatz kann über die konkrete Realisierung sogar sitzungsweise entschieden werden (vgl. [GKR+01]).

(2) *Fachliche Subsysteme* wie beispielsweise MDV-Subsysteme können nicht tief in der Systemschicht verborgen werden, sondern müssen unmittelbar an fachliche Systembestandteile angebunden werden. Üblicherweise kennen hierbei ausschließlich die relevanten Werkzeuge und Automaten das jeweilige fachliche Subsystem. Den Materialien und Behältern sind die Subsysteme hingegen nicht bekannt. Sollen Materialien und Behälter die Funktionalität der fachlichen Subsysteme nutzen, dann müssen sie ausschließlich von den auf ihnen arbeitenden Werkzeugen und Automaten mit aus dem Subsystem bezogenen Daten versorgt werden. Dies deckt sich mit den WAM-Metaphern, in denen alleine Werkzeuge und Automaten aktiv sind und Materialien und Behälter sowie die Umgebung sich passiv verhalten.

Am Beispiel des MDV-Subsystems von *JRing* zeige ich in diesem Kapitel, dass die herkömmliche Modellierung fachlicher Subsysteme problematisch ist, wenn die Materialien (1) graphisch komplex sind oder (2) fachlich komplexe Funktionalität anbieten. Stattdessen schlage ich eine neuartige Kapselung dieser komplexen Subsysteme direkt durch die Materialien bzw. ihrer Fachwerte vor. Durch eine geschickte Aufteilung der Funktionalität auf sowohl die Kapsel als auch das Subsystem lässt sich mit diesen Verfahren zudem die in Kapitel 3 aufgestellte Forderung erfüllen, dass *JRing* auch bei fehlendem MDV-Subsystem noch sinnvoll einsetzbar sein soll.

Dazu diskutiere ich in diesem Kapitel in zwei getrennten Abschnitten jeweils für graphisch komplexe Materialien (Abschnitt 6.1) und komplexe Fachwerte (Abschnitt 6.2)

1. inwiefern sich deren Modellierungsanforderungen von denen herkömmlicher, d.h. einfacher Materialien und Fachwerte unterscheiden,

2. wie sie sich als schlanke graphisch komplexe Materialien (Abschnitt 6.1.1) und schlanke komplexe Fachwerte modellieren lassen (Abschnitt 6.1.2) und welche Vorteile dies mit sich bringt sowie

3. auf welche Weise diese Modellierung in *JRing* verwendet wird, um Partituren (Abschnitt 6.1.2) und Notizen (Abschnitt 6.2.2) zu realisieren.

Da ich Fachwerte bisher noch nicht ausführlich diskutiert habe, stelle ich das Konzept in Abschnitt 6.2 kurz vor und zeige, warum die Notizbestandteile musikwissenschaftlicher Analysesysteme sich gut als Fachwerte auffassen und modellieren lassen.

6.1 Partituren als schlanke graphisch komplexe Materialien

Entsprechend den herkömmlichen Modellierungsrichtlinien des WAM-Ansatzes sind Materialien im Zuge der Trennung von Funktion und Interaktion nicht für ihre Darstellung an der Bedienschnittstelle verantwortlich. Lediglich Werkzeuge besitzen eigene Interaktionskomponenten, die um die plattformspezifischen Besonderheiten der Bedienschnittstelle wissen (siehe Abschnitt 3.2). Die Materialien machen keinerlei Annahmen über die interaktive Umgebung, in die sie eingebettet sind (vgl. [WAM98, S. 237]), sondern besitzen lediglich Operationen, anhand derer Werkzeuge ihren Zustand sondieren und eine dementsprechende graphische Präsentation generieren können (vgl. [WAM98, S. 169f]).

Um Materialien wirklich nur in ihren fachlichen Aspekten zu modellieren, enthalten sie nicht nur keine Bezüge „nach oben“ auf die Bedienschnittstelle (Kontext *Handhabung & Präsentation*), sondern – wie bereits erwähnt – auch nicht „nach unten“ auf die Art und Weise ihrer persistenten Speicherung (Kontext *verwendete Technik*). Materialien greifen daher nicht unmittelbar auf Persistenzmedien wie Dateien, Datenbanken oder andere Ressourcen zu, um ihren Zustand mit programmexternen Stellen zu synchronisieren, sondern sind hierzu auf als Softwareautomaten realisierte Materialversorger angewiesen. In Anlehnung an die Verwaltungswissenschaften sind diese Materialversorger in WAM als Registraturen realisiert, die von einem Registrar verwaltet werden. An diesen Registrar wenden sich Werkzeuge und Automaten, wenn sie ein Material bearbeiten wollen (vgl. [GHR+00]). Den Materialien selbst sind sowohl die Registratur als auch der Registrar unbekannt (vgl. [WAM98, S. 299f]).

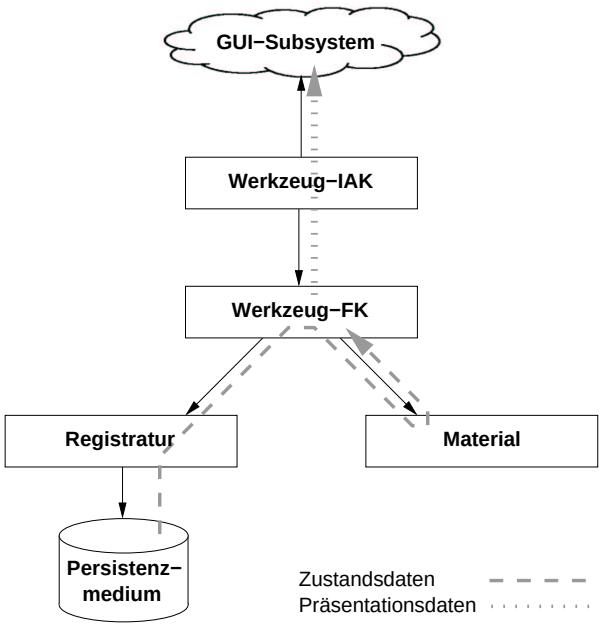


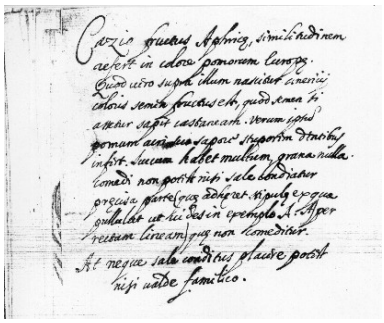
Abb. 6.1-1 Datenfluss bei der Darstellung von Materialien im Werkzeug- und Material-Ansatz WAM.

Die Registratur ist der Übersichtlichkeit halber nicht in weitere Klassen untergliedert.

Diese Art der Modellierung, bei der Materialien vollkommen auf Werkzeuge angewiesen sind, um dargestellt zu werden, ist für den Bereich der Sachbearbeitertätigkeiten, auf den sich WAM bisher primär konzentriert hat, eine angemessene Lösung. Die dort vorhandenen Materialien wie z.B. Konten und Verträge haben einen formularartigen Charakter, deren Layout von den sie bearbeitenden Werkzeugen jederzeit problemlos automatisch generiert und mit Standard-Interaktionstypen (Textfeldern, Listen, Auswahlmenüs) angezeigt werden kann. Dieses Verfahren ist auch für Materialien geeignet, die sich in Form einer Tabelle (als Raster von Textfeldern; vgl. [WAM98, S. 187]) oder eines Diagramms darstellen lassen (als einfache Zeichnung in einer Zeichenfläche; vgl. [Wul95] und [Nie98]).

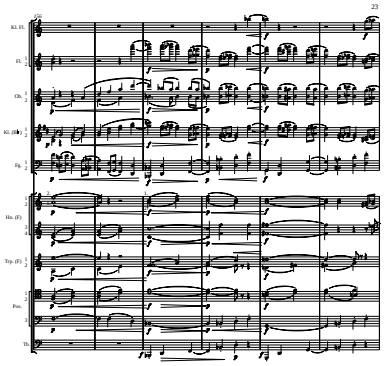
Problematisch ist diese Art der Modellierung für Materialien, deren Präsentation so komplex ist, dass sie sich nicht automatisch in zufriedenstellender Qualität generieren lässt, und die somit auf externe, vorgefertigte Präsentationsdaten zurückgreifen müssen. Beispiele für solche *graphisch komplexen Materialien* sind mittelalterliche Handschriften, Stadtpläne und Partituren:

- In einem Analysewerkzeug für mittelalterliche Handschriften sind die künstlerische Ausführung und der Zustand des Manuskripts ebenso wichtig wie der Inhalt des Texts selbst. Ohne eine detailgetreue, graphische Wiedergabe des Manuskripts kann keine vollständige Analyse durchgeführt werden (vgl. [Mat94] und [Wie96]). Die graphischen Informationen sind aus dem reinen Text schlechterdings nicht rekonstruierbar.



- Ein Tourenplaner-Werkzeug erstellt zwar unter anderem auch eine Liste von Navigationsanweisungen (beispielsweise „..., nach links auf die Esplanade, wird nach 100m zur Lombardsbrücke, wird nach 300m zum Glockengießerwall, dann nach 100m rechts auf den Ballindamm, ...“) ist aber ohne eine graphische Darstellung der Route zur Detailnavigation im Innenstadtbereich und zur optischen Kontrolle nicht denkbar. Ansprechendes Kartenlayout gilt als nicht automatisch generierbar.[1]

- Im Rahmen der musikwissenschaftlichen Analyse müssen Partituren nicht nur nach logischen Gesichtspunkten durchsucht werden können (beispielsweise nach dem nächsten Vorkommen der melodischen Grobkontur „gleich, gleich, abwärts“), sondern auch graphisch darstellbar sein (siehe Abschnitt 3.2). Polyphoner Notensatz ist aber eine Expertentätigkeit, so dass die eine graphische Partiturdarstellung nicht



[1] Ein instruktives Negativbeispiel für automatisch generierte Innenstadtpläne stellt http://www.mapquest.com dar.

automatisch aus den logischen Partiturdaten erzeugt werden kann.[2]

Begriff 6.1 graphisch komplexes Material *(graphically complex material)*

Die bildliche Repräsentation eines graphisch komplexen Softwarematerials kann von den Werkzeugen, die es darstellen, nicht automatisch generiert werden. Seine Präsentationsdaten müssen entweder als Raster- oder Vektorgraphik vor dem Start des Anwendungssystems vor-layoutet und zur Laufzeit eingelesen werden.

Unabhängig davon, ob die Layoutdaten als eingescannte Rastergraphiken (für alle drei Beispiele denkbar) oder als Vektorgraphiken vorliegen (für Handschriften ausgeschlossen), ist der Speicherplatzbedarf eines kompletten graphisch komplexen Materials sehr hoch. Wenngleich das Material seine Präsentation über eine einzige Operation wie z.B. `gibPraesentation()` sondierbar machen kann, so müssen im Rahmen der WAM-üblichen Modellierung von Materialien auch die zur Präsentation notwendigen Daten bereits vollständig in ihm vorhanden, d.h. von der Registratur geladen und an das bearbeitende Werkzeug weitergereicht worden sein. Aufgrund des potentiell sehr großen Umfangs der Präsentationsdaten kann dies zu einer intolerablen Inanspruchnahme des Hauptspeichers führen und die Verfügbarkeit des Systems negativ beeinflussen. Obwohl in den meisten Fällen nur ein kleiner Ausschnitt der Präsentationsdaten angezeigt werden soll, erlaubt besagte Modellierung es nicht, dass die Materialien selbst – gemäß den tatsächlichen Anforderungen und unter Umgehung der Werkzeuge und der zentralen Registratur – in minimaler Weise auf die Persistenzmedien zugreifen, um den tatsächlich benötigten Teil der Präsentation eigenständig zu laden. Das in [WAM98] auf den Seiten 494f beschriebene Optimierungskonzept für große Objektgeflechte bezieht sich lediglich auf Geflechte *mehrerer* Materialien – wie z.B. eines Behälters und seines Inhalts – und ist daher auf die geschilderten, großen, fachlich nicht weiter teilbaren Materialien nicht anwendbar.

Um auch graphisch komplexe Materialien in WAM modellieren zu können, ohne dabei zur Laufzeit Speicherplatzprobleme zu bekommen, müssen Lösungsmöglichkeiten gefunden werden, die eine gezielte Versorgung der Materialien mit ausschließlich den aktuell benötigten Präsentationsdaten ermöglichen:

- **Graphisch komplexe Materialien werden in zwei eigenständige Materialien getrennt – eines mit den fachlichen und eines mit den Präsentationsdaten.** Das Präsentationsmaterial wird so lange weiter zerlegt, bis die separat vom Materialversorger geladenen Einzelteile einen vertretbaren Speicherplatzbedarf aufweisen. Werkzeuge arbeiten dann gleichzeitig mit sowohl einem fachlichen als auch – je nach gewähltem Ausschnitt – mindestens einem Präsentationsmaterial. Für diese Lösung wird allerdings das bei WAM zentrale Prinzip der Anwendungsorientierung bei der Modellierung von Materialien aufgegeben und einem rein technischen Kriterium untergeordnet.

[2] Für einstimmige Werke wie z.B. Volkslieder ist der automatische Notensatz aufgrund der nicht notwendigen chronologischen Synchronisation in zufriedenstellender Qualität möglich. Beispiele hierfür sind *MUP*[CS97, S. 593] und die Kombination aus *EsSCORE*[Net98b, S. 161] und *SCORE*, mit denen aus einstimmigen Melodiedaten Notensatzdaten im Seitenbeschreibungsformat „Encapsulated Postscript“ erzeugt werden können.

- **Materialien und Registratur werden zu einem einzigen fachlichen Dienstleister verschmolzen.** Fachlichen Dienstleister sind im WAM-Kontext verwendete technische Automaten ohne eigene Bedienschnittstelle, die Teile der fachlichen Funktionalität von Softwarewerkzeuge und -automaten zu realisieren (vgl. [OS00]). Im Gegensatz zu rein technischen Automaten wie z.B. einer Datenbank-Engine besitzen fachliche Dienstleister eine anwendungsfachliche Schnittstelle. Ein fachlicher Dienstleister `Kundenverwaltung` hätte dementsprechende keine Methode `getBlob( String key )`, sondern `gibKunden( fwKundenNummer nr )`. Fachliche Dienstleister sind sinnvoll, wenn mehrere Werkzeuge ähnliche Funktionalität redundanzlos zur Verfügung stellen sollen (vgl. [Boh00]) oder um Funktionalität zu bündeln (vgl. [Stu00, S. 38]).

  Der fachliche Dienstleister, der das graphisch komplexe Materialien und dessen Persistenz realisiert, stellt modifizierende und sondierende Operationen zur Verfügung und gibt auf Anfrage eine dem aktuellen Zustand entsprechende graphische Materialrepräsentation heraus. Damit wird zwar der Persistenzmechanismus der Materialien besser gekapselt als bei der ersten Alternative, aber die Trennung in einen fachlichen und eine präsentationsbezogenen Materialteil bleibt bestehen. Im Gegensatz zur ersten Alternative geht der fachliche Materialteil dabei lediglich im neu geschaffenen fachlichen Dienstleister auf. Eine bruchlose anwendungsorientierte Modellierung der Materialien kann auf diese Weise nicht gewährleistet werden.

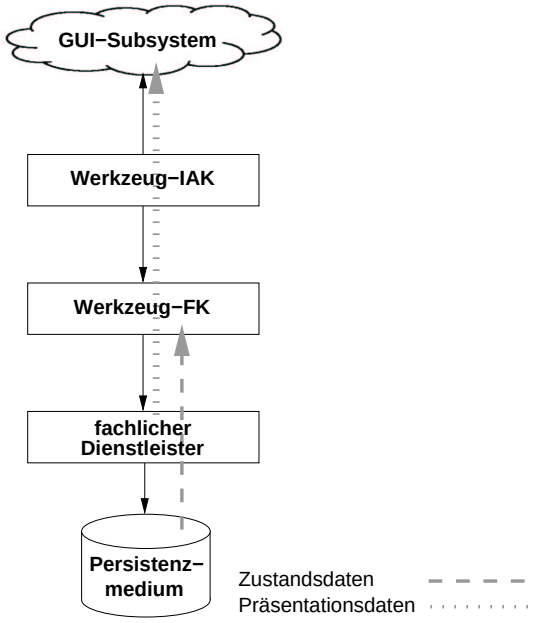


Abb. 6.1-2 Datenfluss bei der Modellierung von graphisch komplexen Materialien als fachliche Dienstleister.

Eine dritte, weniger problematische Lösung, stellen *schlanke graphisch komplexe Materialien* dar.

6.1.1 Schlanke graphisch komplexe Materialien

Hinter der Schnittstelle, die der eines herkömmlichen Materials gleicht, delegieren schlanke graphisch komplexe Materialien Anfragen bezüglich der Repräsentation eigenständig an einen Präsentationsversorger. Sie selbst verwalten lediglich ein Raster, mit dessen Hilfe sie fachliche Koordinaten („Esplanade“) auf einen bestimmten Teil der graphischen Repräsentation abbilden können („Planquadrat N24“) und dann die eigentliche graphische Repräsentation vom Präsentationsversorger beziehen und unmittelbar an das aufrufende Werkzeug durchreichen. Um die größte hauptspeicherplatzbezogene Verbesserung gegenüber der herkömmlichen Modellierung von Materialien zu erreichen, muss das Raster dabei für jedes Material individuell so gewählt werden, dass seine Zellen mit den kleinsten fachlich noch relevanten rechteckigen Einheiten korrespondieren.

Der Präsentationsversorger muss zusammen mit dem Material entworfen werden und mit der gleichen Rastergröße arbeiten wie dieses. Zu jeder Rasterzelle weiß er, wie die dementsprechende Kachel vom durch ihn gekapselten Persistenzmedium geladen werden kann, und liefert sie an das Material zurück. Ob und wie er Kacheln gegebenenfalls zwischenspeichert, ist für das Material irrelevant. Anstatt unmittelbar selbst auf ein Persistenzmedium zuzugreifen, kann der Präsentationsversorger auch einen weiteren internen Dienstleister oder ein externes Subsystem aufrufen, und dessen Daten so aufbereiten, dass sie dem vom Material erwarteten Format entsprechen. Sowohl hinsichtlich seiner fachlich zugeschnittenen Schnittstelle als auch bezüglich seiner Kapselungswirkung ist der Präsentationsversorger somit eine fachlicher Dienstleister. Beispielsweise kann ein Präsentationsversorger für Stadtpläne Anfragen nach Kartenausschnitten an externe, als Subsystem angebundene Kartographiesoftware eines dritten Herstellers weiterleiten, indem er zunächst die Planquadrat-Koordinaten umrechnet, die resultierende Graphik dann zurechtschneidet und vom TIFF-Format in das vom Material erwartete GIF-Fomat konvertiert.

Begriff 6.2 schlankes graphisch komplexes Material
(lightweight graphically complex material)

Ein graphisch komplexes Softwarematerial ist schlank, wenn es seinen Klienten eine einheitliche Schnittstelle bietet, über die diese sowohl auf fachliche als auch auf präsentationsbezogene Informationen zugreifen können. Die Präsentationsdaten hält es aber nicht selbst vor, sondern bezieht sie von einem Präsentationsversorger.

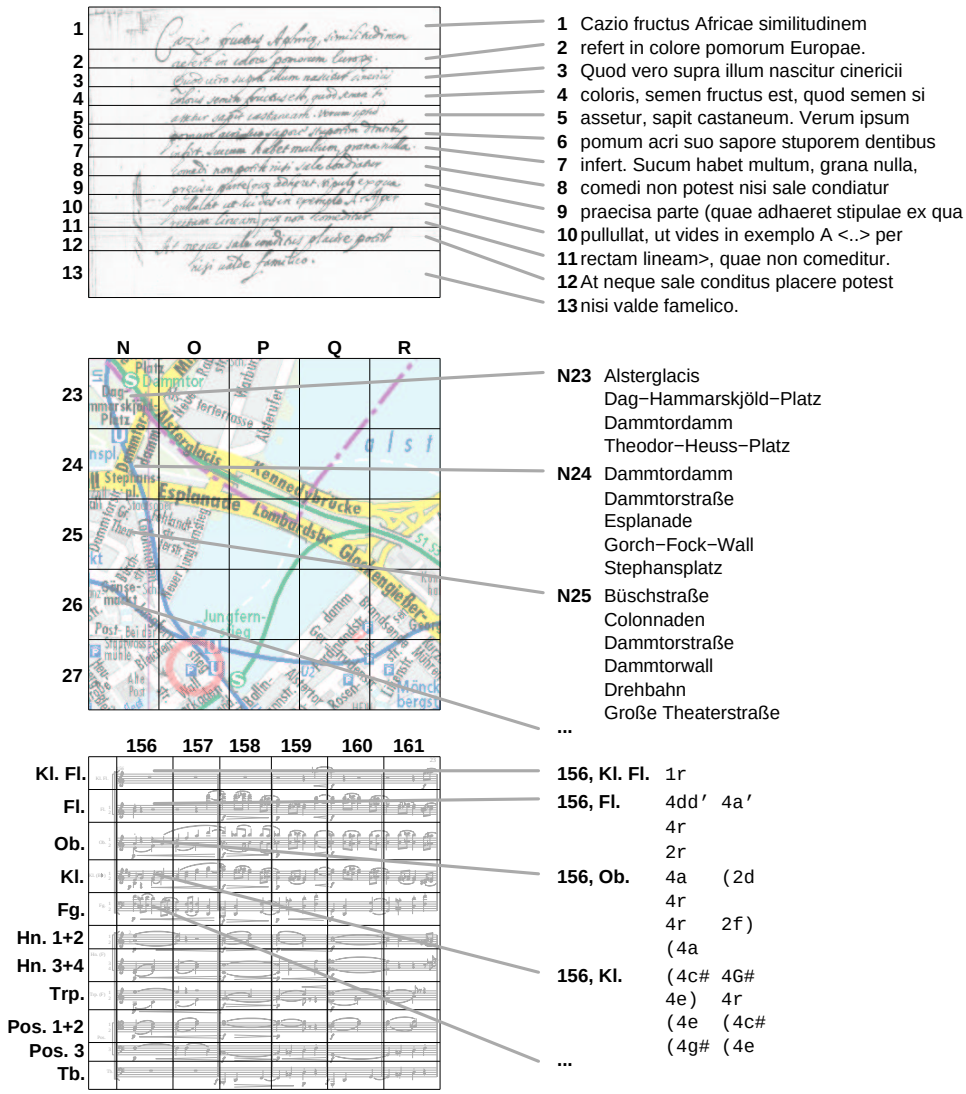


Abb. 6.1-3 Mögliche Raster für Handschriften, Stadtpläne und Partituren.

Für einspaltige Handschriften lässt sich außer der vertikalen Unterteilung in Zeilen für die meisten Anwendungen keine horizontale Unterteilung finden. Bei Stadtplänen bieten sich Planquadrate an. Partituren lassen sich in Zellen von der Größe eines Taktes einer Stimme unterteilen

Schlanke graphisch komplexe Materialien besitzen als eigene Materialrepräsentation nur ein grobes Raster, dessen Zellen je genau einer kleinsten fachlich und graphisch sinnvollen Unterteilung entsprechen – beispielsweise einem Takt einer Stimme in einer Partitur, einer Zeile einer Handschrift oder einem Planquadrat einer Karte. Bei repräsentationsbezogenen Anfragen ermitteln sie lediglich den betroffenen Rasterbereich und delegieren die Anfrage in reduzierter Form an ihren Präsentationsversorger. Der Präsentationsversorger ist ein spezieller fachlicher Dienstleister, der zu einem gegebenen Rasterbereich die entsprechende graphische Repräsentation liefern kann.

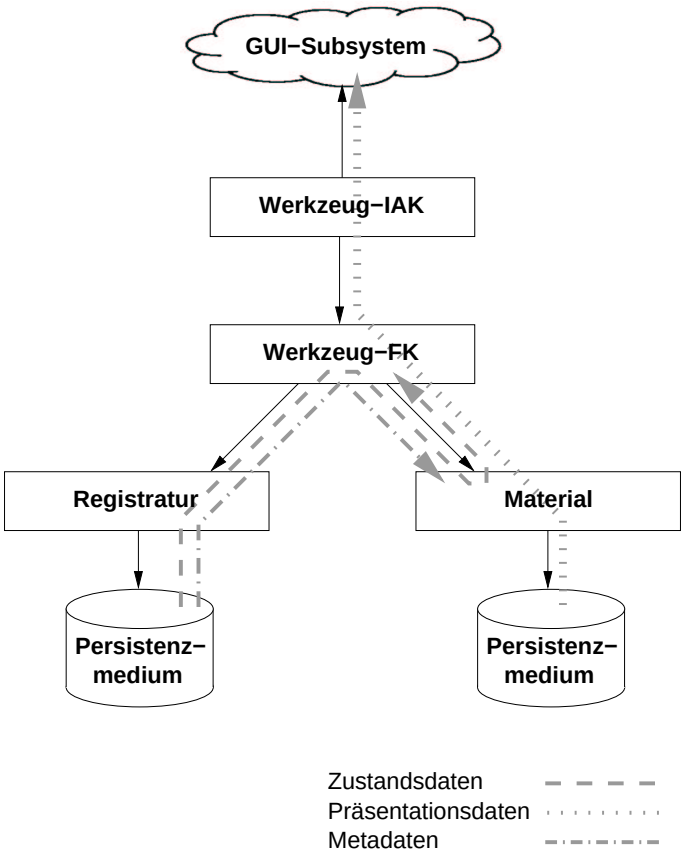


Abb. 6.1-4 Datenfluss bei der Darstellung von schlanken, graphisch komplexen Materialien.

Im Gegensatz zur herkömmlichen Materialmodellierung (siehe Abb. 6.2-1) werden die Präsentationsdaten nicht erst im Werkzeug generiert, sondern von einem Präsentationsversorger bezogen.

Für graphisch komplexe Materialien ist eine schlanke gegenüber der herkömmlichen Modellierung in vielerlei Hinsicht *vorteilhaft*:

- Das Material muss nicht die gesamten zu seiner Präsentation notwendigen Bilddaten selbst enthalten. Dadurch, dass nur ein Raster der graphischen Struktur des Layouts permanent im Material gehalten wird und die eigentlichen Bilddaten aus einem separaten Präsentationsversorger bezogen werden, sinkt der Speicherplatzbedarf erheblich.

- Da das Material nicht unmittelbar, sondern über einen Präsentationsversorger auf Persistenzmedien zugreift, bleibt die Entkopplung von Details der technischen Realisierung gewahrt.

- Die fachliche Einheit der Materialien bleibt erhalten.

- Die Schnittstelle zwischen Werkzeugen und Materialien bleibt unverändert.

- Mehrere Materialien können auf den gleichen Präsentationsversorger zurückgreifen und so den Implementationsbedarf weiter verringern. In *JRing* greifen beispielsweise sowohl Partituren als auch Notizen (für die Exzerpte) auf die Partitur-Engine zu.

Die einzigen *Nachteile* der schlanken Modellierung von graphisch komplexen Materialien sind der höhere Implementationsaufwand und eine ggf. leicht erhöhte Laufzeit durch die zusätzlichen Indirektionen. Hinsichtlich der folgenden Eigenschaften verhalten sie sich neutral zu herkömmlich modellierten graphisch komplexen Materialien:

1. Materialien werden weiterhin nicht von sich aus aktiv. Sie initiieren nur dann eine Anfrage an den Präsentationsversorger, wenn ihnen vom Werkzeug aus mitgeteilt wird, dass sich die Sicht auf das Material verändert hat.

2. Materialien sind aus der für WAM primären Perspektive der Anwendungsorientierung fachlich weiterhin vollkommen passiv, so dass kein Widerspruch zur Entwurfsmetapher *Material* entsteht. Die Aktivität, einen Präsentationsversorger aufzurufen, ist rein technisch und für die Anwender nicht erkennbar. Die Modellierung wäre nur dann problematisch, wenn Materialien sich neben diesen technischen Aktivitäten auch aus fachlicher Sicht aktiv verhielten und beispielsweise den Zustand anderer Materialien und Werkzeuge eigenmächtig modifizieren würden.

3. In Fehlersituationen ist es unerheblich, ob graphisch komplexe Materialien schlank oder herkömmlich modelliert sind: In beiden Fällen führt ein Ausfall des Präsentationsversorgers bzw. der Registratur dazu, dass das Material nicht mehr sinnvoll dargestellt werden kann. Ein Nachteil für die schlanke Modellierung ergibt sich somit nicht.

6.1.2 Anwendung innerhalb *JRings*

Mit der Modellierung von Partituren und Notizen als schlanke graphisch komplexe Materialien ist in *JRing* das Problem gelöst, wie diese Materialien zur Laufzeit mit einem vertretbaren Maß an Hauptspeicherplatz auskommen können. Noch offen ist die Frage, wie sie so modelliert werden können, dass mit ihnen auch dann gearbeitet werden kann, wenn aus ökonomischen oder technischen Gründen kein MDV-Subsystem vorhanden ist (siehe Kapitel 3). Auch dieses Problem ist durch die Modellierung als schlanke graphisch komplexe Materialien bereits fast vollkommen gelöst, wenn man den bisher geschilderten Zusammenhang zwischen den Werkzeugen des Kernsystems, den Materialien, der Partitur-Engine und dem MDV-Subsystem betrachtet:

- Die mit Partituren und Notizen arbeitenden **Werkzeuge** greifen nicht unmittelbar auf die Partitur-Engine oder das MDV-Subsystem zu. Zur Versorgung mit Präsentationsdaten rufen sie ausschließlich `gibPräsentation()` der Materialien auf (siehe Abschnitt 6.1.1). Zum fachlichen Durchsuchen und Vergleichen wenden sie sich an deren dementsprechende Operationen (siehe Kapitel 3).

- Die **Materialien** implementieren nur einfache Operationen zur Gänze selbst. Alle drei Arten von komplexen Operationen werden hingegen an andere Stellen delegiert:

  1. Die Präsentationsdaten werden von der **Partitur-Engine** bezogen, welche die Rolle des Präsentationsversorgers übernimmt (siehe Abschnitt 6.1.1).

2. Die auf die Notizbestandteile bezogenen Operationen werden an die als Fachwerte realisierten Notizbestandteil-Bausteine weitergeleitet (siehe Abschnitt 6.2).

3. Die fachlichen Such- und Vergleichsoperationen übernimmt das **MDV-Subsystem** (siehe Abschnitt 3.3).

Das bedeutet, dass bereits zwei von drei Arten der Operationen der Materialien ohne direkte Zugriffe auf das MDV-System ausgeführt werden können. Um maximal von einem aus ökonomischen oder technischen Gründen fehlenden MDV-Subsystem unabhängig zu sein, muss lediglich noch sichergestellt sein, dass die Partitur-Engine und die Fachwerte für die Notizbestandteile auch nicht indirekt in ihrer Funktionalität auf ein MDV-Subsystem angewiesen sind.

- Für die *Partitur-Engine* ist dies bereits durch den Einschub-Ansatz gewährleistet: Alle Einschubrahmen – und damit auch die Einschubrahmen *„Partitur-Engine“* und *„MDV-Subsystem“* – sind voneinander unabhängig und besitzen keine Referenzen aufeinander (siehe Begriff 5.1). Da *„MDV-Subsystem“* kein Zwangseinschubrahmen ist, kann es in ihm auch keinen internen Dienstleister geben, über den Partitur-Engine-Einschübe in gekapselter Form auf MDV-Subsysteme zugreifen könnten (siehe Abschnitt 5.1.4).

- Die Lösung für die *Fachwerte* diskutiere ich im nachfolgenden Abschnitt 6.2.

Um unabhängig vom MDV-Subsystem zu sein, werden die in *Humdrum-Layout* vorliegenden Daten zur Partiturdarstellung (siehe Abschnitt 3.3.2) von der Partitur-Engine nicht direkt über *Humdrum* eingelesen. Stattdessen wird bereits vor dem Start *JRings* – gegebenenfalls vom Lieferanten der Partitur auf dessen Rechner – zu jeder Partitur anhand der *Humdrum-Layout*-Daten eine komplette Repräsentation der Partitur in einer Notensatzsymbol-basierten Vektordarstellung generiert, die anschließend in serialisierter Form unabhängig von der *Humdrum-Kern*-Partitur im Dateisystem abgelegt wird. Nur auf diese Datei mit serialisierten Notensatzsymbolen greift die Partitur-Engine zu und ist somit unabhängig vom MDV-Subsystem. Außer der musikwissenschaftlichen Suchfunktion kann daher auch bei fehlendem MDV-Subsystem im vollem Umfang mit *JRing* gearbeitet werden.

6.2 Notizbestandteile als schlanke komplexe Fachwerte

Da ich Fachwerte im Gegensatz zu Materialien bisher weder in fachlicher noch in technischer Hinsicht explizit diskutiert habe, stelle ich die damit verbundenen Konzepte zunächst kurz vor, bevor ich auf die Probleme bei der Modellierung komplexer Fachwerte eingehe.

Fachwerte sind in WAM anwendungsspezifische Datentypen mit einem definierten Wertebereich und festgelegten Operationen (vgl. [WAM98, S. 62]). Sie sind in all jenen

Systemteilen sinnvoll, in denen es nicht um die Modellierung rein mathematischer bzw. generischer Zeichenkettenoperationen geht, sondern in denen aus dem Anwendungsbereich stammende fachliche Werte Verwendung finden, die mit den eingebauten numerischen und Zeichenkettendatentypen wie `long` und `String` nicht angemessen repräsentiert werden können.

- Beispielsweise ist eine Bankleitzahl zwar ein numerischer Wert, der im Wertebereich eines `long` unterzubringen ist, aber zum einen ist ihr Wertebereich von `100 000 00` bis `999 999 99` eingegrenzt und zum anderen sind auf ihr andere Operationen als auf einer Ganzzahl definiert. So sind z.B. einerseits Divisionen fachlich widersinnig, während es andererseits sinnvoll sein kann, Operationen zur Verfügung zu stellen, die anhand der ersten Stellen Auskunft über die zuständige Landeszentralbank geben.

- Obwohl musikwissenschaftliche Abstraktionen wie tonale Intervalle als `String` repräsentiert werden können (`* K2 G3 g2 g2 k2 g2 p5 P4 G2 G2 K2 k2 g2`; siehe Abb. 3.4-3 auf S. 72) sind die für Zeichenketten üblichen Operationen `toUpper` und `toLower` nicht sinnvoll, da die Groß- und Kleinbuchstaben in der gewählten Repräsentation darüber Auskunft geben, ob es sich um auf- oder absteigende Intervalle handelt. Stattdessen sind Operationen notwendig, mit denen die Ähnlichkeit mit anderen Intervallen ermittelbar ist oder mit denen festgestellt werden kann, ob eine bestimmte Melodie das Intervall beinhaltet.

- Geldbeträge können zwar beliebig groß sein, sind aber auf zwei Nachkommastellen begrenzt. Geldbeträge können weder miteinander multipliziert noch durcheinander geteilt werden und die meisten komplexen mathematischen Funktionen wie `sin`, `cos`, etc. sind nicht nützlich. Benötigt werden dementgegen Operationen, mit denen auf eine bestimmte Dezimalstelle gerundet werden kann.

Abgesehen von solchen Fachwert-spezifischen Operationen muss jeder Fachwert über die folgenden Operationen verfügen (vgl. [WAM98, S. 318]):

- Das Prädikat `istGueltig` gibt zu einem numerischen oder Zeichenkettenargument zurück, ob es als gültiger Fachwert geeignet ist oder nicht. Wie auch bei den Fachwert-spezischen Operationen darf hier nur kontextfrei geprüft werden, d.h. ohne auf anwendungsspezifisches Wissen zurückzugreifen, das über die reine Datentypdefinition hinausgeht. Der Geldbetrag `DM 12.345,67` muss daher vom Fachwert `Geldbetrag` als gültig akzeptiert werden, auch wenn das Buchungswerkzeug im Kontext einer bestimmten Kontonummer feststellt, dass eine Abbuchung in dieser Höhe aufgrund des bisherigen Kontostandes und des Dispokreditrahmens unzulässig ist.

- Mit `hatEndlichVieleWerte` kann zwischen Fachwerten mit unendlichem (`Geldbetrag`, `tonalesIntervall`) und endlichem Wertebereich unterschieden werden (`Bankleitzahl`). Die Werte endlicher Fachwerte können mit `gibAlleWerte` ausgelesen werden, z.B. um sie in einer Auswahlliste darzustellen.

- Ebenfalls zu Darstellungszwecken existiert die Funktion `gibAlsZeichenkette`.

Sowohl fachlich als auch technisch bilden Fachwerte in der herkömmlichen Modellierung im WAM-Ansatz die Endknoten von Aggregations- bzw. Verweisstrukturen (vgl. [WAM98, S. 323]):

- *Fachlich* sind sie die kleinsten Einheiten, mit denen Werkzeuge und Automaten arbeiten und aus denen Materialien bestehen. In *JRing* bestehen Notizen beispielsweise bis auf das graphische Exzerpt ausschließlich aus als Fachwert modellierten Notizbestandteilen. Falls ein Fachwert wie beispielsweise `Uhrzeit` sich aus mehreren Bestandteilen zusammensetzen lassen sollte, so sind die einzelnen Bestandteile stets wieder Fachwerte – z.B. `Stunde` und `Minute` (vgl. [WAM98, S. 318]).

- *Technisch* sind Fachwerte als einfache, leichtgewichtige Kapseln um eingebaute Datentypen realisiert und ebenso wie diese sind sie Endpunkte von Objektgeflechten. Falls Fachwerte mit einer Kombination der Entwurfsmuster Fliegengewicht und Fabrik umgesetzt sind (vgl. [WAM98, S. 320f] und siehe S. 88), bestehen allerdings weitere Verweise auf die Fabrik und die einzelnen Fliegengewichte. In beiden Fällen handelt es sich jedoch um rein interne Aggregate wie z.B. private Klassenvariablen, die keine Zielpunkte außerhalb des Fachwert-Bereichs besitzen. Zugriffe auf externe Subsysteme sind nicht vorgesehen.

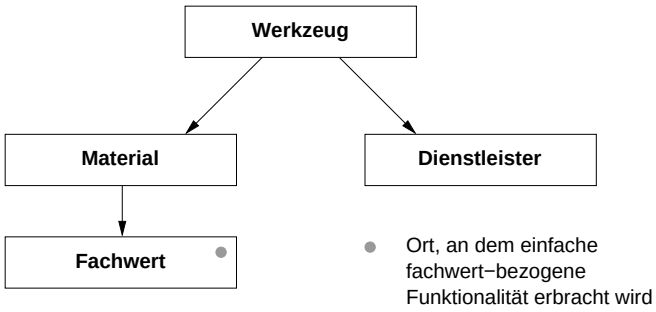


Abb. 6.2-1 Schematisches Verhältnis von Werkzeugen, Materialien, Fachwerten und Dienstleistern bei herkömmlicher Modellierung.

Während diese herkömmliche Modellierung fachlich stets sinnvoll ist, so stellt sie technisch gesehen nur dann eine befriedigende Lösung dar, wenn sie auf *einfache Fachwerte* angewendet wird, wie es in den in der WAM-Literatur gewählten Beispielen ausnahmslos der Fall ist. Diese einfachen Fachwerte zeichnen sich durch zwei Eigenschaften aus:

1. **Ihr Wertebereich ist entweder sehr klein oder unendlich.** Einerseits gibt es bei Wochentags- und Monatsnamen nur sieben bzw. zwölf mögliche Werte, während sich andererseits bei Geldbeträgen, Kontonummern, Zinssätzen und Jahreszahlen keine Begrenzung des Wertebereichs angeben lässt (vgl. [WAM S. 317ff]).

2. **Ihre speziellen Operationen sind sehr einfach.** Die Vergleichsoperationen lassen sich – wie bei den obigen Beispielen – in wenigen Schritten auf Operationen der eingebauten Datentypen wie `==`, `!=`, `<=`, `<`, `>`, `>=` sowie `startsWith`, `endsWith` oder `subString` zurückführen.

Diesen einfachen Fachwerten stehen komplexe Fachwerte gegenüber, deren Wertebereiche zwar sehr groß aber endlich bzw. deren Operationen aufwendig sind:

- *Sehr großer endlicher Wertebereich: Bankleitzahlen*. Der Wertebereich umfasst die Bankleitzahlen aller deutscher Kreditinstitute. Es kann somit kontextfrei im Rahmen von `istGueltig` geprüft werden, ob beispielsweise die Nummer 200 691 40 zu einem ein Kreditinstitut gehört oder nicht. Algorithmisch sind Bankleitzahlen hingegen nicht aufwendiger als einfache Fachwerte.

- *Aufwendige Operationen: musikwissenschaftliche Abstraktionen*. Wie in Abschnitt 3.2.1 diskutiert, sind musikwissenschaftliche Vergleichsoperationen sehr komplex. Die Abstraktionen ähneln einfachen Fachwerten aber darin, dass sie wie beispielsweise Geldbeträge einen unendlichen Wertebereich besitzen

- *Sehr großer endlicher Wertebereich oder aufwendige Operationen: Kalenderdaten.* Ein Datum ist nur mittelmäßig komplex, solange es – wie in der WAM-Literatur – nur Standardoperationen wie `istGueltig` besitzt, mit der z.B. 31.12. von 31.11., 29.2.1996 von 29.2.1997 sowie 29.2.2000 von 29.2.1900[3] abgrenzbar ist. Sollen aber z.B. für einen betrieblichen Urlaubsplaner Operationen wie `istFeiertag` genutzt werden, so wird ein Datum zum komplexen Fachwert. Diese Komplexität kann auf zweierlei Weise realisiert werden:

  1. *Komplexe kalendarische und astronomische Berechnungen*. Der Ostersonntag fällt beispielsweise auf den ersten Sonntag nach dem ersten Frühlingsvollmond.

  2. *Sehr großer Wertebereich*. Alle Feiertage liegen über einen sehr langen Zeitraum tabelliert vor, so dass für `istFeiertag` keine Berechnung angestellt, sondern eine Wertebereichsprüfung für Feiertage durchgeführt wird.

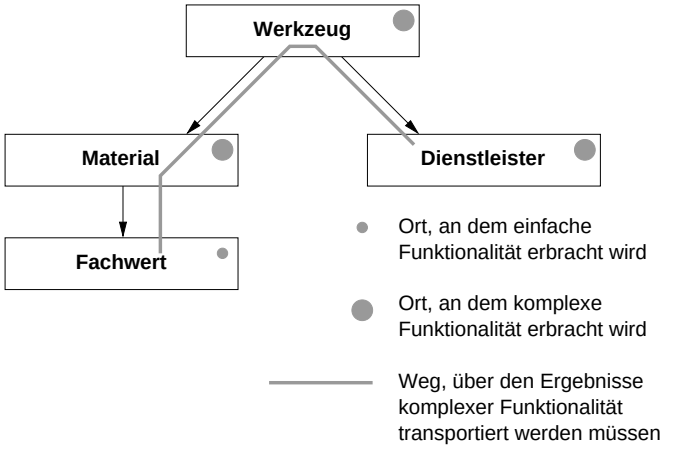
Begriff 6.2 komplexer Fachwert *(complex domain value)*

Ein komplexer Fachwert hat eine sehr große, aber endliche Wertemenge und / oder bietet Operationen an, deren Ausführung aufwendiger Algorithmen bedarf.

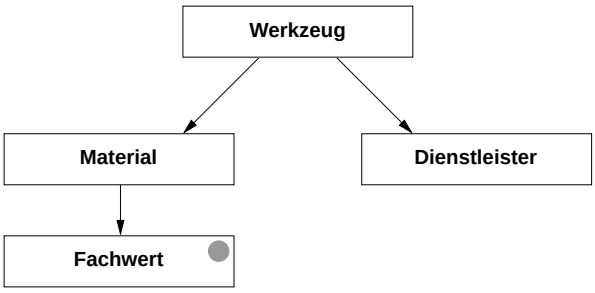
Je nachdem, in welchem Maße die Eigenschaften einfacher Fachwerte beibehalten werden sollen, ergeben sich verschiedene Lösungsmöglichkeiten, um den Besonderheiten komplexer Fachwerte Rechnung zu tragen:

[3] Nach dem Gregorianischen Kalender ist jedes durch 4 teilbare Jahr ein Kandidat für ein Schaltjahr. Ist das Jahr zugleich durch 100 teilbar, so ist es aber kein Schaltjahr, außer es ist auch durch 400 teilbar. Deswegen ist das Jahr 2000 ein Schaltjahr, 1900 aber nicht.

- Sollen auch komplexe Fachwerte als leichtgewichtige Endknoten von Objektgeflechten erhalten bleiben, so muss die komplexe Funktionalität in die enthaltenden Werkzeuge, Automaten oder Materialien ausgelagert werden. Beispielsweise könnte die Klasse `Kalenderdatum` als schlanke Kapsel um die Klasse `Date` aus der Java-API realisiert werden, bei der die komplexe Funktionalität für `istFeiertag` vom Werkzeug erbracht wird. Dies führt aber sowohl zu einer Vermischung von kontextfreien und kontextsensitiven Operationen als auch zu Redundanzen, sofern mehr als ein Werkzeug, Automat oder Material den Fachwert benötigt.



- Wird das Kriterium der Leichtgewichtigkeit fallen gelassen, können komplexe Fachwerte ihre Funktionalität vollkommen selbständig implementieren. Als Endknoten von Objektgeflechten bleibt es ihnen jedoch weiterhin unmöglich, diese Funktionalität aus Subsystemen zu beziehen, so dass ihre Implementation erhebliche Ausmaße annehmen kann. Im Beispiel der musikwissenschaftlichen Abstraktionen müsste dabei die aufwendige Vergleichsfunktionalität komplett in jedem Fachwert neu implementiert werden.



In beiden Fällen ist die Realisierung von komplexen Fachwerten mit sich in großen Abständen ändernden Wertebereichen (neue Bankleitzahlen) oder Berechnungsformeln problematisch („Tag der deutschen Einheit“ vor und nach 1990). Für diese Änderungen müssen die Fachwert-Klassen erneut geöffnet werden, obwohl sich weder die fachlichen Tätigkeiten der Anwender noch die technischen Anforderungen geändert haben.

Wir die Forderung fallengelassen, dass Fachwerte nicht nur in fachlicher, sondern auch in technischer Hinsicht Endpunkte von Verweisstrukturen sein sollen, dann ergibt sich mit *schlanken komplexen Fachwert* eine dritte Lösungsmöglichkeit, die nicht mit den bisher diskutierten Nachteilen behaftet ist.

6.2.1 Schlanke komplexe Fachwerte

Schlanke komplexe Fachwerte sind ebenso wie einfache Fachwerte leichtgewichtig. Im Gegensatz zu ihnen bilden sie jedoch nicht die Endknoten von Objektgeflechten, sondern bedienen sich – ebenso wie schlanke graphisch komplexe Materialien – eines separaten Dienstleisters, um ihre Schnittstelle zu erfüllen. Die Fachwerte leiten die an sie gerichteten Anfragen lediglich in modifizierter Form an einen Dienstlei-

ster um und bereiten dessen Ergebnisse so auf, dass sie an die Klienten zurückgegeben werden können. Aus Sicht des komplexen Fachwerts ist es belanglos, ob der aufgerufene Dienstleister seine Funktionalität selbst implementiert oder seinerseits auf einen weiteren internen Dienstleister oder ein externes Subsystem zurückgreift, da dieser weitere Zugriff durch den Dienstleister vollständig gekapselt wird. Um Fachwert und Dienstleister maximal zu entkoppeln, greifen Fachwerte nicht direkt auf den Dienstleister zu, sondern wenden sich zunächst an die ihnen ohnehin bekannte Fachwert-Fabrik (siehe S. 163). Nur diese Fabrik ruft Methoden am Dienstleister auf.

Begriff 6.3 schlanker komplexer Fachwert *(lightweight complex domain value)*

Ein komplexer Fachwert ist schlank, wenn er für die Ausführung mindestens eine seiner Operationen für seine Klienten unsichtbar auf separaten Dienstleister zurückgreift. Falls der Fachwert über eine Fabrik erzeugt wurde, geschieht der Zugriff auf den Dienstleister indirekt über diese Fabrik.

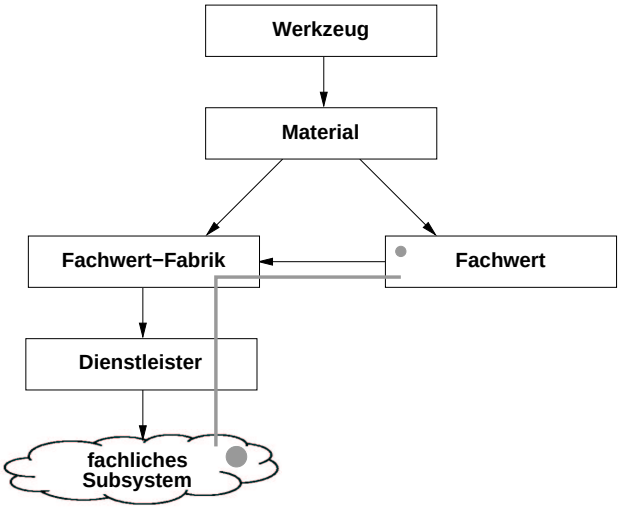


Abb. 6.2-2 Schematisches Verhältnis von schlanken komplexen Fachwerten zu Werkzeugen, Materialien, Dienstleistern und Subsystemen.

Schlanke komplexe Fachwerte haben die folgenden *Vorteile*:

- Sie sind trotz ihrer Komplexität leichtgewichtig.
- Ihre fachliche Einheit bleibt erhalten, da komplexe, aber kontextfrei ausführbare Operationen nicht von den Klienten übernommen werden müssen.
- Aus fachlicher Sicht ist ein komplexer nicht von einem einfachen Fachwert zu unterscheiden, da der interne Zugriff auf den systeminternen Dienstleister – wie auch bei graphisch komplexen Materialien – komplett verborgen wird.
- Ändert sich der komplexe Faktor, muss die Fachwert-Klasse nicht geöffnet und modifiziert werden, sondern lediglich der die Funktionalität tatsächlich realisierende Dienstleister. Stützt dieser sich auf ein externes Subsystem, dann kann auch der Dienstleister und damit der gesamte Quelltext unverändert bleiben.

- Ähnliche komplexe Fachwerte können ggf. auf einen einzigen Dienstleister zurückgreifen, wodurch sich der Änderungsaufwand zusätzlich reduziert.

Schlanke komplexe Fachwerte haben zwei *Nachteile*:

- Wenn der aufgerufene Dienstleister die Anfragen letztendlich an ein externes Subsystem delegiert, dann können unerwünschte Plattformabhängigkeiten entstehen. Ein Ausfall des externen Subsystems kann sich bis auf die Fachwerte auswirken.

- Wie bei graphisch komplexen Materialien ruft eine in herkömmlichen WAM-Systemen „passive“ Modellierungseinheit nun eigenständig Dienstleister auf. Dies erscheint jedoch akzeptabel, da sich aus der für WAM primären, fachlichen Sicht nichts ändert und Fachwerte auch weiterhin von sich aus keine anderen Werkzeuge oder Materialien in ihrem Zustand verändern.

6.2.2 Anwendung innerhalb *JRings*

Sämtliche Notizbestandteile, die musikwissenschaftliche Abstraktionen darstellen, sind in *JRing* als Einschübe realisiert, die schlanke komplexe Fachwerte verkörpern. Die komplexen Operationen dieser Fachwerteinschübe rufen Operationen eines im Einschubrahmen *„Notizbestandteile“* vorhandenen internen Dienstleisters auf, der Anfragen – sofern im Einschubrahmen *„MDV-Subsystem“* ein Einschub vorhanden ist – an den eigentlichen abstrakten Dienstleister dieses Einschubrahmens weiterleitet. Diese Indirektion ist notwendig, da ansonsten im Einschub-Ansatz unerlaubte direkte Zugriffe auf Einschübe aus einen Nicht-Zwangseinschubrahmen erfolgten (siehe Abschnitt 5.1.4). Ist kein MDV-Subsystem-Einschub vorhanden, dann sind die entsprechenden komplexen Operationen der Fachwerte nicht verfügbar. Einfache Operationen wie das Auslesen des Wertes oder Vergleiche auf Gleichheit können jedoch weiterhin ausgeführt werden, da sie ausschließlich mit Zeichenketten-Operationen auf der internen Repräsentation realisierbar sind. Da einfache Fachwerte wie `Partiturposition`, `Name`, `Kommentar` und `Farbe` ohnehin als einfache Fachwerte modelliert sind, können sämtliche auf Notizen arbeitenden Werkzeuge auch bei fehlendem MDV-Subsystem-Einschub weiterhin arbeiten:

- *Notizeditoren und Notizbetrachter* sind uneingeschränkt verwendbar, da die komplexen musikwissenschaftlichen Fachwerte von ihnen lediglich mit Hilfe von deren einfachen Operationen angezeigt werden.

- *Notizauswähler und Markierer* sind in ihrer Funktionalität eingeschränkt, da zum Vergleichen bzw. zum Extrahieren musikwissenschaftlicher Abstraktionen komplexe Funktionalität aus einem MDV-Subsystem notwendig ist. Einfache Operationen komplexer Fachwerte und sämtliche Operationen einfacher Fachwerte stehen aber weiterhin zur Verfügung.

6.3 Zusammenfassung

In diesem Kapitel habe ich zunächst die in der allgemeinen softwaretechnischen Literatur vorgeschlagenen Entwurfsmuster zur Kapselung von Subsystemen diskutiert und dann dargestellt, wie die daraus resultierenden Kapseln im Kontext des Werkzeug- und Material-Ansatzes an die fachlich motivierten Systembestandteile angebunden werden können: Subsysteme werden dabei ausschließlich von den unmittelbar handhabbaren Systembestandteilen (Werkzeugen und Automaten) referenziert, während Materialien und Behälter keinen direkten Zugriff auf Subsysteme besitzen. Sollten Materialien und Behälter Daten oder Funktionalität aus Subsystemen benötigen, so müssen sie von den sie bearbeitenden Werkzeugen und Materialien beliefert werden.

In den Abschnitten 6.1 und 6.2 habe ich jeweils für Materialien und Fachwerte herausgearbeitet, dass diese Art der Kapselung problematisch ist, wenn die komplexe Funktionalität bzw. die komplexen Daten des Subsystems sich nicht auf die Werkzeuge und Automaten abbilden lassen, sondern aus fachlicher Sicht vielmehr den Materialien bzw. deren Fachwerten zuzuordnen sind:

- Bei *graphisch komplexen Materialien* wie mittelalterlichen Handschriften, Stadtplänen und Partituren stößt das WAM-übliche Verfahren, die Präsentation der Materialien ad hoc ausschließlich anhand des fachlichen Materialzustands im Werkzeug zu generieren, an seine Grenzen. Die technische Trennung der fachlich motivierten Materialien in einen in der Materialklasse angesiedelten funktionalen Teil und einen in der Werkzeugklasse angesiedelten präsentationsbezogenen Teil fällt hier besonders ins Gewicht, da die bereits vor dem Start des Anwendungssystems generierten Präsentationsdaten mit hohem technischen Zusatzaufwand im Werkzeug mit dem Materialzustand synchronisiert werden müssen.

- Bei *komplexen Fachwerten* wie Bankleitzahlen, Kalenderdaten und musikwissenschaftlichen Abstraktionen sprengt der Implementationsumfang den von normalen Fachwerten um ein Vielfaches, da es bei ihnen nicht ausreicht, einfache Kapseln um eingebaute Datentypen wie `int`, `long` oder `String` zu konstruieren. Das bisher in WAM übliche Verfahren, die Fachwerte weiterhin schlank zu implementieren, führt bei komplexen Fachwerten dazu, dass erhebliche Teile der kontextfreien Funktionalität, die aus fachlicher Sicht eindeutig dem Fachwert zuzuordnen ist, aus technischen Gründen im einbettenden Material oder Werkzeug erbracht wird.

Mit den von mir diskutierten Konstruktionsansätzen für schlanke graphisch komplexe Materialien (Abschnitt 6.1.1) und schlanke komplexe Fachwerte (Abschnitt 6.2.1) habe ich eine Lösung vorgelegt, mit der es möglich ist, die technische Vermischung von fachlich getrennten Arbeitsmitteln zu verhindern. In beiden Fällen wird hierbei die komplexe Funktionalität nicht in das einbettende Arbeitsmittel ausgelagert, sondern die Materialien und Fachwerte greifen unmittelbar und ohne Vermittlung von Werkzeugen und Automaten auf Subsysteme zu, welche die komplexe Funktionalität bereitstellen. Um die Materialien und Fachwerte nicht an die spezielle, ggf. proprietäre, plattformabhängige oder in einer anderen Programmiersprache vorliegende Schnittstelle eines Subsystems zu binden, wird das eigentliche Subsystem dabei durch einen fachlichen Dienst-

leister gekapselt, der die Funktionalität des konkreten Subsystems in abstrakter Form zur Verfügung stellt. Bei schlanken komplexe Materialien heißt dieser fachliche Dienstleister Präsentationsversorger. Ihm wird vom Material lediglich eine Kennung des anzuzeigenden Materialbereichs in Form von Rasterkoordinaten übergeben, aus denen er dann selbständig die passenden Präsentationsdaten aus dem nur ihm bekannten Subsystem bezieht und diese in eine Form konvertiert, die das Material problemlos an die es darstellenden Werkzeuge weiterreichen kann.

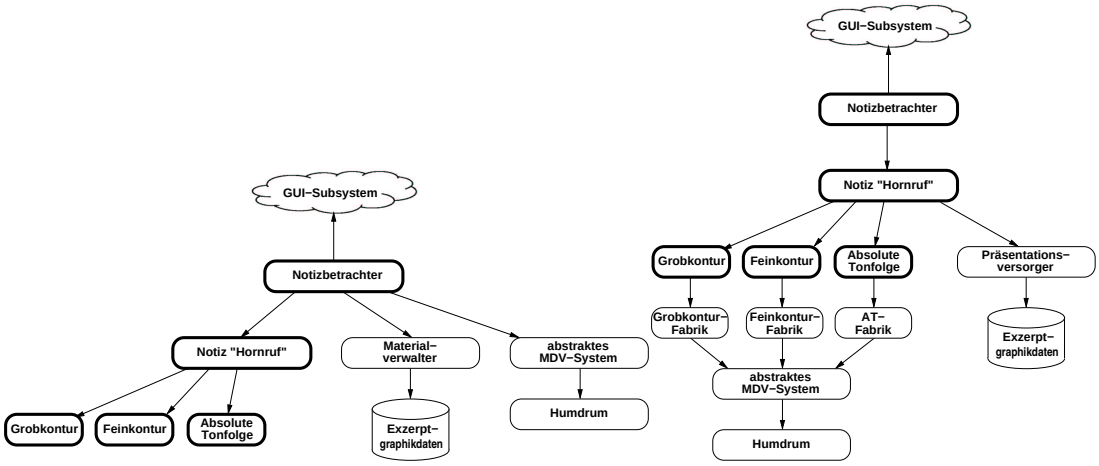


Abb. 6.3-1 Gegenüberstellung der herkömmlichen und der schlanken Modellierung von Materialien und Fachwerten am Beispiel des *JRing*-Materials einer Notiz.

Die aus Werkzeug, Material und Fachwerten bestehende fachliche Struktur ist in beiden Modellierungsvarianten identisch. Sie ist durch starke Umrandung hervorgehoben.

Die schlanke Konstruktion von graphisch komplexen Materialien und komplexen Fachwerten hat gegenüber der herkömmlichen Konstruktionsweise den Vorteil, dass die fachliche Einheit der Materialien und Fachwerte gewahrt bleibt und Daten und Funktionsaufrufe nicht aus rein technischen Gründen und unter Bruch der fachlichen Kapselung von den einbettenden Werkzeugen durchgereicht werden müssen. Auf diese Weise werden sowohl Werkzeuge von ihnen fachlich ohnehin nicht zugeordneter Funktionalität entlastet und die Schnittstelle von Materialien und Fachwerten kann von einem beträchtlichen Teil rein technisch motivierter Operationen befreit werden, der bei herkömmlicher Konstruktionsweise für die externe „Befüllung“ durch die einbettenden Arbeitsmittel notwendig wäre.

Da wie auch bei der herkömmlichen Modellierung die Subsysteme, welche die komplexe Funktionalität bzw. die komplexen Daten implementieren, durch geeignete Dienstleister gekapselt sind, bleiben die diesbezüglichen Vorteile der herkömmlichen Konstruktionsweise erhalten: Ein Dienstleister kann von mehreren voneinander unabhängigen Klienten genutzt werden und die konkreten Subsysteme können ausgetauscht werden, ohne dass die Klienten der Dienstleister angepasst werden müssen. Obwohl Materialien und Fachwerte bei schlanker Modellierung von sich aus auf Subsysteme zugreifen, entsteht kein Widerspruch zu der passiven Rolle, die beide in WAM-Anwendungssystemen einnehmen, da sie nicht von sich aus aktiv werden, sondern nur als Reaktion auf Anfragen vom einbettenden Arbeitsmittel.

In Verbindung mit dem Einschub-Ansatz können komplexe Subsysteme auch sitzungsweise flexibel und auf für die Anwender verständliche Weise von den Anwendern ausgetauscht werden, da jeder Bestandteil des Modells eines Einschubrahmens mit einem Bestandteil des Modells eines schlanken graphisch komplexen Materials bzw. eines schlanken komplexen Fachwerts korrespondiert.

Einschubrahmen	schlankes, graphisch komplexes Material	schlanker komplexer Fachwert
Kunde	graphisch komplexes Material	komplexer Fachwert
Einschubexemplaranbieter	Präsentationsversorger	fachlicher Dienstleister
Einschub	komplexes Subsystems	komplexes Subsystem

Wird beispielsweise das graphisch komplexe *JRing*-Material Partitur mit dem Einschub-Ansatz verknüpft, so ergibt sich die folgende Konstruktion, die eine vereinfachte Konkretisierung des in Abb. 5.1-13 auf S. 140 abgebildeten Modells eines allgemeinen Einfacheinschubrahmens darstellt.

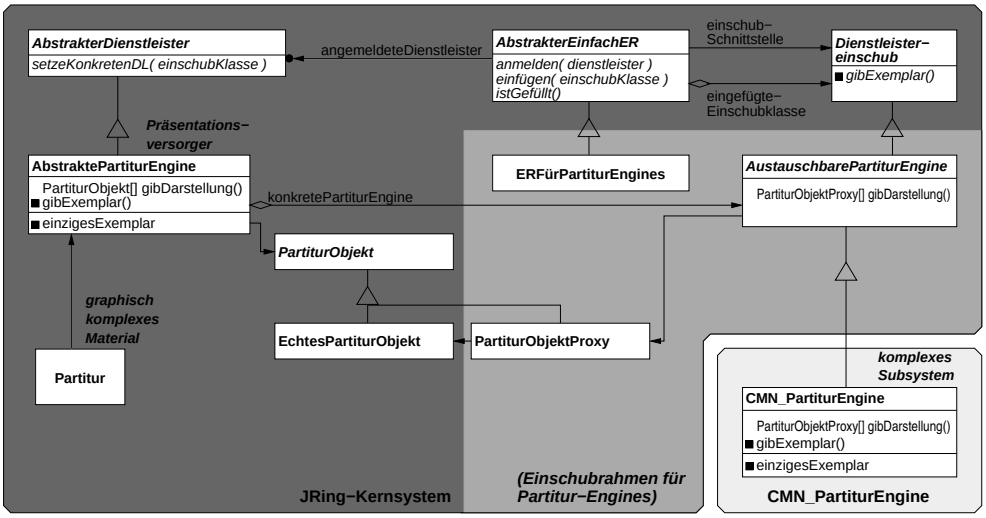


Abb. 6.3-2 Der Einschubrahmen „Partitur-Engine“ des *JRing*-Systems.

Schließlich habe ich in Abschnitt 6.2.2 gezeigt, dass die Kombination des Einschub-Ansatzes mit der schlanken Konstruktionsweise von Fachwerten der Schlüssel zur Lösung des bereits in Kapitel 3 aufgeworfenen Konstruktionsproblems ist, das darin besteht, dass das *JRing*-System auch dann noch sinnvoll nutzbar sein soll, wenn aus Gründen knapper Ressourcen kein MDV-Subsystem-Einschub verfügbar ist. Da nicht die gesamte, sondern nur die komplexe Funktionalität der komplexen Fachwerte durch das als Einschub realisierte komplexe Subsystem erbracht wird, steht die einfache Funktionalität der Fachwerte auch dann noch zur Verfügung, wenn MDV-Subsystem-Einschübe fehlen.

7 Zusammenfassung und Ausblick

Ausgehend von dem wiederholt in der Musikwissenschaft vorgetragenen Wunsch nach effektiver, flexibler Rechnerunterstützung bei einer Vielzahl von Analysearten habe ich in dieser Dissertation sowohl softwaretechnische als auch musikdatenverarbeitungsbezogene Lösungen erarbeitet, um die in der Literatur diskutierten Probleme zu überwinden:

1. Aufgrund der Verschiedenartigkeit der Analyseaufgaben existiert keine einheitliche Grundlage, auf der ein flexibel anpassbares Analysesystem erstellt werden könnte. Stattdessen gibt es nur eine Vielzahl kleiner, fachlich und technisch sehr eingeschränkt einsetzbarer und untereinander inkompatibler Spezialsysteme.

2. Für eine flexible, sitzungsweise Anpassung eines einheitlichen Analysesystems durch dessen Anwender gibt es keinen geeigneten softwaretechnischen Modellierungsansatz. Bei Rahmenwerk-Ansätzen führt ein Streben nach maximaler Flexibilität zum *fatware*-Problem und die nach Komponenten-Ansätzen erstellten binären Auslieferungseinheiten sind zu technisch, als dass sie von den Anwendern auf verständliche Weise gehandhabt und durch *glue code* zusammengefügt werden könnten.

3. Obwohl der anwendungsorientierte Werkzeug und Material-Ansatz gut zur Modellierung von Analysesystemen geeignet ist, sind die musikwissenschaftlichen Materialien und Subsysteme so komplex, dass eine Fortschreibung der vorgeschlagenen Modellierungsrichtlinien zu massiven Kapselungsproblemen führt, da die Werkzeuge die komplexe Funktionalität, die sie an den Materialien abrufen, zuvor selbst erbracht haben müssen.

Mit den von mir herausgearbeiteten Ergebnissen ist die Lösung aller drei Problemkomplexe und somit auch die Realisierung des erwünschten Analysesystems möglich:

1. Anhand einer detaillierten fachlichen und Analyse nach WAM sowie einer Untersuchung der verfügbaren Musikdatenverarbeitungstechnik habe ich habe gezeigt, dass sich so unterschiedliche Analysearten wie die thematische, die leitmotivische und die Set-Theory-Analyse mit einem einzigen, flexibel anpassbaren Kernsystem adäquat unterstützen lassen. Dieses Kernsystem bietet Such-, Vergleichs- und Verwaltungswerkzeuge, mit denen Partituren und Notizen bearbeitbar sind, für die ich mit einer kanonischen Partiturdarstellung und expliziter Aufnahme musikwissenschaftlicher Abstraktionen auf den Notizzettel die Grundlage für eine rechnergestützte Handhabung gelegt habe. Analysesysteme müssen somit nicht mehr von Grund auf neu spezifiziert und entwickelt werden, sondern sind als in einigen Komponenten unterschiedliche Varianten dieses Kernanalysesystems formulierbar.

2. Indem ich die Eigenschaften von technischen Geräten mit Einschüben und Einschubrahmen in Form einer Metapher auf Anwendungssysteme übertragen habe, habe ich eine Basis geschaffen, auf der sich Rahmenwerk- und Komponenten-Ansätze auf eine solche Weise kombinieren lassen, dass sich deren Nachteile aufheben. Die Einschubrahmen des Kernsystems geben dabei eine genaue Schnittstelle vor, anhand derer ersichtlich ist, welche Funktionalität Einschübe erbringen müssen und welche Funktionalität des Kernsystems sie in Anspruch nehmen können. Anwender können Einschubsysteme mit nach fachlichen Kriterien zugeschnittenen Einschubrahmen mit geeigneten Einschüben sitzungsweise auf einfache Weise ihren aktuellen Anforderungen anpassen, da ihnen sowohl die Handhabung der Einschübe als auch deren Funktionalität verständlich ist.

   Die technischen Probleme von reinen Rahmenwerk- oder Komponenten-Ansätzen werden vermieden, wenn das Kernsystem mit Rahmenwerk- und die Einschübe mit Komponententechnologien realisiert werden: Der im Kernsystem festgelegte Kontrollfluss macht individuell programmierten *glue code* zum Zusammenfügen von Komponenten überflüssig. Die in Form von Einschubrahmen fest vorgegebenen Schnittstellen schließen funktionale Überlappungen und komplexe Abhängigkeitsgeflechte zwischen Komponenten aus. Die separate Auslieferbarkeit von Komponenten verhindert die Bildung von *fatware* und eröffnet zugleich die Möglichkeit, Teile von Anwendungssystemen entlang der von den Einschubrahmen vorgegebenen Schnittstellen unabhängig voneinander zu entwickeln und beliebig zu erweitern. Das Prinzip der Anwendungsorientierung kann dadurch, dass die Anwender Anwendungssysteme somit sitzungsweise funktional umkonfigurieren und sogar eigenständig erweitern können, auf die Laufzeit ausgedehnt werden.

3. Zur Behebung der Kapselungsprobleme komplexer Materialien und Subsystemen in WAM habe ich schlanke graphisch komplexe Materialien und schlanke komplexe Fachwerte entwickelt. Eine Versorgung von Materialien und Fachwerten erfolgt dabei nicht mehr durch die sie nutzenden Arbeitsmittel, sondern Materialien und Fachwerte greifen unmittelbar auf geeignete Dienstleister zu. Die technischen Details des konkreten komplexen Versorgungsmechanismus werden somit vollkommen gekapselt. Technische Änderungen an graphisch komplexen Materialien und komplexen Fachwerten ziehen keine Änderungen in anderen Systemteilen nach sich.

Anhand des in Java mit *JWAM* bis hin zur Produktreife implementierten *JRing*-Einschubsystems zur Unterstützung der musikwissenschaftlichen Analyse habe ich die Ergebnisse aus allen drei Bereichen konstruktiv miteinander verknüpft und validiert. Neben der gewählten Auslieferung von Einschüben und Kernsystem samt Konfigurationsdesktop und Einschubmanager in JARs habe ich dabei auch die Realisierungsoptionen CORBA und COM diskutiert. Anhand des Beispiels der als Einschub umgesetzten Partitur-Engine habe ich gezeigt, wie die Subsysteme hinter graphisch komplexen Materialien ohne Auswirkungen auf die sie benutzenden Werkzeuge ausgetauscht werden können.

Im Folgenden zeichne ich die Argumentationskette dieser Arbeit mit ihren Zwischenergebnissen nach und zeige abschließend diejenigen Stellen auf, an denen sich Anknüpfungspunkte für weitere Forschungsarbeiten ergeben.

Am Beispiel von Bildbearbeitungssystemen, Web-Browsern und Simulationssystemen habe ich zunächst in Kapitel 2 das für diese Arbeit zentrale Konzept der Anwendungsfamilie herausgearbeitet und gegen ähnliche Konzepte aus dem Bereich der erstmals von Parnas diskutierten „Programmfamilien“ abgegrenzt. Eine Anwendungsfamilie ist im Sinne dieser Arbeit eine Menge von Anwendungssystemen eines Herstellers, die verschiedene Ausführungen ein und derselben chronologischen Version einer Anwendung darstellen. Sie besteht aus einem Kernsystem, das allen Familienmitgliedern gemein ist, und Komponenten, welche die Funktionalität enthalten, hinsichtlich derer sich die einzelnen Mitglieder unterscheiden. Andere „Programmfamilien“, die Produkte unterschiedlicher Hersteller oder Versionen zusammenfassen bzw. über kein gemeinsames Kernsystem verfügen, wurden nicht weiter betrachtet.

Anschließend habe ich die Struktur von Anwendungsfamilien näher betrachtet und dabei insbesondere untersucht, mit welchen Arten von Komponenten welche Arten von Anpassungen des Kernsystems möglich sind und dazu den Begriff der Anpassungsstelle eingeführt und ausdifferenziert:

- Diejenigen Stellen des Kernsystems, die mit Komponenten angepasst werden können, die höchstens einmal im Anwendungssystem vorkommende zentrale Dienste verkörpern, sind je nachdem, ob der zentrale Dienst für die Lauffähigkeit des Anwendungssystems optional oder essentiell ist, entweder *Einfachanpassungsstellen* oder *Zwangsanpassungsstellen*.

- Stellen des Kernsystems, an denen das Kernsystem mit beliebig vielen gleichartigen, bausteinähnlichen Komponenten angepasst werden kann, sind *Mehrfachanpassungsstellen*.

Abschließend habe ich Anpassungsstellen kritisch mit ähnlichen Konzepten aus der Literatur verglichen und sowohl für *dimensions of extensibility* (vgl. [Szy98]) und *hot spots* (vgl. [Pre97]) gezeigt, dass sie für Anwendungsfamilien problematisch bzw. weniger mächtig sind: Szyperskis „Erweiterungsdimensionen“ sind im Widerspruch zur Anschauung nicht orthogonal und werden nicht absolut durch das Kernsystem vorgegeben, sondern entstehen erst durch konkrete Komponenten. Prees Hot-Spots vermögen nur Einfachanpassungsstellen auszudrücken. Beide Konzepte sagen nichts über Zwangsanpassungsstellen aus.

Aufgrund ihrer besonders hohen Flexibilitätsanforderungen, bei der alle drei Arten von Anpassung zum Tragen kommen, habe ich als Hauptbeispiel in Kapitel 3 eine Anwendungsfamilie von musikwissenschaftlichen Analysesystemen spezifiziert. Da bestehende Analysesysteme zu wenig anwendungsorientiert und entweder zu speziell oder zu inflexibel sind, habe ich zunächst für alle drei Kontexte des Software-Entwicklungsprozesses gezeigt, (1) dass sich Analysesysteme anwendungsorientiert gestalten lassen und dass (2) sie sich trotz unterschiedlicher Analysearten, Partiturarten und verwendeter Musikdatenverarbeitungssysteme als eine einzige Anwendungsfamilie mit Anpassungsstellen und Komponenten auffassen lassen.

(1) Um *Anwendungsorientierung* zu erreichen, habe ich im Kontext *Anwendungsbereich* auf der Basis einer nach WAM durchgeführten Analyse zunächst die Arbeits-

mittel und Tätigkeiten dreier Analysearten identifiziert. Im Kontext *Handhabung & Präsentation* habe ich die Rolle des geeigneten Leitbilds diskutiert und gezeigt, dass das WAM-Leitbild des *Arbeitsplatzes für eigenverantwortliche Expertentätigkeit* besser zur musikwissenschaftlichen Analyse passt, als bestimmte ablaufsteuernde und technische Leitbilder. Da bei Musikwissenschaftlern allerdings kein technisches Vorwissen vorausgesetzt werden kann und die Analyse nur einen Teil musikwissenschaftlicher Tätigkeiten ausmacht, habe ich das spezialisierte Leitbild *Werkzeugunterstützung für eigenverantwortliche Expertentätigkeit* eingeführt. Bei ihm muss auch die Konfiguration des Systems besonders anwenderfreundlich durchführbar sein und Arbeitsergebnisse in Softwaresysteme außerhalb des Analysesystems transferiert werden können. Das im Kontext *verwendete Technik* angesiedelte Problem bisheriger Analysesysteme, die Partituren mangels Daten nur in reduzierter Form graphisch darstellen zu können, wird durch das in meiner Diplomarbeit erstellte *scr2hmd* gelöst, mit dem der größte Teil weltweit existierender Notensatzdaten als Datenquelle nutzbar gemacht wird.

Im Kern stellt ein anwendungsorientiertes Analysesystem die Materialien Partitur, Notiz und Notizkatalog zur Verfügung, die mit Werkzeugen zum Exzerpieren, Suchen, Vergleichen und Verwalten bearbeitet werden können. Eine vollautomatische Analyse ist aufgrund der Komplexität der Kompositionsregeln, der Notenschrift, der menschlichen Sprache und der Größe der Werke nicht möglich. Automatisch ermittelte Suchergebnisse können lediglich als Vorschläge gelten, über deren Annahme der Analysator individuell entscheiden können muss. Um elektronische Partituren ohne Seiten- und Systemumbrüche durchblättern und Strukturelemente kontinuierlich markieren zu können, werden sie kanonisch, das heißt stets auch mit pausierenden Stimmen dargestellt. Automatisch unterstütztes Suchen und Vergleichen wird ermöglicht, indem die bei der herkömmlichen Analyse implizit gebildeten musikalischen Abstraktionen zu expliziten Notizbestandteilen gemacht werden.

(2) Mit drei Anpassungsstellen für Notizbestandteile, Partitur-Engines und MDV-Subsysteme ist es möglich, sämtliche Unterschiede innerhalb der Anwendungsfamilie hinsichtlich der Analyseart, der Partiturart und der Systemplattform zu modellieren. Für den *Anwendungsbereich* liegen die Unterschiede bei der thematischen, der leitmotivischen und der Set-Theory-Analyse nicht in den grundlegenden Tätigkeiten und den verwendeten Arbeitsmitteln, sondern in deren Benennung und Gewichtung. Die erheblichen musiktheoretischen Unterschiede bei der Bewertung der Ergebnisse kommen dementgegen erst im kreativen Bereich der Texterstellung zum Tragen, der aufgrund seiner Komplexität ohnehin nicht durch Spezialsoftware unterstützbar ist. Um eine bestimmte Art von Analyse durchzuführen reicht es daher aus, die Notizen aus den dementsprechenden speziellen Notizbestandteilen aufzubauen, deren Vorrat durch die dem Kernsystem hinzugefügten Notizbestandteil-Komponenten festgelegt werden kann. Im Bereich *Handhabung & Präsentation* können unterschiedliche Partituren durch geeignete Partitur-Engine-Komponenten für CMN-, Mensural- oder chromatische Notation dargestellt werden und für verschiedene Systemplattformen ist es im Kontext *verwendete Technik* möglich, verschiedene plattformabhängige MDV-Subsystem-Komponenten zu verwenden. Da sich die Analyseart, die analysierte Partitur und die Plattform, auf der die Analyse durchgeführt wird, von Sitzung

zu Sitzung ändern kann, müssen die Komponenten an den drei Anpassungsstellen bei jedem Start des Systems auf einfache Weise neu umzukonfigurieren sein.

In Kapitel 4 habe ich gezeigt, welche Voraussetzungen das softwaretechnische Modell und die binären Auslieferungseinheiten einer Anwendungsfamilie erfüllen müssen, damit die gewünschte sitzungsweise Anpassung auf für die Anwender verständliche Weise durchführbar ist. In einem ersten Schritt habe ich zunächst für das softwaretechnische Modell herausgearbeitet, dass Anpassungsstellen und Komponenten in ihrer allgemeinen Form durch eine Kombination der aus [GHJV98] bekannten Entwurfsmuster Brükke, Fabrikmethode und Fliegengewicht ausdrückbar sind. Zusammen mit den zur Kommunikation zwischen Komponenten und Kernsystem notwendigen Klassen ergeben sich fünf Modellelemente: (1) Die *Komponenten-Klasse*, welche die austauschbare Funktionalität implementiert; (2) die vom Kernsystem vorgegebene *Schnittstelle*, die von allen zur Anpassungsstelle passenden Komponenten-Klasse implementiert wird; (3) *Dienstleister innerhalb des Kernsystems*, über die dessen Kunden auf die Funktionalität der Komponenten zugreifen; (4) *Zentrale Dienstleister des Kernsystems*, auf die Komponenten-Klassen zugreifen können und (5) *Parameter-Klassen*, die sowohl im Kernsystem als auch den Komponenten bekannt sind, um Daten auszutauschen.

Um zu untersuchen, auf welche Weise das spezifizierte softwaretechnische Modell auf binäre Einheiten partitioniert werden muss, damit es durch die Anwender sitzungsweise angepasst werden kann, habe ich anschließend zunächst die – beispielsweise bei Meyer und Szyperski – unzureichend voneinander abgegrenzten Rahmenwerk- und Komponenten-Konzepte herauspräpariert und begrifflich neu gefasst. Bei *Rahmenwerken* gilt es dabei, zwischen (1) der Sichtbarkeit der Implementation und (2) der Verwendung zu unterscheiden:

(1) Bei *White-Box-Rahmenwerken* ist die Implementation sichtbar während sie bei *Black-Box-Rahmenwerken* vor den Augen der Verwender verborgen ist.

(2) *Spezialisierungsrahmenwerke* werden verwendet, indem die Entwickler Unterklassen zu Schnittstellen und Klassen des Rahmenwerks bilden. Das Rahmenwerk muss dazu eine *White Box* sein. *Parametrisierungsrahmenwerke* werden dementgegen verwendet, indem die Verwender das Rahmenwerk an dafür vorgesehenen Stellen mit passenden Objekten parametrisieren. Ob die Implementation des Rahmenwerks dabei einsehbar ist (*White Box*) oder nicht (*Black Box*) spielt dabei keine Rolle.

Wenn über Komponenten gesprochen wird, muss klar sein, zu welchem Zeitpunkt sie eingesetzt werden, um eine System zu bilden:

- Klassen, Klassenbibliotheken, Spezialisierungsrahmenwerke und Makros sind im Quelltext vorliegende *Konstruktionskomponenten*, die von den Entwicklern während des Entwurfs verwendet werden.

- Prozedurbibliotheken, Parametrisierungsrahmenwerke und JavaBeans sind in Binärform vorliegende *Implementationskomponenten*, die Entwickler verwenden, um eine bestimmte Funktionalität nicht selbst implementieren zu müssen

- In DLLs, JARs, COM- und CORBA-Komponenten vorliegende dynamische Bibliotheken sind *Laufzeitkomponenten*, die erst unmittelbar vor dem Start eines Systems bzw. zu dessen Laufzeit dynamisch hinzugebunden werden, um dessen Funktionalität zu realisieren bzw. anzupassen.

Für die Auslieferung von zur Laufzeit anpassbaren Anwendungsfamilie scheiden damit Spezialisierungsrahmenwerke sowie Konstruktions- und Implementationskomponenten von vorneherein aus, da sie nur von Entwicklern zur Entwicklungszeit, nicht aber von Anwendern zur Laufzeit handhabbar sind. Doch auch eine reine Auslieferung als Parametrisierungsrahmenwerk oder als eine Menge von ungefähr gleich großen Laufzeitkomponenten muss verworfen werden: Parametrisierungsrahmenwerke sind Monolithen, die zur Laufzeit nicht erweiterbar sind und somit nur herstellerseitig, nicht aber durch die Anwender anpassbar sind. Technisch entsteht bei ihnen das *fatware*-Problem, wenn die Hersteller in Unkenntnis der konkreten Anforderung der individuellen Anwender versuchen, möglichst viele Funktionalitätsvarianten mitauszuliefern. Dementgegen bieten Laufzeitkomponenten zwar die technische Möglichkeit, die Funktionalität eines Anwendungssystems sitzungsweise neu festzulegen, die von Anwendern jedoch nicht realisiert werden kann, da dazu eine *Scripting* genannte Form der Programmierung erforderlich ist. Zudem besteht bei – im Sinne des in Teilen der Komponenten-Literatur postulierten Idealbildes – vollkommen unabhängig voneinander entwickelten Laufzeitkomponenten die Gefahr substantieller funktioneller Überlappung und semantischer Inkonsistenzen zwischen den einzelnen Komponenten.

Anwender können ein Kernsystems einer Anwendungsfamilie zur Laufzeit daher nur dann beliebig und sicher anpassen, wenn Rahmenwerk- und Komponenten-Ansätze kombiniert werden, um deren entgegengesetzte Schwächen aufzuheben: Das Kernsystem wird als ein Parametrisierungsrahmenwerk ausgeliefert, dessen Anpassungsstellen zur Laufzeit mit geeigneten Laufzeitkomponenten parametrisiert werden können. Technisch wird durch diese Art der Partitionierung der Anwendungsfamilie sowohl das *fatware*-Problem der reinen Rahmenwerk-Ansätze als auch das Scripting-Problem der reinen Komponenten-Ansätze gelöst. Fachlich müssen zwei Bedingungen erfüllt sein, damit die Anwender das Kernsystem sicher anpassen können: (1) Die Komponenten müssen eine für sie verständliche Funktionalität enthalten und (2) die binären Auslieferungseinheiten der Anwendungsfamilie müssen auf aus Anwendersicht verständliche Weise handhabbar sein. Die technische Bündelung zu DLLs, *shared libraries*, COM- und CORBA-Server allein ist nur eine notwendige Vorbedingung für die Anpassbarkeit.

Die erste Bedingung lässt sich durch einen Zuschnitt der Komponenten erreichen, der sich nicht an technischen Gesichtspunkten (vgl. [Bäu98]), sondern an fachlichen Einheiten orientiert, die sich während der Analyse ergeben haben. Die zweite Bedingung erfordert, dass das für die Anwendung gewählte Leitbild auf die Auslieferungseinheiten selbst ausgedehnt wird. Dafür präge ich den Begriff der *Systemmetapher*, die eine bildliche Vorstellung davon gibt, wie eine Auslieferungseinheit gehandhabt werden kann, um daraus eine lauffähiges System zusammenzustellen.

Im Rest des Kapitels habe ich daraufhin herausgearbeitet, dass die in der Literatur genannten Metaphern für mehrteilig ausgelieferte Softwaresysteme nur dann verwendbar sind, wenn erhebliche Anschauungsbrüchen in Kauf genommen werden:

- *Nicht-technische Metaphern*, die sich auf Gegenstände wie Legosteine, Dokumente oder Steckbretter und -steine beziehen, sind problematisch, weil sie nur unzureichend veranschaulichen können, dass die Einheiten eine bestimmte Funktionalität aufweisen.

- *Rein technische Metaphern* wie Client und Server sowie Plug-Ins sind wenig hilfreich, da sich die allermeisten Anwender nichts Konkretes unter ihnen vorstellen können und somit keine die Anschauung leitende Analogie gebildet werden kann.

- *Auf Alltagstechnik fußende Metaphern* wie Stecker und Steckdose bzw. Gerät einer Stereoanlage vermeiden die Nachteile der vorgenannten Arten von Metaphern. Die existierenden Metaphern sind jedoch für Anwendungsfamilien problematisch, da die Analogien nicht weit genug tragen, um zu veranschaulichen, dass der Kontrollfluss im Kernsystem angesiedelt ist und dass es verschiedene Arten von Anpassungsstellen gibt.

In Kapitel 5 habe ich die aus dem Bereich der Alltagstechnik stammende Systemmetapher von Einschub und Einschubrahmen eingeführt, die im Gegensatz zu den in der Literatur diskutierten Metaphern dazu geeignet ist, die Eigenschaften der Auslieferungseinheiten einer Anwendungsfamilie samt der Anpassungsstellen des Kernsystems bruchlos zu veranschaulichen: Geräte mit Einschubrahmen haben eine gewisse Grundfunktionalität, die an Einschubrahmen mit genau passenden Einschüben angepasst werden kann. Dabei wird über Steckverbindungen sowohl vorgegeben, welche Dienstleistungen vom Einschub erwartet werden als auch, welche Dienste des Kernsystems ein passender Einschub nutzen kann. Ein Gerät kann mehrere voneinander unabhängige Einschubrahmen mit unterschiedlich geformten Schnittstellen besitzen. Da Einschübe auch für das Funktionieren des Kernsystems notwendige (Einschub-)Motoren sein können, vermag die Metapher neben Einfach- und Mehrfach- auch Zwangsanpassungsstellen zu vergegenständlichen. Da die Verbindung im Einschubrahmen über Steckverbindungen hergestellt wird, veranschaulicht die Metapher auf zutreffende Weise, dass Konfigurationen sitzungsweise änderbar sind.

Anwendungsfamilien, deren Auslieferungseinheiten nach anwendungsorientierten Kriterien zugeschnitten sind und die sich gemäß der Systemmetapher Einschub und Einschubrahmen handhaben lassen, habe ich Einschubsysteme genannt. Nachdem das Konzept des Einschubsystems es Anwendern nunmehr ermöglicht, Anwendungssysteme selbständig sitzungsweise gemäß ihren aktuellen fachlichen Anforderungen sitzungsweise zusammenzustellen, habe ich mich im verbleibenden Teil des Kapitels der softwaretechnischen Umsetzung von Einschubsystemen zugewandt.

- *Softwaretechnisches Modell*. Im Gegensatz zu einer monolithisch entwickelten und ausgelieferten Anwendungsfamilie, muss im softwaretechnischen Modell eines Einschubsystems berücksichtigt werden, dass die als Einschübe aufgefassten Komponenten allein anhand der eingehenden und ausgehenden Schnittstelle des Einschubrahmens isoliert vom Kernsystem implementierbar und zur Laufzeit austauschbar sind. Unabhängigkeit vom Kernsystem bei der Implementation wird gewährleistet, indem interne Dienstleister und Parameterklassen durch Proxies im Einschubrahmen repräsentiert werden. Somit bietet sich Einschubentwicklern eine sta-

bile Schnittstelle, gegen die sie Einschübe entwickeln können, und die sie vor Veränderungen im Inneren des Kernsystems abschirmt. Unabhängigkeit der Einschübe untereinander zur Laufzeit entsteht dadurch, dass die im Einschubrahmen vorhandenen internen Dienstleister das einzige Mittel darstellen, anhand dessen die in sie hineinpassenden Einschübe die Funktionalität des Einschubsystems nutzen können. Da es den Einschüben aber verborgen bleibt, ob die Funktionalität der internen Dienstleister kernsystemintern oder aber von anderen Einschüben erbracht wird, können keine komplexen Abhängigkeitsgeflechte bei der Konfiguration entstehen.

- *Implementation.* Am Beispiel von Java habe ich abschließend gezeigt, wie ein Einschubsystem konkret realisierbar ist. Java eignet sich deshalb besonders gut, weil es anders als mit den bereits in Kapitel 4 diskutierten Technologien CORBA und COM mit Hilfe von standardmäßig vorhanden Mechanismen möglich ist, Black Boxes, statische Typisierung sowie eine minimale Zeitverzögerung beim Zugriff auf die Einschübe umzusetzen. Sowohl das Kernsystem als auch die Einschübe lassen sich in Java problemlos als JARs ausliefern. Da der Mechanismus, mit dem sich JARs dynamisch hinzubinden lassen, kein einschubgemäßen Prüfungen durchführt, ist ein Einschubmanager notwendig, der sicherstellt, dass Einschübe auch zu dem gewählten Einschubrahmen passen und alle softwaretechnischen Anforderungen an Einschübe erfüllen. Er übernimmt außerdem die Aufgabe, die eingefügten Einschübe im Einschubsystem bekannt zu machen. Da JARs als technische Binäreinheiten für Anwender nicht verständlich handhabbar sind, gibt es einen Konfigurationsdesktop, auf dem Einschübe und Einschubrahmen sondiert und Einschübe per *drag and drop* in die passenden Einschubrahmen verbracht werden können.

Mit dem Einschub-Ansatz steht damit ein konstruktiv validierter Ansatz zur Verfügung, mit dem Anwendungsfamilien bruchlos von der Analyse über den Entwurf, die Implementation, die Auslieferung bis hin zur verständlichen sitzungsweisen Konfiguration durch die Anwender realisierbar sind. Er weist in zwei unterschiedlichen Richtungen untersuchenswertes Potential für weitere Forschungsarbeiten auf:

- Für große Rahmenwerke wie beispielsweise *JWAM* bietet er die Möglichkeit, technische Entwurfsentscheidungen aufzuschieben, indem sie hinter der vom Einschubrahmen vorgegebenen Schnittstelle gekapselt werden. Ein Persistenzmechanismus kann beispielsweise zu Beginn der Entwicklung zunächst unmittelbar auf dem Dateisystem basieren und zu späteren Zeitpunkten bei höherem Datenaufkommen durch Einschübe ersetzt werden, die sich spezialisierter Datenbanken bedienen.

- Auf dem Gebiet der Komponenten-Ansätze bieten Einschubsysteme aufgrund der beherrschbar gestalteten Verflechtungen zwischen den einzelnen Komponenten und der Verbindung mit Rahmenwerk-Ansätzen eine andere Perspektive als die maximal flexiblen, rein auf den Kontext *verwendete Technik* beschränkte Komponententechnologien. Es wäre zu prüfen, inwiefern der aus Gründen der Anwendungsorientierung auch die Kontexte *Anwendungsbereich* und *Handhabung & Präsentation* einbeziehende Einschub-Ansatz auch als generelles Hilfsmittel bei der Diskussion und Konstruktion anderer Komponentensysteme erleichtern kann.

In Kapitel 6 habe ich schließlich einen in Kapitel 3 begonnen Diskussionsfaden wiederaufgenommen, der dort mit der Frage begonnen hatte, wie die Materialien Partitur und Notiz auf eine solche Weise modelliert werden können, dass sie zwar maximal die Funktionalität eines eventuell vorhandenen MDV-Systems nutzen können aber zugleich noch weiterhin nutzbar sind, falls ein solches System aufgrund knapper Ressourcen nicht zur Verfügung steht. Dieses spezielle Problem habe ich zunächst verallgemeinert und auf die generelle Modellierung von graphisch komplexen Materialien und komplexen Fachwerten ausgedehnt, wobei ich beide Begriffe anfangs definiert habe:

- *Graphisch komplexe Materialien* zeichnen sich dadurch aus, dass die Daten zu ihrer Präsentation nicht alleine anhand ihres fachlichen Zustands innerhalb der sie bearbeitenden Werkzeuge generierbar ist. Stattdessen ist es bei ihnen notwendig, auf von Menschenhand erzeugte Präsentationsdaten zurückzugreifen. Beispiele sind neben Partituren Stadtpläne und handgeschriebene Dokumente.

- *Komplexe Fachwerte* weisen – wie beispielsweise Bankleitzahlen – einen großen, endlichen Wertebereich auf oder bieten komplexe Funktionalität an, die – wie die Funktion `istFeiertag` eines Datums – nicht als herkömmliche leichtgewichtige Kapsel um eingebaute Datentypen realisierbar ist.

Die herkömmliche Lösung für beide besteht darin, die Funktionalität des komplexen Anteils nach oben in ein Werkzeug herauszuziehen, das sich dann wiederum an ein geeignetes Subsystem wendet. Auf diese Weise ist die aufgeworfene Konstruktionsfrage zwar lösbar, da die einfache Funktionalität ja im Material bzw. Fachwert verbleibt und auch bei fehlendem Subsystem zur Verfügung steht, aber die fachliche Kapselung der graphisch komplexen Materialien und komplexen Fachwerte wird durchbrochen, da die sie nutzenden Werkzeuge selbst aktiv durchreichend tätig werden müssen, um die gewünschte Funktionalität zu erhalten, obwohl dies eigentlich ausschließlich Sache der Materialien und Fachwerte sein sollte.

Die von mir entwickelte Lösung zieht die komplexe Funktionalität nicht nach oben in die Werkzeuge, sondern nach unten aus den Materialien und Fachwerten heraus. Beide erhalten dabei unmittelbaren Zugriff auf geeignete, Subsysteme kapselnde Dienstleister, so dass die von den einbettenden Werkzeugen gewünschte Funktionalität ohne deren Mitwirken erbracht werden kann. Die im WAM-Ansatz geforderte Passivität von Materialien und Fachwerten bleibt erhalten, da beide weiterhin nicht von sich aus, sondern nur auf Anfrage von Werkzeugen und Automaten hin aktiv werden.

Motiviert durch eine musikwissenschaftliche Fragestellung habe ich im Verbund meiner Diplomarbeit und dieser Dissertation versucht, die sich daraus für die Musikdatenverarbeitung und Softwaretechnik ergebenden Fragestellungen zu beschreiben und konstruktiv zu diskutieren. Zusätzlich zu den wissenschaftlichen Ergebnissen, hoffe ich, Musikwissenschaftlern mit *scr2hmd* und *JRing* die langersehnten geeigneten Werkzeuge an die Hand gegeben zu haben, um die oft seit über hundert Jahren offenen Formprobleme großer Werke wie bei *Der Ring des Nibelungen* lösen zu können.

Literaturverzeichnis

Ack94 Philipp Ackermann, „Direct Manipulation of Temporal Structures in a Multimedia Application Framework“ in: *Proceedings of the Second ACM International Conference on Multimedia* (San Francisco, 1994), S. 51-58

Alp74 Bo Alphonce, *The Invariance Matrix*, Dissertation (New Haven, CT: Yale University, 1974)

Bäu98 Dirk Bäumer, *Softwarearchitekturen für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme*, Dissertation (Hamburg, 1998)

Bau88 Hans-Joachim Bauer, *Richard-Wagner-Lexikon* (Bergisch Gladbach: Lübbe, 1988)

BC89 Kent Beck und Ward Cunnigham, „A Laboratory for Object-Oriented Thinking“ in: *Proceedings of OOPSLA ’89* (New Orleans, 1989) S. 1-6

BH93 Dorothea Blostein und Lippold Haken, „The Tilia Music Representation: Extensibility, Abstraction, and Notation Contexts for the LIME Music Editor“ in: *Computer Music Journal*, 17. Jahrgang, 3. Heft (Herbst 1993), S. 43-58

BJ94 Kent Beck und Ralph Johnson, „Patterns Generate Architectures.“ in: *ECOOP’94 Conference Proceedings* (Berlin, Heidelberg: Springer, 1994), S. 139-149

BK97 Albert Brennink und Mario Koppers, „A Salute to our Readers“ in: *Chroma Report*, 1. Jahrgang, 1. Heft (Juli 1997)

Bla99a Günther Blaschek, „Objektorientierte Programmierung“ in: Gustav Pomberger und Peter Rechenberger, *Informatik-Handbuch*, 2. Auflage (München: Carl Hanser, 1999), S. 529-552

Bla99b Günther Blaschek, „Mensch-Maschine-Kommunikation“ in: Gustav Pomberger und Peter Rechenberger, *Informatik-Handbuch*, 2. Auflage (München: Carl Hanser, 2000), S. 791-803

BM48 Harold Barlow und Sam Morgenstern, *A Dictionary of Musical Themes* (New York: Crown, 1948)

BM66 Harold Barlow und Sam Morgenstern, *A Dictionary of Opera and Song Themes* (New York: Crown, 1966)

BM77 Robert Boyer und J. Strother Moore, „A Fast String Searching Algorithm“ in: *Communications of the ACM*, 20. Jahrgang, 10. Heft (Oktober 1977), S. 762-772

Boh00 Holger Bohlmann, *Thin Clients in JWAM*, Diplomarbeit (Hamburg, 2000)

Boo94 Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2. Auflage, (Redwood City, CA: Benjamin/Cummings, 1994)

Bor99 Lothar Borrmann, „Betriebssysteme“ in: Gustav Pomberger und Peter Rechenberger, *Informatik-Handbuch*, 2. Auflage (München: Carl Hanser, 1999), S. 633-673

Box97 David Box, *Creating Components with DCOM and C++* (Reading, MA: Addison-Wesley, 1997)

Bro99 F.A. Brockhaus, *Brockhaus. Die Enzyklopädie in vierundzwanzig Bänden*, 20. Auflage (Leipzig, Mannheim: Brockhaus, 1999)

Byr94 Donald Byrd, „Music Software Notation and Intelligence“ in: *Computer Music Journal*, 18. Jahrgang, 1. Heft (Frühjahr 1994), S. 17-20

BZ90 Reinhard Budde und Heinz Züllighoven, „Softwarewerkzeuge in einer Programmierwerkstatt“ in: *Berichte der GMD*, Nr. 182 (München, Wien: Oldenbourg, 1990)

Car67 Norman Carrell, *Bach the Borrower* (London: George Allen & Unwin, 1967)

Cas94 Peter Castine, *Set Theory Objects: Abstractions for Computer-Aided Analysis and Composition of Serial and Atonal Music* (Frankfurt: Peter Lang, 1994)

CDFR95 Michael Conner, Scott Danforth, Ira Forman und Larry Raper, „Release-to-Release Binary Compatibility in SOM“ in: *ACM SIGPLAN Notices*, 30. Jahrgang, 10. Heft (Oktober 1995), S. 426-438

CG96 Gianpaolo Cugola und Carlo Ghezzi, „Program Families: Some Requirements Issues for the Process Languages?“ in: *Proceedings of the 10th International Software Process Workshop* (Dijon: IEEE, 1996), S. 51-53

CHW98 James Coplien, Daniel Hoffman und David Weiss, „Commonality and Variability in Software Engineering“ in: *IEEE Software*, 15. Jahrgang, 6. Heft (November/Dezember 1998), S. 37-45

CKV96 James Coplien, Norm Kerth und John Vlissides (Hrsg.) *Pattern Languages of Program Design 2* (Reading, MA: Addison-Wesley, 1996)

CJJO92 Magnus Christerson, Ivar Jacobson, Patrick Jonsson und Gunnar Overgaard, *Object-Oriented Software Engineering* (Reading, MA: Addison-Wesley / ACM Press, 1992)

Coo68 Deryk Cooke, *An Introduction to „Der Ring des Nibelungen“* (London: Decca, 1968)

Cop87 David Cope, „An Expert System for Computer-Assisted Music Composition“ in: *Computer Music Journal*, 11. Jahrgang, 4. Heft (Winter 1987), S. 30-46

Cop98 David Cope, „Signatures and Earmarks: Computer Recognition of Patterns in Music“ in: *Computing in Musicology* 11 (Cambridge, MA: MIT Press, 1998), S. 129-138

CIR98 Tim Crawford, Costas Iliopoulos und Rajeev Raman, „String Matching Techniques for Musical Similarity and Melodic Recognition“ in: *Computing in Musicology* 11 (Cambridge, MA: MIT Press, 1998), S. 73-100

CS95 James Coplien und Douglas Schmidt (Hrsg.) *Pattern Languages of Program Design* (Reading, MA: Addison-Wesley, 1995)

CS97 Edmund Correia, Jr., und Eleanor Selfridge-Field, „Glossary“ in: Eleanor Selfridge-Field (Hrsg.), *Beyond MIDI: The Handbook of Musical Codes* (Cambridge, MA: MIT Press, 1997), S. 581-610

CW98 David Cuka und David Weiss, „Engineering Domains: Executable Commands as an Example“ in: *Proceedings of the Fifth International Conference on Software Reuse* (Victoria, BC: IEEE, 1998), S. 26-34

CY79 Larry Constantine und Edward Yourdon, *Structured Design: Fundamentals of a Discipline of Computer Programs and Systems Design* (Englewood Cliffs, NJ: Prentice Hall, 1979)

Dal86a Carl Dalhaus, „Die Musik“ in: Ulrich Müller und Peter Wapnewski, *Richard-Wagner-Handbuch* (Stuttgart: Kröner, 1986), S. 197-221

Dal86b Carl Dalhaus, „Wagners Stellung in der Musikgeschichte“ in: Ulrich Müller und Peter Wapnewski, *Richard-Wagner-Handbuch* (Stuttgart: Kröner, 1986), S. 60-85

Der96 Kurt Derr, *Applying OMT – A Practical Step-by-Step Guide to Using the Object Modeling Technique* (New York: SIGS Books/Prentice Hall, 1996)

DG99 Jürgen Dorn und Georg Gottlob, „Künstliche Intelligenz“ in: Gustav Pomberger und Peter Rechenberger, *Informatik-Handbuch*, 2. Auflage (München: Carl Hanser, 1999), S. 975-998

DH80 Geoff Dowling und Patrick Hall, „Approximate string matching“ in: *ACM Computing Surveys*, 12. Jahrgang, 4. Heft (Oktober 1980), S. 381-402

Dit99 Klaus Dittrich, „Datenbanksysteme“ in: Gustav Pomberger und Peter Rechenberger, *Informatik-Handbuch*, 2. Auflage (München: Carl Hanser, 1999), S. 875-908

DNF96 Elisabetta Di Nitto und Alfonso Fuggetta, „Product Lines: What Are the Issues?“ in: *Proceedings of the 10th International Software Process Workshop* (Dijon: IEEE, 1996), S. 51-53

Dud90 Dudenredaktion, *Duden. Das Fremdwörterbuch*, 5. Auflage (Mannheim: Dudenverlag, 1990)

Dyd97 Stephen Dydo, „DARMS: The Note-Processor Dialect“ in: Eleanor Selfridge-Field (Hrsg.), *Beyond MIDI: The Handbook of Musical Codes* (Cambridge, MA: MIT Press, 1997), S. 175-191

Ebc86 Kemal Ebcioglu, „An Expert System for Harmonization of Chorales in the Style of J. S. Bach“, Dissertation (Buffalo, NY: State University of New York at Buffalo, 1986)

Ebc92 Kemal Ebcioglu, „An Expert System for Harmonization of Chorales in the Style of J. S. Bach“ in: Mira Balaban, Kemal Ebcioglu und Otto Laske, *Understanding Music with AI: Perspectives on Music Cognition* (Cambridge, MA: AAAI Press/MIT Press, 1992), S. 294-334

Egg67 Hans Heinrich Eggebrecht (Hrsg.), *Riemann Musiklexikon* (Mainz: B. Schott’s Söhne, 1967)

Fin97 Ludwig Finscher (Hrsg.), *Die Musik in Geschichte und Gegenwart*, 20 Bände in 2 Teilen, 2. Ausgabe (Kassel: Bärenreiter, 1997)

Fir93 Ramin Firoozyi, „Examining the Starview Application Framework“ in: *Dr. Dobbs Journal*, 17. Jahrgang, 12. Heft (Dezember 1993)

For77 Allen Forte, *The Structure of Atonal Music*, 2. Auflage (New Haven, CT: Yale University Press, 1977)

Fuc99 Norbert Fuchs, „Logische Programmierung“ in: Gustav Pomberger und Peter Rechenberger, *Informatik-Handbuch*, 2. Auflage (München: Carl Hanser, 1999), S. 567-566

FZ99 Christiane Floyd und Heinz Züllighoven, „Softwaretechnik“ in: Gustav Pomberger und Peter Rechenberger, *Informatik-Handbuch*, 2. Auflage (München: Carl Hanser, 1999), S. 763-790

GHJV98 Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*, 2. Auflage (Bonn: Addison-Wesley, 1998)

GHR+00 Guido Gryczan, Andreas Havenstein, Stefan Roock, Ingrid Wetzel und Heinz Züllighoven, „Frameworkbasierte Anwendungsentwicklung (Teil 5): Unterstützung von Kooperation mit persistenten fachlichen Behältern“ in: *Objektspektrum* (Januar/Februar 2000), S. 82-87

GJJ+00 Lu Jian, Liu Jianzhong, Zhang Guanqun, Cai Min, Tao Xianping, Ma Xiaoxing und Li Yingjun, „A Hierarchical Framework for Parallel Seismic Applications“ in: *Communications of the ACM*, 43. Jahrgang, 10. Heft (Oktober 2000), S. 55-59

GJS96 James Gosling, Bill Joy und Guy Steele, *The Java Language Specification* (Reading, MA: Addison-Wesley, 1996)

GJKW97 Neeraj Gupta, Lalita Jagadeesan, Eleftherios Koutsofios und David Weiss, „Auditdraw: Generating Audits the FAST Way“ in: *Proceedings of the Third IEEE International Symposium On Requirement Engineering* (Los Alamitos, CA: IEEE Computer Society Press, 1997), S. 188-197

GKZ94 Guido Gryczan, Klaus Kilberth und Heinz Züllighoven, *Objektorientierte Anwendungsentwicklung – Konzepte, Strategien, Erfahrungen*, 2. Auflage (Braunschweig, Wiesbaden: Vieweg, 1994)

GLL+99 Guido Gryczan, Carola Lilienthal, Martin Lippert, Stefan Roock, Henning Wolf und Heinz Züllighoven, „Frameworkbasierte Anwendungsentwicklung (Teil 1)“ in: *Objektspektrum* (Januar/Februar 1999), S. 90-99

GKR+01 Guido Gryczan, Andreas Kornstädt, Stefan Roock, Henning Wolf und Heinz Züllighoven, „Das JWAM-Framework und Komponenten. Eine konzeptionelle Bestandsaufnahme“ in: *Tagungsband des 3.Workshops zu komponentenorientierter betrieblicher Anwendungsentwicklung* (Frankfurt: April 2001), S. 15-21

GNT92 Simon Gibbs, Oscar Nierstrasz und Dennis Tsichritzis, „Component-Oriented Software Development“ in: *Communications of the ACM*, 35. Jahrgang, 9. Heft (September 1992), S. 160-165

Goo98 Peggi Goodwin, *The Microsoft Foundation Class Library and Active Template Library for Windows CE* (Redmond, WA: Microsoft Press, 1998) http://msdn.microsoft.com/library/backgrnd/html/msdn_cemfcatl.htm

Gra82 Hermann Grabner, *Allgemeine Musiklehre*, 14. Auflage (Kassel: Bärenreiter, 1982)

Gri98 Frank Griffel, *Componentware* (Heidelberg: dpunkt, 1998)

Gry96 Guido Gryczan, *Prozeßmuster zur Unterstützung kooperativer Tätigkeit* (Wiesbaden: Deutscher Universitätsverlag, 1996)

GS96 David Garlan und Mary Shaw, *Software Architecture. Perspectives on an Emerging Discipline* (Upper Saddle River, NJ: Prentice-Hall, 1996)

GZ99 Gerhard Goos und Wolf Zimmermann, „Programmiersprachen“ in: Gustav Pomberger und Peter Rechenberger, *Informatik-Handbuch*, 2. Auflage (München: Carl Hanser, 1999), S. 469-516

Ham97 Graham Hamilton, *Java Beans*, Version 1.01 (Palo Alto, CA: Sun Microsystems, Inc., 1997), http://java.sun.com/products/javabeans/docs/spec.html

Hee93 Beat Heeb, *Debora: A System for the Development of Field Programmable Hardware and its Application to a Reconfigurable Computer* (Zürich: Verlag der Fachvereine, 1993)

Hei97 Joseph Heim, „Integrating Distributed Simulation Objects“ in: *Proceedings of the 1997 Winter Simulation Conference* (Atlanta, GA 1994), S. 532-538

Hen01 Norman Hendrich: *HADES Simulation Framework Homepage* (Hamburg: 2001) http://tech-www.informatik.uni-hamburg.de/applets/hades/

Hew97 Walter Hewlett, „*MuseData*: Multipurpose Representation“ in: Eleanor Selfridge-Field (Hrsg.), *Beyond MIDI: The Handbook of Musical Codes* (Cambridge, MA: MIT Press, 1997), S. 402-447

HO92 David Huron und Keith Orpen, „Measurement of Similarity in Music: A Quantitative Approach for Non-parametric Representations“ in: *Computers in Music Research*, 4. Jahrgang (1992), S. 1-44

Hur94 David Huron, *The Humdrum Toolkit Reference Manual* (Menlo Park, CA: CCARH, 1994)

Hur96 David Huron, „The Melodic Arch in Western Folksongs“ in: *Computing in Musicology* 10 (Stanford: CCARH, 1996), S. 3-23

Hur97 David Huron, „*Humdrum* and *Kern*: Selective Feature Encoding“ in: Eleanor Selfridge-Field (Hrsg.), *Beyond MIDI: The Handbook of Musical Codes* (Cambridge, MA: MIT Press, 1997), S. 376-401

ITU97 Telecommunication Standardization Sector of ITU, *Recommendation X.509 (08/97) – Information technology – Open Systems Interconnection – The Directory: Authentication framework*, (Genf: International Telecommunication Union, 1997)

Jac92 Ivar Jacobson, *Object-Oriented Software Engineering – A Use Case Driven Approach* (Reading, MA: Addison-Wesley, 1992)

JF88 Ralph Johnson und Brian Foote, „Designing Reusable Classes“ in: *The Journal of Object-Oriented Programming*, 1. Jahrgang, 2. Heft (Juni/Juli 1988), S. 22-35

JM94 Hans-Dieter Junge und Albrecht Möschwitzer, *Lexikon Elektronik* (Weinheim: VCH, 1994)

Joh92 Ralph Johnson, „Documenting Frameworks Using Patterns“ in: *ACM SIGPLAN Notices*, 27. Jahrgang, 10. Heft (Oktober 1992), S. 63-70

JR91 Ralph Johnson und Vincent Russo, *Reusing Object-Oriented Designs* (Urbana-Champaign: Department of Computer Science, University of Illinois, 1991)

Kor96a Andreas Kornstädt, *Formbildung durch Muster: Leitmotive und Objekte. I: Ein Werkzeug zur leitmotivischen Analyse von ‚Der Ring des Nibelungen‘*, Diplomarbeit (Hamburg, 1996)

Kor96b Andreas Kornstädt, „SCORE-to-Humdrum: A Graphical Environment for Musicological Analysis“ in: *Computing in Musicology 10* (Stanford: CCARH, 1996), S. 105-122

Kor97a Andreas Kornstädt, *An Interactive Tool for Large-Scale Leitmotivic Analysis*, Vortrag auf dem 16. Kongress der International Musicological Society (London: Imperial College, 18. 8. 1997)

Kor97b Andreas Kornstädt, *Software Development for Musicologists*, Vortrag auf der Tagung der IMS Study Group on Musical Data and Computer Applications (London: Kings College, 21. 8. 1997)

Kor98a Andreas Kornstädt, „Themefinder: A Web-based Melodic Search Tool“ in: *Computing in Musicology* 11 (Cambridge, MA: MIT Press, 1998), S. 231-236

Kor98b Andreas Kornstädt, „CCARH’s Musical Databases on the Web: A Gold Mine for Musicologists“ in: *Proceedings of the XII Colloquium on Musical Informatics* (Gorizia: Assoziazione di Informatica Musicale Italiana, 1998), S. 280-281

Kor99 Andreas Kornstädt, *Document Analysis*, Vortrag auf dem Workshop Creating A Digital Corpus (Utrecht: Universität Utrecht, 10. 6. 1999)

Kor00 Andreas Kornstädt, *A Flexible Slide-In System for Computer-Assisted Musicological Analysis*, Vortrag am Center for Computer Research in Music and Acoustics (Stanford: CCRMA, 13. 12. 2000)

Kor01a Andreas Kornstädt, „A Flexible Slide-In System for Computer-Assisted Musicological Analysis“ in: *Proceedings of the 5th World Multiconference on Systemics, Cybernetics and Informatics*, 10. Band (Orlando: International Institute of Informatics and Systemics, 2001), S. 51-56

Kor01b Andreas Kornstädt, „The JRing System for Computer-Assisted Musicological Analysis“ in: *Proceedings of the 2nd Annual International Symposium on Music Information Retrieval*, (Bloomington, IN, 2001), in Druck

Kra00 Anita Krabbel, *Entwurf, Auswahl und Anpassung aufgabenbezogener Domänensoftware*, Dissertation, (Hamburg, 2000)

Leh80 Meir Lehmann, „Programs, Life Cycles, and Laws of Software Evolution“ in: *Proceedings of the IEEE*, 68. Band, 9. Heft (1980), S. 1060-1076

Lip99 Martin Lippert, *Die Desktop-Metapher in Systemen nach dem Werkzeug- und Material-Ansatz*, Diplomarbeit (Hamburg, 1999)

LRWZ01 Martin Lippert, Stefan Roock, Henning Wolf und Heinz Züllighoven, „JWAM and XP – Using XP for Framwework Development“ in: Giancarlo Succi und Michele Marchesi (Hrsg.), *Extreme Programming Examined* (Reading, MA: Addison-Wesley, 2001), S. 103-117

Lor24 Alfred Lorenz, *Das Geheimnis der Form bei Richard Wagner, 1. Band: Der musikalische Aufbau des Bühnenfestspiels „Der Ring des Nibelungen“* (Berlin: Max Hesse, 1924)

LY99 Tim Lindholm und Frank Yellin, *The Java™ Virtual Machine Specification*, 2. Auflage (Palo Alto, CA: Sun Microsystems, Inc., 1999), http://java.sun.com/products/jdk/1.2/docs/books/vmspec/html/VMSpecTOC.doc.html

Maa94 Susanne Maaß, „Maschine, Partner, Medium, Welt … Eine Leitbildgeschichte der Software-Ergonomie“ in: Hans-Dieter Hellige (Hrsg.), *Leitbilder der Informatik- und Computer-Entwicklung* (Bremen: artec, 1994)

Mat94 Thomas Mathiesen, „Transmitting Text and Graphics in Online Databases: The TML Model“ in: *Computing in Musicology* 9 (Menlo Park: CCARH, 1994), S. 33-48

McI68 Doug McIlroy, „Mass Produced Software Components“ in: Peter Naur und Brian Randell (Hrsg.), *Proceedings, NATO Conference on Software Engineering* (Brussels: NATO Science Comittee, 1968), S. 88-98

MD95 Microsoft Corporation and Digital Equipment Corporation, *The Component Object Model Specification*, Entwurfsversion 0.9 (Redmond, WA: Microsoft Press, 1995) http://msdn.microsoft.com/library/specs/S1D13A.HTM

Mey90 Bertrand Meyer, *Objektorienterte Softwareentwicklung* (München: Hanser, 1990)

Mey97 Bertrand Meyer, *Object-Oriented Software Construction*, 2. Auflage (Upper Saddle River, NJ: Prentice-Hall, 1997)

MNZ96 Guerino Mazzola, Thomas Noll und Oliver Zahorka, „The RUBATO Platform“ in: *Computing in Musicology* 10 (Stanford: CCARH, 1996), S. 143-149

MO92 Susanne Maaß und Horst Oberquelle, „Perspectives and Metaphors for Human-Computer-Interaction“ in: Christiane Floyd, Heinz Züllighoven, Reinhard Budde und Reinhard Keil-Slawik (Hrsg.), *Software Development and Reality Construction* (Berlin, Heidelberg: Springer, 1992), S. 233-251

Mös99 Hanspeter Mössenböck, „Systemsoftware“ in: Gustav Pomberger und Peter Rechenberger, *Informatik-Handbuch*, 2. Auflage (München: Carl Hanser, 1999), S. 723-736

Mor92 Philip Morehead, *Bloomsbury Dictionary of Music* (London: Bloomsbury, 1992)

Müh99 Max Mühlhäuser, „Verteilte Systeme“ in: Gustav Pomberger und Peter Rechenberger, *Informatik-Handbuch*, 2. Auflage (München: Carl Hanser, 1999), S. 675-708

MZ94 Guerino Mazzola und Oliver Zahorka, „The RUBATO Workstation for Musical Analysis and Performance“ in: *Proceedings of the 3rd ICMPC* (Liège: 1994), S. 351-352

Nel95 Chris Nelson, *OpenDoc and Its Architecture* (Armonk, NY: International Business Machines Corporation, 1995)
http://www.ibm.com/software/ad/opendoc/library/aixpert.aug95.opendoc.html

Net98a Netscape Communications, *Plug-in Guide* (Mountain View, CA: Netscape Communications Corporation, 1998)

Net98b Nigel Nettheim, „Melodic Pattern-Detection Using *MuSearch* in Schubert's *Die schöne Müllerin*“ in: *Computing in Musicology* 11 (Cambridge, MA: MIT Press, 1998), S. 159-168

ND95 Oscar Nierstrasz und Laurent Dami, „Component-Oriented Software Technology“ in: Oscar Nierstrasz und Dennis Tsichritzis, *Object-Oriented Software Composition* (Upper Saddle River, NJ: Prentice-Hall, 1995), S. 3-28

Nie98 Lara Niemeyer, *Konzeption und Realisierung eines Verlaufsdokumentationssystems in der Psychiatrie nach der Werkzeug-Material-Metapher*, Diplomarbeit (Hamburg, 1998)

NT95 Oscar Nierstrasz und Dennis Tsichritzis, *Object-Oriented Software Composition* (Upper Saddle River, NJ: Prentice-Hall, 1995)

Obe97 Oberon microsystems, *BlackBox Developer and BlackBox Component Framework* (Zürich: Oberon microsystems, 1997), http://www.oberon.ch/prod/blackbox/index.html

OMa92 Donncha Ó Maidín, „Representation of Music Scores for Analysis“ in: Alan Marsden und Anthony Pople (Hrsg.), *Computer Representations and Models in Music* (London: Academic Press, 1992), S. 67-93

OMG97a Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Version 2.0, (Needham, MA: Object Management Group, 1997)

OMG97b Object Management Group, *The Object Management Architecture Guide* (Needham, MA: Object Management Group, 1997)

OMG97c Object Management Group, *CORBAservices: Common Object Services Specification* (Needham, MA: Object Management Group, 1997)

OS96 Leon Osterweil und Stanley Sutton, „Product Families and Product Lines“ in: *Proceedings of the 10th International Software Process Workshop* (Dijon: IEEE, 1996), S. 109-111

OS00 Michael Otto und Norbert Schuler, *Fachliche Services: Geschäftslogik als Dienstleistung für verschiedene Benutzungsschnittstellen-Typen*, Diplomarbeit (Hamburg, 2000)

OSU00 Open Source UNIX Research Initiative, *Enterprise Adoption of Linux* (Campbell, CA: Survey.com, 2000) http://www.survey.com/bidw/description_linux.html

Par72 David Parnas, „On the Criteria to be Used in Decomposing Systems into Modules“ in: *Communications of the ACM*, 15. Jahrgang, 12. Heft (Dezember 1972), S. 1053-1058

Par75 Denys Parson, *The Dictionary of Tunes and Musical Themes*. (Cambridge: Spencer Brown, 1975)

Par76 David Parnas, „On the Design and Development of Program Families“ in: *IEEE Transactions on Software Engineering*, 2. Jahrgang, 1. Heft (März 1976), S. 1-9

Pom99 Gustav Pomberger, „Prozedurorientierte Programmierung“ in: Gustav Pomberger und Peter Rechenberger, *Informatik-Handbuch*, 2. Auflage (München: Carl Hanser, 1999), S. 517-528

Pre97 Wolfgang Pree, *Komponentenbasierte Softwareentwicklung mit Frameworks* (Heidelberg: dpunkt, 1997)

Ree97 Thomas Reed, *Directory of Music Notation Proposals* (Kirksville, MO: Notation Research Press, 1997)

Rin81 Curt Rint (Hrsg.), *Handbuch für Hochfrequenz- und Elektrotechniker* (München: Hüthig und Pflaum, 1981)

Rog97 Dale Rogerson, *Inside COM* (Redmond, WA: Microsoft Press 1997)

Sam97 Johannes Sametinger, *Software Engineering with Reusable Components* (Berlin: Springer, 1997)

San01 Ulrich Sandl, *Digitale Signatur* (Berlin: Bundesministerium des Inneren und Bundesministerium für Wirtschaft und Technologie, 2001), http://www.sicherheit-im-internet.de/themes/themes.phtml?ttid=38

Sch35 Robert Schumann, „Sinfonie von H. Berlioz“ in: *Neue Zeitschrift für Musik*, 3. Band (1835/II)

Sch89 Bill Schottstaedt, „Automatic Counterpoint“ in: Max V. Mathews und John R. Pierce, *Current Directions in Computer Music Research* 11 (Cambridge, MA: MIT Press, 1989), S. 199-214

Sel97a Eleanor Selfridge-Field, „Introduction“ in: Eleanor Selfridge-Field (Hrsg.), *Beyond MIDI: The Handbook of Musical Codes* (Cambridge, MA: MIT Press, 1997), S. 3-38

Sel97b Eleanor Selfridge-Field, „*DARMS*, Its Dialects, and Its Uses“ in: Eleanor Selfridge-Field (Hrsg.), *Beyond MIDI: The Handbook of Musical Codes* (Cambridge, MA: MIT Press, 1997), S. 163-174

Sel97c Eleanor Selfridge-Field, „Appendix 3: Code-Translation Programs“ in: Eleanor Selfridge-Field (Hrsg.), *Beyond MIDI: The Handbook of Musical Codes* (Cambridge, MA: MIT Press, 1997), S. 543-550

Sel98 Eleanor Selfridge-Field, „Conceptual and Representational Issues in Melodic Comparison“ in: *Computing in Musicology* 11 (Cambridge, MA: MIT Press, 1998), S. 3-64

Shn83 Ben Shneiderman, „Direct Manipulation: A Step Beyond Programming Languages“ in: *IEEE Computer*, 16. Jahrgang, 8. Heft (August 1983), S. 57-69

Smi97 Leland Smith, „SCORE“ in: Eleanor Selfridge-Field (Hrsg.), *Beyond MIDI: The Handbook of Musical Codes* (Cambridge, MA: MIT Press, 1997), S. 252-280

Sol84 Larry Solomon, *Music Analysis System* (Tuscon, AZ: Soft stuff, 1984)

ST96 Murray Stainton und William Forde Thomspon, „Using *Humdrum* to Analyze Melodic Structure“ in: *Computing in Musicology* 10 (Stanford: CCARH, 1996), S. 24-33

Stu00 Thorsten Sturm, *Entwicklung von Sprachkonzepten zur Vereinheitlichung von Nebenläufigkeit und Verteilung in Dejay*, Diplomarbeit (Hamburg, 2000)

Sun98a Sun Microsystems, Inc., *jar – The Java Archive Tool* (Palo Alto, CA: Sun Microsystems, Inc., 1998), http://java.sun.com/products/jdk/1.2/docs/tooldocs/solaris/jar.html

Sun98b Sun Microsystems, Inc., *jarsigner – JAR Signing and Verification Tool* (Palo Alto, CA: Sun Microsystems, Inc., 1998), http://java.sun.com/products/jdk/1.2/docs/tooldocs/solaris/jarsigner.html

Sun98c Sun Microsystems, Inc., *Permissions in the Java™ 2 SDK* (Palo Alto, CA: Sun Microsystems, Inc., 1998), http://java.sun.com/products/jdk/1.2/docs/guide/security/permissions.html

Sun98d Sun Microsystems, Inc., *Policy Tool – Policy File Creation and Management Tool* (Palo Alto, CA: Sun Microsystems, Inc., 1998), http://java.sun.com/products/jdk/1.2/docs/tooldocs/solaris/policytool.html

Sun98e Sun Microsystems, Inc., *Security Managers and JDK 1.2* (Palo Alto, CA: Sun Microsystems, Inc., 1998), http://java.sun.com/products/jdk/1.2/docs/guide/security/smPortGuide.html

Sun98f Sun Microsystems, Inc., *X.509 Certificates and Certificate Revocation Lists (CRLs)* (Palo Alto, CA: Sun Microsystems, Inc., 1998), http://java.sun.com/products/jdk/1.2/docs/guide/security/cert3.html

Sun01 Sun Microsystems, Inc., *Java™ Platform Ports* (Palo Alto, CA: Sun Microsystems, Inc., 2001), http://java.sun.com/cgi-bin/java-ports.cgi

Szy98 Clemens Szyperski, *Component Software* (Harlow: Addison Wesley Longman, 1998)

Tay96 Michael Taylor, *Humdrum Graphical User Interface*, MA Thesis (Belfast: Queen’s University, 1996)

Tur99 Kevin Turner, *Writing a GIMP plug-in* (1999) http://www.poboxes.com/kevint/gimp/doc/plugin-doc.html

Wag Richard Wagner, *Das Rheingold* (Mainz: B. Schott’s Söhne, o.J.)

Wag83 Richard Wagner, „Oper und Drama“ in: Dieter Borchmeyer, *Richard Wagner. Dichtungen und Schriften*, Band 7 (Frankfurt: Insel, 1983)

WAM98 Heinz Züllighoven, *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz* (Heidelberg: dpunkt, 1998)

Wec97 Wolfgang Weck „Independently extensible component frameworks" in: Max Mühlhäuser (Hrsg.), *Special Issues in Object-Oriented Programming – ECOOP96 Workshop Reader* (Heidelberg: dpunkt, 1997)

Wie96 Frans Wiering, „Italian Music Treatise on CD-ROM" in: *Computing in Musicology* 10 (Stanford: CCARH, 1996), S. 183-188

Win Lothar Windsperger, *Das Buch der Motive und Themen aus sämtlichen Opern und Musikdramen Richard Wagners* (Mainz: B. Schott's Söhne, o.J.)

Wol96 Hans von Wolzogen, *Führer durch die Musik zu Richard Wagners Festspiel ‚Der Ring des Nibelungen'* (Leipzig: Reinboth, 1896)

WPS01 Apcon WPS, *The JWAM Homepage* (Hamburg, 2001) http://www.jwam.de

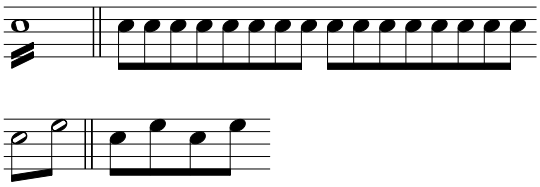
Wul95 Martina Wulf, *Konzeption und Realisierung einer Umgebung zur Koordination rechnergestützter Tätigkeiten in kooperativen Arbeitsprozessen*, Diplomarbeit (Hamburg, 1995)

WWW90 Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener, *Designing Object-Oriented Software* (Upper Saddle River, NJ: Prentice-Hall, 1990)

Anhang A Glossar musikwissenschaftlicher Begriffe

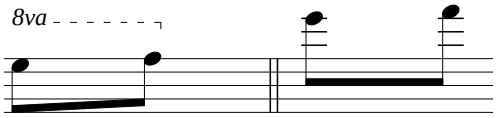
Abbreviatur. Abkürzung in der Notenschrift aus Gründen der Raumersparnis oder besseren Übersicht.

1. Tonwiederholungen. Über längere Zeit zu wiederholende Noten oder zweinotige Folgen. Notiert mit unverbundenen (eine Note) bzw. verbundenen (zweinotige Folge) →Querbalken, deren Anzahl den Wert der einzelnen Noten angibt.



Bsp.: eine Note (oben), zweinotige Folge (unten)

2. Oktavzeichen zur Vermeidung vieler Hilfslinien. Bis zum Ende der Linie werden alle Noten um sieben →diatonische Tonstufen höher (8va) oder niedriger (8va bassa) gespielt.



Bsp.: 8va

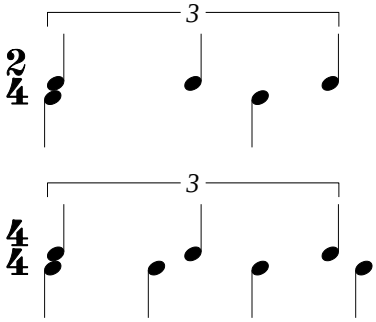
3. →Verzierungen

Akkolade. Eigentlich Klammer, die zusammengehörige →Systeme in einer →Partitur verbindet. Auch Bezeichnung für diese Systeme als Ganzes.

Akkord. Gleichzeitiger geordneter Zusammenklang von mindestens drei Tönen verschiedener Tonhöhe. Zum Bereich der →Harmonik gehörend.

Akt, bei Wagner: **Aufzug.** Hauptabschnitt einer Oper. Weitere Unterteilung in →Szenen möglich.

Anomale Teilung. Im Bereich der anomalen Teilung werden Töne einer bestimmten Dauer gleichmäßig gedehnt oder verkürzt. Die Anzahl der Noten bestimmt den Namen der anomalen Teilung (Duole, Triole, Quintole, etc.) und wird an einer Klammer oder am Querbalken notiert.



Bsp.: verkürzende Triole (3x1/6 statt 2x1/4; oben), verlängernde Triole (3x1/3 statt 4x1/4; unten)

Anschlag. Art und Weise der Tongebung bei Tasteninstrumenten, bestimmt durch die Bewegung der Finger.

Artikulation. Verbindungs- oder Trennungsarten von Tönen, die aufgrund bestimmter Spielweisen miteinander verschmolzen (legato) oder aber durch winzige Pausen voneinander getrennt werden (staccato, portato, marcato). Durch →Bindebögen oder Artikulationszeichen direkt über oder unter den Noten notiert.

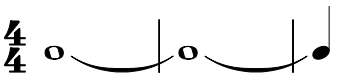
Bsp.: legato, staccato, portato, marcato

Auszug. Die volle Werklänge abdeckende Teilpartitur, in der nur eine oder einige wenige Orchesterstimmen notiert sind. Dient als Notenblattsammlung für die jeweiligen Musiker.

Bindebogen. Zwei oder mehr Noten umfassender Bogenstrich in der Partitur.

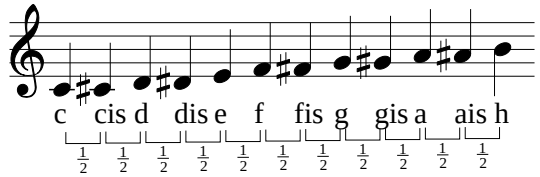
1. zwischen Noten verschiedener Höhe (Legatobogen): Töne sollen gebunden, d.h. ohne abzusetzen gespielt werden (→Artikulation).

2. zwischen Noten gleicher Höhe (Haltebogen): Zusammenfassung von Einzelnoten zu einem Ton, dessen Länge der Summe der Einzeltonlängen entspricht.



Bsp.: Haltebogen für einen Ton mit 9/4-Dauer

Chromatische Tonleiter. Tonleiter vom Umfang einer →diatonischen Tonleiter, die aber im Gegensatz zu dieser ausschließlich in gleich großen Halbtonschritte voranschreitet (+½ +½ +½ +½ +½ +½ +½ +½ +½ +½ +½ (+½).



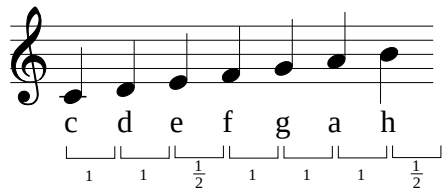
Bsp.: Chromatische Tonleiter ab C

Diatonische Tonleiter. Tonleiter, die in Ganz- und Halbtonschritten von c bis h schreitet: c (Grundton) d e f g a h (c, Grundton der nächsthöheren Oktave).

Dur-Tonleiter: Die Abstände zwischen den Tönen sind vom Grundton ausgehend +1 +1 +½ +1 +1 +1 +½ (1=Ganzton, ½=→Halbton).

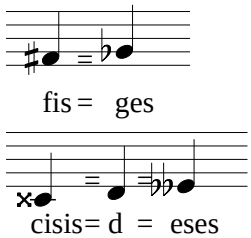
Moll-Tonleiter: Die Abstände zwischen den Tönen sind vom Grundton ausgehend +1 +½ +1 +1 +1 +1 +½.

Die Tonleitern werden nach ihren Grundtönen benannt (C-Dur, a-Moll).



Bsp.: C-Dur-Tonleiter

Enharmonische Töne. Verschieden notierte Töne, die aber gleich klingen.



Enharmonische Verwechslung. Vertauschung →enharmonischer Töne in der Notenschrift zur logischen Weiterführung von Harmonie oder Melodie.

Entwicklungsform. Musikalische Form, bei der sich das musikalische Material im Verlauf des Werks weiterentwickelt und nicht lediglich wie bei →Reihungsformen im Wesentlichen unverändert wiederholt und aneinandergereiht wird.

Epoche. Zeitraum innerhalb der Musikgeschichte, in dem bestimmte Kompositionsstile vorherrschend waren – beispielsweise Barock, Klassik und Romantik. Da Stile fließend ineinander übergehen und sich regional nicht gleichmäßig verbreiten, sind absolute Abgrenzungen nicht möglich.

Fuge. →Reihungsform mit dem Grundschema ||: Durchführung Zwischenspiel :|| Durchführung. Dabei gilt für die Durchführung: 1.Thema in 1. Stimme, dann 1. Thema in 2. Stimme mit →kontrapunktischem 2. Thema in 1. Stimme. Dieses Themenpaar wandert durch alle Stimmen (2+3, 4+5, ...), wobei sie sich innerhalb einer Durchführung nicht verändern, während von Durchführung zu Durchführung jeweils eine melodische oder rhythmische Veränderung stattfindet. In der letzten Durchführung sind die Themen dabei wieder wie in der ersten. Die Zwischenspiele: können unabhängig oder abhängig von den Durchführung gestaltet sein.

Gattung. Zusammenfassender Begriff für Musikstücke, die bestimmte Merkmale miteinander gemein haben. Aufgrund der vielfältigen Ausprägungen von Musik kann sich eine Gattung unter anderem sowohl auf den Gegenstand (sakral, säkular), die Besetzung (instrumental, vokal) als auch auf die Form beziehen (→Lied, Kantate, →Symphonie, →Oper, Oratorium).

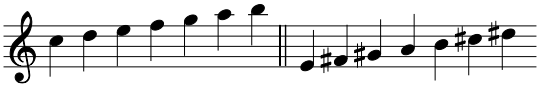
Halbton. Die kleinste Tonstufe des →diatonischen Tonsystems. Bei →diatonischen Dur-Tonleitern sind nur die Abstände zwischen dem 3. und 4. sowie dem 7. und 8. Ton Halbtonschritte. Bei →chromatischen Tonleitern sind alle Abstände Halbtonschritte. Halbtöne, die nicht einer diatonischen Tonleiter angehören, werden durch →Versetzungszeichen kenntlich gemacht.

Ein Ganztonschritt entspricht zwei Halbtonschritten.

Harmonik. Die Lehre von der Bedeutung des gleichzeitigen Zusammenklangs von Tönen verschiedener Höhe, die tonartlich zueinander in Beziehung stehen, und deren Verbindung. Neben →Melodik und →Rhythmik ein Hauptelement der Musik.

Instrument. Gerät zum Hervorbringen musikalisch brauchbarer Töne. In der musikalischen Praxis untergliedert in Streichinstrumente, Blasinstrumente und Schlaginstrumente.

Transponierende (Blas-)Instrumente werden in der Partitur anders notiert, als sie klingen, um dem Musiker das Spielen zu erleichtern oder um Hilfslinien zu sparen. Transponierende Instrumente der ersten Gruppe sind Englischhorn, Klarinette, Trompete und Waldhorn. Zur zweiten Gruppe gehören Piccoloflöte (klingt 1 Oktave höher als notiert), Kontrafagott und Kontrabaß (je 1 Oktave tiefer).



notiert Klang

Bsp.: C-Dur-Tonleiter für Horn in E

Klavierauszug. Für das Klavier notierte, unter größtmöglicher Beibehaltung der →Melodik, →Harmonik und →Rhythmik reduzierte →Partitur.

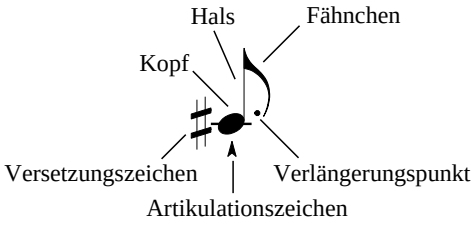
Kontrapunkt. Im engeren Sinne eine im 14. Jahrhundert entwickelte Kompositionstechnik, bei der zu einem Thema neue melodisch selbständige Stimmen erfunden und nach strengen Regeln fortgeführt werden. Im weiteren Sinne jede Art von →polyphoner Komposition.

Leitmotivtechnik. Das Verfahren, ein musikalisches Drama (→Oper) durch ein dichtes Gewebe leitmotivischer Beziehungen von innen heraus zusammenzuhalten.

Melodik. Die Lehre von der Beschaffenheit von Tonfolgen, die als selbständige charakteristische Gebilde auftreten. Neben →Harmonik und →Rhythmik ein Hauptelement der Musik.

Monophonie. Einstimmige (→Stimme) Komposition. Gegensatz: →Polyphonie.

Note. Zeichen zur schriftlichen Festlegung von →Tönen. Gibt die →Tondauer durch ihre Gestalt und die →Tonhöhe durch die Stellung des Notenkopfs im →System in Verbindung mit →Notenschlüssel und →Versetzungszeichen an. Besteht aus Notenkopf, Notenhals, und Fähnchen. Um eine Note herum können →Artikulationszeichen, →Versetzungszeichen und →Verlängerungspunkte angeordnet sein, die den Notenwert beeinflussen.



Notendauer. Die Notengrunddauern sind (wie bei →Pausen) die Potenzen von ½ von 0 aufwärts, also 1, 1/2, 1/4, 1/8, 1/16, 1/32, 1/64, etc. Die Notengrunddauer wird durch eine Kombination von Notenkopf, Notenhals und Fähnchen ausgedrückt:

Ganze Noten bestehen nur aus einen unausgefüllten Kopf,

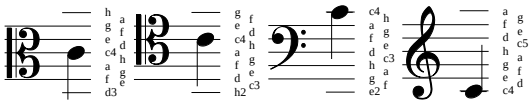
Halbe Noten besitzen zusätzlich einen Hals,

Viertel Noten sind wie halbe Noten, jedoch mit ausgefülltem Kopf,

Achtel, Sechzehntel-, Zweiundreißigstelnoten, etc. besitzen zusätzlich zu Viertelnoten ein Fähnchen je weitere Halbierung der Notendauer (Achtel: 1, Sechszehntel: 2, Zweiunddreißigstel: 3, etc.). Von den Grunddauern abweichende Tondauern können durch →Verlängerungspunkte oder →anomale Teilung erreicht werden.

Bsp.: Notengrunddauern von 1/1 bis 1/64

Notenschlüssel. Stilisierte Buchstaben, die je nach ihrer vertikalen Anordnung die Zuordnung von →diatonischen Tonwerten zu den Linien eines →Systems festlegen. Werden verwendet, um bei der Notation mit möglichst wenig Hilfslinien auszukommen.



C-Schlüssel als Altschl. *C-Schlüssel als Tenorschl.* *F-Schlüssel als Bassschl.* *G-Schlüssel als Violinschl.*

Oper. Musikalisches Bühnenwerk mit Darstellung einer Handlung durch Gesang und Instrumentalmusik. Je nach Operntyp steht das zugrundeliegende Drama, die Musik oder die Verschmelzung der beiden (Musikdrama) im Vordergrund. Besteht meistens aus →Ouvertüre und mindestens einem ggf. in →Szenen unterteilten →Akten.

Ouvertüre. Instrumentale Einleitung einer →Oper.

Partitur. Notenschriftliche Aufzeichnung ein- oder mehrstimmiger Musik in der die einzelnen →Stimmen innerhalb von →Akkoladen so übereinander angeordnet sind, daß der Verlauf der Einzelstimmen, ihre Koordination und die Zusammenhänge abgelesen werden können.

Pause. Zeichen zur schriftlichen Festlegung der Dauer des Schweigens einer Orchesterstimme. Die Pausengrunddauern sind (wie bei →Noten) die Potenzen von ½ von 0 aufwärts, also 1, 1/2, 1/4, 1/8, 1/16, 1/32, 1/64, etc. und werden durch die Form des Pausenzeichens ausgedrückt.

Von den Grunddauern abweichende Pausendauern können durch →Verlängerungspunkte ausgedrückt werden.

Bsp.: Pausengrunddauern von 1/1 bis 1/64

Phrase. Melodische Sinneinheit. Kann vom Komponisten unvollkommen durch einen →Bindebogen bezeichnet werden. Wegen Überlagerungen oft nicht eindeutig identifizierbar.

Pitch Class. Die Menge aller Töne, die oktavenunabhängig die gleiche Position in der 12stufigen →chromatischen Tonleiter innehaben. Für jede der 12 Pitch Classes steht eine Zahl von 0 bis 11, wobei für 10 und 11 auch A und B geschrieben wird, um ohne zweiteilige Bezeichner auszukommen.

Polyphonie. Mehrstimmige (→Stimme) Komposition, bei der sich die Stimmen →melodisch und →rhythmisch relativ unabhängig voneinander entwickeln. Gegensatz: →Monophonie. Eine spezielle Form des mehrstimmigen Satzes, bei der die Stimmen nur melodisch voneinander unabhängig sind, ist die Homophonie.

Probennummer. Zeitliche Markierung in einer →Partitur, an der das Einsetzen zu Probezwecken sinnvoll erscheint.

Querbalken. Ein dicker Strich, der bei einer Gruppe von Achtel- oder kürzeren →Noten die Notenhälse miteinander verbindet. Die Anzahl der Querbalken entspricht der Anzahl der (nun nicht mehr gezeichneten) Fähnchen. Dient der Übersichtlichkeit, der Phrasierung (→Phrase) bei Singstimmen oder (bei ganzen, halben und Viertelnoten) als →Abbreviatur.

Reihungsform. Musikalische Form, bei der sich das musikalische Material im Verlauf des Werks nicht weiterentwickelt (→Entwicklungsformen), sondern wie in einem Lied in gleicher Form – ggf. mit einem anderen Text – unverändert wiederholt und aneinandergereiht wird.

In komplexen Reihungsformen wie der →Fuge entwickelt sich das musikalische Material ausschließlich sprungartig von Abschnitt zu Abschnitt weiter.

Rhythmik. Lehre von der Ordnung und Gliederung einer Tonfolge nach Zeitdauer und Gewicht der Einzeltöne. Neben →Melodik und →Harmonik ein Hauptelement der Musik.

Satz. In sich selbständiger Teil eines Instrumentalwerks, z.B. einer →Symphonie, Sonate oder Suite.

Sonatenhauptsatzform. →Polyphone musikalische Form, bei der mindestens zwei Themen über zwei Teile hinweg in einem →tonartlichen Zusammenhang entwickelt werden (→Entwicklungsform). Im ersten Teil werden – ggf. nach einer Einleitung – meist zwei Themen in unterschiedlichen Tonarten vorgestellt und am Ende wiederholt. Im zweiten Teil werden die Themen und in selteneren Fällen auch Material aus der Einleitung entwickelt und schließlich in der Haupttonart wiederholt, bevor der Satz mit einer freien Coda schließt.

Stilrichtung. Im eigentlichen Sinne die Art und Weise, in der Musiker Werke in einer bestimmten Region oder einer bestimmten →Epoche aufgeführt haben. Im übertragenen Sinne auch die Eigenschaften eine Gruppe von Werken, die diese Aufführungsweisen explizit aufgreifen und in Notenschrift festhalten.

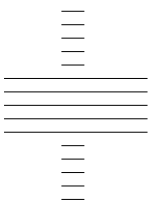
Stimme. Teil einer →Partitur, die ein Musiker (sowohl Sänger als auch Instrumentalist) auszuführen hat. Bei Streichern werden auch Gruppen von gleichartigen Instrumenten zu einer Stimme zusammengefaßt. Durch Voranstellung der Anweisung „divisi“ kann verlangt werden, daß Streicherstimmen zeitweise als getrennte Stimmen spielen sollen.

Einzeln oder zusammen mit einer anderen Stimme in einem →System notiert.

Symphonie. →Polyphones Orchesterwerk, bestehend aus einer Folge von meist drei oder vier →Sätzen in →Sonatenhauptsatzform.

System. Gruppe von fünf übereinander angeordneten parallelen, waagerechten Linien, auf und zwischen denen die Noten eingetragen werden. Falls die fünf Linien zur Notation von besonders hohen oder tiefen Noten nicht ausreichen, werden kurze Hilfslinien oberhalb oder unterhalb des Systems hinzugefügt. Die Zuordnung von Tonhöhen zu bestimmten Linien und Zwischenräumen wird durch →Notenschlüssel und →Tonartversetzungszeichen festgelegt.

Eine oder zwei →Stimmen können in einem System notiert werden.



Bsp.: leeres System mit Hilfslinien

Szene. Unterabschnitt eines →Aktes einer →Oper, die durch das Auf- und Abtreten einer oder mehrerer Personen begrenzt ist.

Takt. Musikalisches Maß- und Bezugssystem, das die Betonungsabstufung und zeitliche Ordnung der Töne regelt. Die Taktart wird am Beginn eines Stückes durch einen Bruch angegeben.

Bsp.: Betonung im 4/4-Takt. Die Hauptbetonung liegt auf dem 1., die Nebenbetonung auf dem 3. Ton

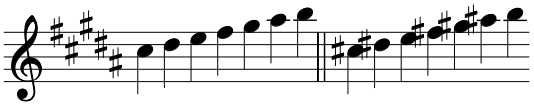
Taktstrich. Senkrechter Strich durch ein →System oder eine ganze →Akkolade, der bei der nachfolgenden Note einen Schwerpunkt anzeigt und der die →Takte abteilt.

Tempo. Die Geschwindigkeit eines Musikstücks. Kann entweder mathematisch (88 Viertelnoten pro Minute) oder menschlich (largo, andante, allegro) angegeben werden. Erst durch eine Tempoangabe können die absoluten Zeitwerte der relativ spezifizierten →Notendauern bestimmt werden.

Ton. Die elementare Einheit musikalischen Materials. Hat (im Gegensatz zu einem Geräusch) eine eindeutig bestimmbare →Tonhöhe und →Tondauer.

Tonart. Die Zugehörigkeit eines Stücks zu einer Tonart geschieht durch Untersuchung der darin verwendeten Tonhöhen. Je nachdem, wie die Töne voranschreiten, gehört das Stück zu einer Dur- oder Molltonart (→diatonische Tonleiter).

Tonartversetzungszeichen. Direkt hinter dem →Notenschlüssel notierte →Versetzungszeichen, die für alle Noten bis zu den nächsten Tonartversetzungszeichen gelten und somit individuelle Versetzungszeichen vor den Noten überflüssig machen. Charakteristisch für →Tonarten.



Bsp.: H-Dur Tonleiter mit und ohne Tonartversetzungszeichen

Tondauer. Für jeden →Ton wird in der Notenschrift normalerweise genau eine →Note gesetzt, aus deren Gestalt seine Dauer hervorgeht. Dauert der Ton länger als einen →Takt, so werden mehrere Noten identischer Höhe durch →Bindebögen verbunden.

Die absolute Tondauer wird je nach →Tempo der Musik unterschiedlich in Noten umgesetzt. Ein Ton von 1 Sekunde Länge entspricht bei einem Tempo von 60 Viertelnoten pro Minute einer Viertelnote, während er bei einem Tempo von 80 Viertelnoten pro Minute einer punktierten Viertelnote (→Verlängerungspunkt) entspricht.

Tonhöhe. Durch die Frequenz festgelegte Eigenschaft eines →Tons. Je größer die Frequenz ist, desto höher wird der Ton wahrgenommen. Aus der unendlichen Anzahl möglicher Töne werden nur Töne, die in einem bestimmten Frequenzverhältnis zueinander stehen, in der Musik verwendet und mit Tonbuchstaben (C, D, E, F, G, A, H, C) bezeichnet. Die kleinste verwendete →Tonstufe ist der →Halbton. Aus der Festlegung des Tons A auf 440 kHz können alle anderen Tonhöhen absolut bestimmt werden.

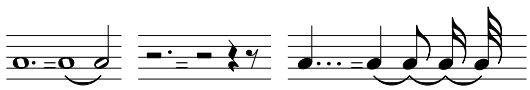
Die Tonhöhe wird in der Notenschrift durch die vertikale Anordnung einer →Note im →System ausgedrückt.

Tonleiter. Eine aufsteigende Folge von Tönen, die in einem festen Tonhöhenverhältnis (→Tonstufe) zueinander stehen: →chromatische und →diatonische Tonleiter.

Tonstufe. Tonhöhendistanz von einem Ton einer →Tonleiter zum nächsten. Kann entweder einen →Halbton oder einen Ganzton (= 2 Halbtöne) betragen.

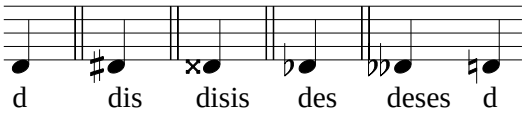
Transponierung. Das Verfahren, ein Musikstück unter Beibehaltung der ihm eigenen →Tonstufendistanzen in eine →Tonart mit einem anderen →Grundton (→diatonische Tonleiter) zu versetzen. Siehe auch transponierende →Instrumente.

Verlängerungspunkt. Zeichen zur Verlängerung der Dauer von →Noten und →Pausen um jeweils die Hälfte ihres bisherigen Wertes. Weitere Punkte verlängern um jeweils die Hälfte des vorangegangenen Punktes.



Bsp.: 3/2 7/8 15/32
=1/1+1/2 =1/2+1/4+1/ =1/4+1/8+1/16+1/32

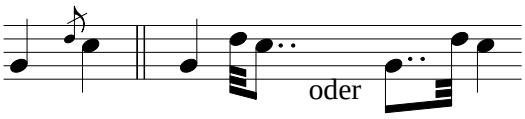
Versetzungszeichen. Zeichen zur Erhöhung oder Erniedrigung von Tönen um einen oder zwei Halbtönen, bzw. deren Aufhebung. Gelten bis zum Ende des →Taktes, in dem sie notiert sind, für alle Noten gleicher Höhe. Auch direkt hinter dem →Notenschlüssel als →Tonartversetzungszeichen notiert.



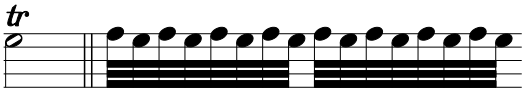
Bsp.: Die Modifikationsmöglichkeiten durch Voranstellen von Versetzungszeichen

Verzierung. Ausschmückende zusätzliche Tonfigur, die nicht zur melodischen Substanz gehört. Meist durch besondere Zeichen oder kleine Noten angedeutet.

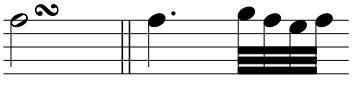
1. Vorschlag



2. Triller



3. Doppelschlag



Zwischenspiel. Verbindende Instrumentalmusik zwischen zwei →Akten einer →Oper.

Anhang B Komponententechnologien

B.1 CORBA

CORBA wurde 1991 von der OMG in seiner ersten Version (1.1) veröffentlicht, um objektorientierte Systeme über die Grenzen von Programmiersprachen, Implementationen, Betriebssystemen und Rechnern hinweg miteinander verbinden zu können. Dazu baut CORBA auf den Prinzipien des Prozedurfernaufrufs (*remote procedure call*) auf, die seit 1984 diskutiert und eingesetzt werden (vgl. [BN84]), um zwischen Rechnern potentiell unterschiedlicher Betriebssysteme nicht nur Daten auszutauschen, sondern auch die Funktionalität des jeweiligen Partners unmittelbar nutzen zu können.

Das Hauptproblem beim Prozedurfernaufruf sind die abweichenden Aufrufkonventionen unterschiedlicher Betriebssysteme, d.h. an welchem Ort Parameter in welcher Reihenfolge und in welchem Format übergeben und wieder zurückerwartet werden (vgl. [Bor99, S. 671f]). Obwohl es möglich ist, die Konventionen des entfernten Rechners individuell im Rahmen der Entwicklung zu ermitteln und dementsprechende, passende Aufrufe zu generieren, ist eine automatische Generierung des zur Konvertierung notwendigen Kompilats wesentlich einfacher. Dazu wird zunächst die Signatur der zum Fernaufruf angebotenen Prozedur mit einer plattformunabhängigen Schnittstellenbeschreibungssprache (*interface definition language, IDL*) beschrieben und von einem Präprozessor neben dem eigentlichen Kompilat der Prozedur zwei weitere Kompilate, sogenannte *stubs*, erzeugt, mit denen Aufrufer (per *client stub*) und Aufgerufener (per *server stub*) miteinander in Verbindung treten können. Der Client-Stub ist dabei auf der Plattform des Aufrufers ausführbar und dient als dortiger Stellvertreter der fernaufrufbaren Prozedur. Wird er aufgerufen, generiert er eine für die Übertragung geeignete Transportform der Argumente und sendet sie als Nachricht dem Server-Stub zu. Dieser interpretiert die Nachricht und erzeugt einen passenden lokalen Prozeduraufruf, den er anschließend ausführt. Das Ergebnis des Aufrufs wird auf dem umgekehrten Wege zurückübermittelt, wobei der Server-Stub diesmal die Nachricht erzeugt und der Client-Stub sie liest (vgl. [Müh99, S. 697f]).

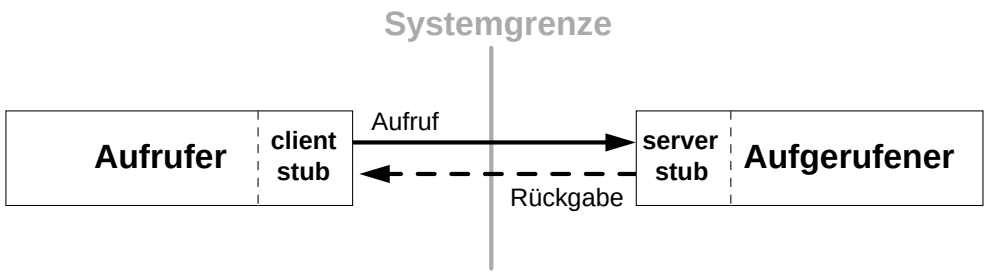


Abb. B-1 Struktur eines Prozedurfernaufrufsystems.

Dieses einfache Verfahren hat zwei Nachteile, die beide von CORBA behoben werden:

1. Bei der Stub-Generierung durch den Präprozessor muss die Plattform des Aufrufenden bereits bekannt sein und angegeben werden, so dass hinzugekommene oder geänderte Plattformen Neuübersetzungen erfordern.

2. Zusätzlich zu Operationsaufrufen in prozeduralen Sprachen müssen in objektorientierten Sprachen Informationen über den Laufzeittyp des Objekts und dessen Zustand übermittelt werden.

CORBA bietet dazu ein *interface repository* an, in dem Schnittstellenbeschreibungen plattformunabhängig aufbewahrt werden können, und Mechanismen, mit denen Laufzeitobjektreferenzen und -zustände zwischen Client und Server austauschbar sind. Um Aufrufer und Aufgerufenen besser zu entkoppeln, kommunizieren sie nicht mehr direkt miteinander, sondern über einen zwischengeschalteten *Object Request Broker (ORB)*.

Während der Entwicklung muss wie beim Prozedurfernaufruf die Schnittstelle der aufrufbaren Klasse in einer Schnittstellenbeschreibungssprache – in diesem Fall der *OMG IDL* – beschrieben werden. Server-Stubs und das Kompilat der Klasse werden in einem server-lokalen *object server* integriert und (1) die Existenz derselben in einem ORB-weit bekannten *implementation repository* registriert, (2) die Schnittstellenbeschreibung in einem ORB-weit bekannten *interface repository* veröffentlicht. Konkrete Client-Stubs werden erst bei Bedarf generiert, wenn ein Präprozessor im Quelltext des intendierten Aufrufers (unter Rückgriff auf das *interface repository*) einen Verweis auf die entsprechende Operation findet.

Zur Laufzeit wird zunächst der ORB gestartet und ihm mitgeteilt, welche Object-Server bereits verfügbar sind. Mehrere ORBs können sich auch gegenseitig bekannt gemacht werden und dann über das *Internet inter-ORB protocol (IIOP)* kooperativ Aufrufern gegenüber wie ein einziger, großer ORB auftreten. Kommt es zu einem statischen Aufruf an einem Client-Stub, wandelt dieser die übergebenen Argumente in eine geeignete Transportform inklusive einer UUID für die Laufzeitobjektreferenz und dem Objektstatus um und sendet sie an den ORB. Dieser ermittelt zunächst anhand des *implementation repository* den Object-Server, der die Operation ausführen kann, startet ihn gegebenenfalls und sendet dem Server-Stub die Transportform zu. Der Server-Stub interpretiert die Transportform, veranlasst einen lokalen Operationsaufruf und übermittelt ein eventuelles Ergebnis auf dem umgekehrten Wege. Bei einem dynamischen Aufruf ist es möglich, eine Operation erst zur Laufzeit auszuwählen, zu parametrisieren und aufzurufen, auch wenn dafür kein statischer Client-Stub vorhanden war. Der ORB kann hierbei allerdings auch erst zur Laufzeit ermitteln, ob es die gewünschte Operation überhaupt gibt und auf welchem Object-Server sie ggf. verfügbar ist.

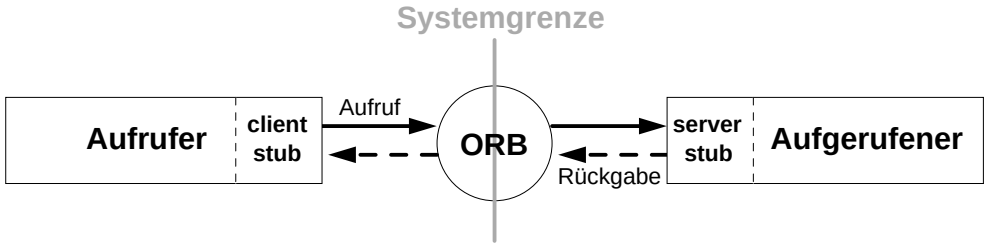


Abb. B-2 Struktur eines CORBA-Systems.

Neben dieser Kernfunktionalität CORBAs bietet die OMG noch die *Object Management Architecture (OMA)* an, die in Ergänzung zu CORBA zusätzliche Dienste (u.a. *CORBAservices*; vgl. [OMG97c])[1] enthält, die die Arbeit mit Objekten in heterogenen Systemen vereinheitlichen. Dies sind unter anderem der *Object Trader Service* (liefert Objekt-Operationen, die bestimmten wählbaren Kriterien genügen), der *Object Query Service* (liefert Objekte gemäß ihres SQL-ähnlich spezifizierbaren Zustands) und der *Persistent Object Service* (ermöglicht die sitzungsübergreifende Speicherung und später Nutzung von Laufzeitobjekten).

B.2 COM

COM wurde 1995 von Microsoft und Digital Equipment in der Version 0.9 veröffentlicht, um implementierte Funktionalität sprachunabhängig über binäre Schnittstellen zur Verfügung zu stellen. COM basiert dabei auf einem weit gefassten Dokumentbegriff, der all jenes als ein Dokument betrachtet, was vom Anwender manipuliert werden kann. Extreminterpretationen sind dabei reine Texte (auf einer Fläche angeordnete Zeichen, die durch Tippen an der Cursorposition manipulierbar sind) und Programme mit einer graphischen Benutzungsschnittstelle (auf einer Fläche angeordnete Bedienelemente, dir mit Maus und Tastatur manipuliert werden können). Dazwischen gibt es zahlreiche Mischformen, wie z.B. HTML-Formulare, die zum einen aus Text und zum anderen aus Feldern und Bedienelementen bestehen, und Sonderfälle, wie z.B. graphisch nicht repräsentierbare „Programm-Dokumente“, die z.B. einen Sortieralgorithmus implementieren (vgl. [MD95]).

Solche als *controls* bezeichneten Dokumente konnten erstmals mit den 1992 von Microsoft entwickelten *Visual Basic Controls*[2] (VBXe) erstellt werden, die aber noch keine COM-Komponenten darstellten und auf eine Visual-Basic-Laufzeitumgebung angewiesen waren. Sie wurden 1995 zu *Object Linking and Embedding Controls* (*OLE controls*, OLXe) ausgebaut, die nun selbständig lauffähig und echte COM-Komponenten waren. OLXe können, wie der Name andeutet, in andere Dokumente eingebettet und dort direkt, ihrer Art entsprechend, editiert werden *(in-place editing)*. Wird z.B. eine MS-*Excel*-Tabelle in einen MS-*Word*-Text eingebettet, so wechseln die Menüs von *Word* sich zu *Excel*-Menüs, sobald der Anwender den Cursor in die Tabelle hineinbewegt (vgl. [Szy98, S. 212]). Da OLXe zwangsweise eine sehr große Menge von Operationen an

[1] Neben den *CORBAservices* enthält die OMA auch noch die sogenannten *CORBAfacilities* und *Application Objects* wie z.B. die bereits erwähnten *Business Objects* (vgl. [OMG97b]).
[2] alternative auch *Extensions*.

insgesamt 16 Schnittstellen implementieren müssen, um diese Funktionalität zu gewährleisten, obwohl sie von der erwähnten GUI-losen Sortierkomponente gar nicht benötigt werden, wurden 1996 die mit weniger Zwangsfunktionalität belasteten *Active Controls* (ActiveXe) eingeführt, die zudem auch Ereignisse versenden können, aber genauso wie OLXe COM-Komponenten sind. Außerdem existieren zahlreiche, auf COM aufbauende und dazu abwärtskompatible 1997 hinzugekommene COM-Erweiterungen wie *Distributed COM* und *COM+* (vgl. [Szy98, S. 207ff]):

- DCOM eröffnet die Möglichkeit zur Nutzung von COM-Komponenten auf entfernten Rechnern.

- COM+ erweitert COM um eine objektorientierte Ausrichtung und eine dementsprechender Laufzeitumgebung mit Garbage-Collector.

COM ist ein reiner Binärstandard, dessen Hauptelemente die sogenannten COM-Komponenten sind.[3] Sie sind ein Kompilat, dass an einer festen Stelle mindestens eine Zeigertabelle besitzt. Die hinteren Tabelleneinträge zeigen ggf. auf weitere gleichartige[4] Zeigertabellen, der erste Eintrag zeigt auf eine Schnittstelle (siehe unten). Die dazwischen liegenden Einträge verweisen bei einer objektorientierten Quellsprache der COM-Komponente als Identitätszeiger auf die die Schnittstelle implementierende Klasse. Diese Einträge sind aber für die Verwendung der COM-Komponente vollkommen irrelevant und werden nicht nach außen gegeben. Für einen Verwender ist es also nicht erkennbar, ob die Funktionalität der Komponente direkt in Maschinensprache, in kompiliertem BASIC, in C, C++ oder Java implementiert ist.

Das einzige, was eine COM-Komponente *garantiert* aufweist, ist mindestens eine Zeigertabelle, deren erster Eintrag auf eine Schnittstelle namens `IUnknown` verweist, und die ihrerseits wiederum exakt drei Operationen anbietet, von denen `QueryInterface`, mit der die von den COM-Komponente implementierten Schnittstellen ermittelt werden können, die wichtigste ist. Auf eine COM-Komponente kann niemals insgesamt, sondern nur über ihre einzelnen Schnittstellen zugegriffen werden, die der Verwender sich über die COM-Programmbibliothek (siehe unten) verschaffen kann. Dabei kann er nicht unmittelbar erfahren, welche weiteren Schnittstellen die COM-Komponente anbietet, sondern er sieht sie *nur* aus dem Blickwinkel heraus, den die ihm vorliegende Schnittstelle gerade bietet. Nur über wiederholtes Aufrufen von `QueryInterface` der `IUnknown`-Schnittstelle aller COM-Komponenten, derer er habhaft werden kann, und der Speicherung der zurückerhaltenen Zeiger auf Schnittstellen zu Vergleichszwekken kann er feststellen, welche COM-Komponente die jeweilige Schnittstelle tatsächlich implementiert.

Eine COM-Schnittstelle ist selbst wiederum eine Tabelle mit Zeigern auf Operationen. Sie hat zu Identifikationszwecken eine *interface identificator (IID)* genannte UUID, die

[3] In Dokumentationen (vgl. [MD95], [Rog97] und [Box97]) sowie einigen softwaretechnischen Publikationen (vgl. [Szy98] und [Gri98]) werden COM-Komponenten als COM-Objekte (*COM objects*) bezeichnet, wodurch unzutreffende Assoziationen mit den nur zur Laufzeit existierenden Objekten im Sinne der Objektorientierung hervorgerufen werden. Da COM-Komponenten statisch im Kompilat vorliegen und zwecks Verwendung erst der Instantiierung bedürfen, verwende ich gemäß Begriff 1.2 die Bezeichnung „COM-Komponente“.

[4] Nur die erste Zeigertabelle besitzt am Ende ggf. mehrere Zeiger auf alle anderen Zeigertabellen, die restlichen Zeigertabellen am Ende genau einen Zeiger auf die erste Zeigertabelle.

von den aus Bequemlichkeit unverbindlich zugeordneten Namen vollkommen unabhängig ist. Die erwähnte `IUnknown` genannte Schnittstelle wird tatsächlich ausschließlich über ihre IID `00000000-0000-0000-C000-000000000046` angesprochen. Jede Schnittstelle hat garantiert als erste drei Einträge Zeiger auf die „`QueryInterface`“, „`AddRef`“ und „`Release`“ genannten Operationen, deren Namen ebenfalls nur die Lesbarkeit erhöhende Konventionen sind, die nicht im Kompilat selbst auftauchen. `AddRef` und `Release` dienen allein der Freispeicherverwaltung, die hier nicht von Interesse ist. `QueryInterface` akzeptiert als Argument eine IID und ermittelt, indem es (1) die Zeigertabellen der COM-Komponente traversiert, (2) dem jeweils ersten Eintrag folgt und (3) die IID der dort liegenden Schnittstelle prüft, ob die COM-Komponente die gewünschte Schnittstelle implementiert oder nicht. Im ersten Fall liefert es einen Zeiger auf die Schnittstelle, im zweiten Fall eine Fehlermeldung zurück.

Über die drei genannten Operationen hinaus kann die die Schnittstelle ausmachende Zeigertabelle beliebig viele weitere Operationszeiger enthalten, die jedoch anstelle eines Verweises auf die Implementation der Operation leer sein können, was bei einem Aufrufversuch zur Fehlermeldung `E_NOTIMPL` führt.

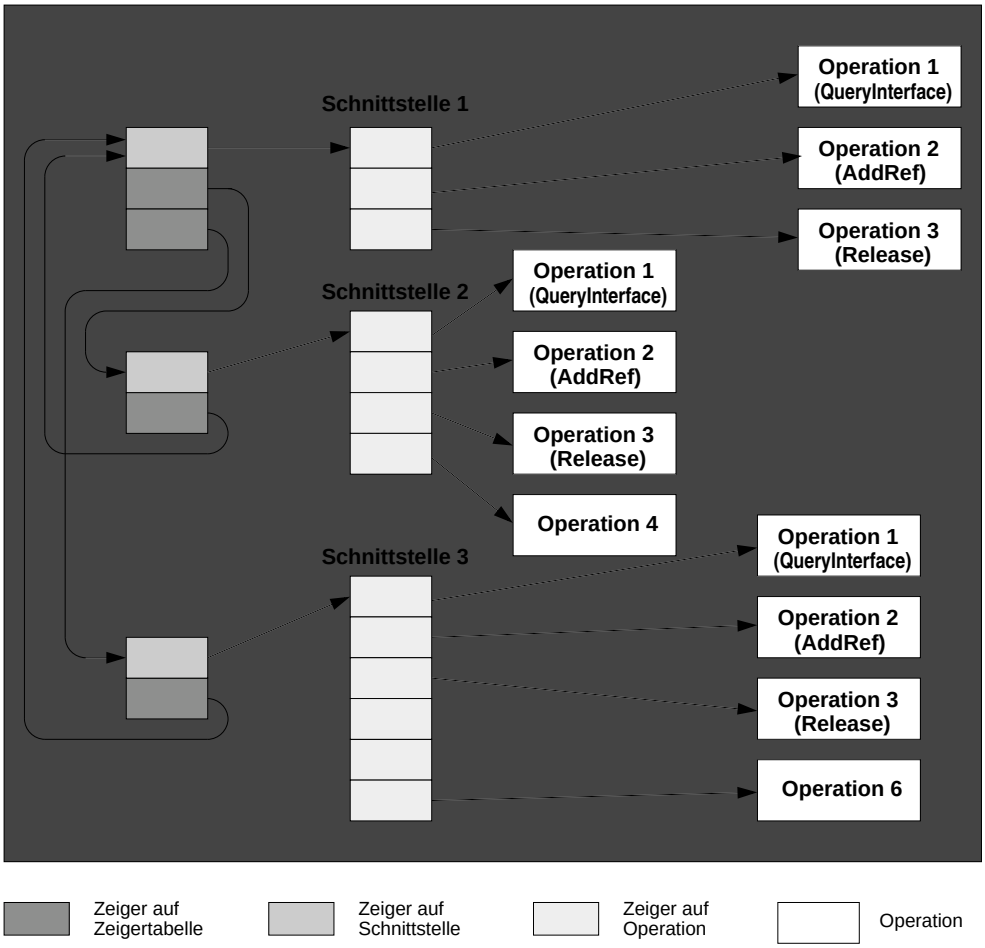


Abb. B-3 Struktur einer COM-Komponente.

Um COM-Komponenten semantisch anhand ihrer Schnittstellen zu gruppieren, gibt es von Microsoft bestimmte Kategorien (*categories*), die jeweils wiederum durch eine CATID genannte UUID identifizierbar sind. Eine COM-Komponente gehört einer Kategorie genau dann an, wenn sie die für die Kategorie vorgeschriebe-

nen Schnittstellen enthält und keine von deren Operationen unimplementiert lässt. Kategorien sind z.B. `40FC6ED9-2438-11CF-A3DB-080036F12502` („`_Printable Objects`“) oder `7DD95802-9882-11CF-9FA9-00AA006C42C4` („`Controls safely initializable from persistent data`“).

Da die Tatsache, dass eine COM-Komponente eine bestimmte Anzahl von Schnittstellen implementiert, noch nichts über die Art und Weise aussagt, in der die dort angebotenen Dienste erbracht werden, gibt es zur Unterscheidung sogenannte Klassen[5] (*classes*), deren *class identificator* (CLSID, erneut ein UUID) allen COM-Komponenten gemein ist, die einen Dienst in gleicher Weise erbringen. Am Beispiel verschiedener MDV-Subsysteme hätten z.B. alle Laufzeitkomponenten auf syntaktischer Ebene die gleichen Schnittstellen (Signaturen), aber je nachdem, ob sie mit *Humdrum* und *CMAP*, mit einem Alternativsystem oder ganz ohne MDV-System arbeiten, unterschiedliche CLSIDs. Die CLSID ist fester Bestandteil jeder COM-Komponente.

Die bis hierher beschriebenen Eigenschaften von COM-Komponenten sagen noch nichts darüber aus, in welcher Form sie ausgeliefert werden bzw. ihre Funktionalität in Anwendungssysteme eingebunden werden kann. Zu diesem Zweck gibt es COM-Server, die eine oder mehrere COM-Komponenten enthalten können und zu jeder dieser Komponenten eine zusätzliche Erzeugungskomponente (*class factory*)[6] besitzen, die – in der Manier eines Konstruktors in objektorientierten Sprachen – die elementaren Initialisierungen ausführt, die vor deren erster Verwendung erforderlich sind, d.h. mindestens die Belegung des notwendigen Hauptspeicherplatzes. Für die eigentliche Herstellung eines fachlich sinnvollen Zustandes hält COM einen separaten, weiter unten erklärten Mechanismus bereit.

Ein COM-Server kann entweder als *inprocess server* in der Form einer *dynamic link library* (`.DLL`) oder als separater *local server* in Form einer ausführbaren Programmdatei (`.EXE`) vorliegen. Bei DCOM sind diese Server, wenn sie von einem entfernten Rechner aus genutzt werden sollen, dann als *remote server* eingestuft.[7] Ein COM-Server kann von jedem dazu geeigneten Übersetzer erzeugt werden. Die Sprache, in der das Programm abgefasst ist, spielt dabei keine Rolle und kann vom späteren Verwender der enthaltenen COM-Komponenten nicht mehr erkannt werden. Wird aus einer typisierten Sprache heraus übersetzt, kann der Übersetzer zusätzlich eine *type library* integrieren, aus der die Signaturen aller enthaltenen Schnittstellen hervorgeht und die zur Laufzeit zur Typprüfung herangezogen werden kann. Die *type library* ist jedoch für alle Sprachen optional und muss, selbst wenn sie erzeugt wird, nicht genutzt werden (vgl. [Szy98, S. 208f]).

[5] Wie auch in Fußnote 3 auf Seite XVIII bezüglich des Begriffs „Objekt“ gesagt, hat eine Klasse im Zusammenhang mit COM nichts mit einer Klasse im Sinne des objektorientierten Begriffsgebäudes zu tun, sondern dient in einem allgemeineren Sinne der *Klass*ifizierung von COM-Komponente.

[6] Erneut eine irreführende Bezeichnung, denn erzeugt wird keine Klasse (weder im objektorientierten noch im abstrakt gruppierenden Sinne einer CLSID), sondern eine einem objektorientierten *Objekt* vergleichbare Laufzeiteinheit.

[7] Bei DCOM werden ähnlich wie bei CORBA *stubs* und ein Transferformat verwendet. Dieses Aspekte sind jedoch – im Gegensatz zu CORBA – kein Kernaspekt von COM und somit im Rahmen dieser Untersuchung ohne Belang.

Um schließlich zu erfahren, wo welche COM-Komponenten mit welcher CLSID (und ggf. auch CATID) in welcher Form vorliegen, muss auf jedem Rechner, auf dem COM-Komponenten nutzbar sein sollen, eine Systemregistratur (*system registry*) existieren, die zu allen CLSIDs aller vorhandenen COM-Server einen Eintrag enthält aus dem hervorgeht, welcher COM-Server die COM-Komponente mit welcher CLSID implementiert und welcher Art dieser COM-Server ist. Falls kein direkter COM-Server vorhanden ist, muss ein Eintrag – z.B. „`TreatAs`“ – gesetzt sein, der Auskunft darüber gibt, unter welcher Ersatz-CLSID nach einem geeigneten COM-Server gesucht werden soll.

```
CLSID
   {00020900-0000-0000-C000-000000000046} = Microsoft Word 6.0 - 7.0 Dokument
      TreatAs = {00020906-0000-0000-C000-000000000046}
   {00020906-0000-0000-C000-000000000046} = Microsoft Word Dokument
      InprocHandler32 = C:\Windows\ole32.dll
      LocalServer32 = "C:\Programme\Microsoft Office\Office\Winword.exe"
```

Damit ein Verwender eine COM-Komponente nutzen kann, muss er sich der prozeduralen COM-Programmbibliothek (COM API) bedienen. Die einzelnen Schritte sind dabei die folgenden (vgl. [MD95]):

1. Der Verwender ruft die Methode `CoCreateInstance` mit einer IID und einer CLSID auf, so dass sowohl die syntaktische Schnittstelle als auch die gewünschte Dienstausgestaltung festgelegt ist.

2. `CoCreateInstance` prüft unter Rückgriff auf die Systemregistratur, ob ein COM-Server für die angegebene CLSID vorhanden ist (unmittelbar oder durch ggf. mehrfaches Verfolgen von „`TreatAs`“-Verweisen) und falls ja, ob der COM-Server die gewünschte Schnittstelle enthält. Sind sowohl COM-Server als auch Schnittstelle vorhanden, wird der Server – falls noch nicht getan – geladen (*inprocess server*, `.DLL`) oder gestartet (*local server*, `.EXE`).

3. `CoCreateInstance` ruft die zur COM-Komponente gehörige Erzeugungskomponente auf und führt die bereits beschriebene, rudimentäre Initialisierung durch.

4. Als Rückgabewert erhält der Verwender entweder einen Zeiger auf die gewünschte Schnittstelle oder, falls in Schritt 2 oder 3 Fehler auftraten, eine Fehlermeldung.

5. Der Verwender kann nun wahlweise eine (wiederum wählbare) fachlich sinnvolle Initialisierung vornehmen und aus einer beliebigen Datenquelle einen persistent gemachten Initialisierungszustand beziehen und diesen einer geeigneten Operation der soeben gewonnenen Schnittstelle übergeben. Als Quelle kommen dabei sowohl einfache Dateien, Ströme oder komplexe *structured storages* in Frage, die im Inneren wie eine Archivierungsdatei beliebige Unterverzeichnisse mit Dateien beliebiger Formate enthalten können (vgl. [Szy98, S. 206]).

Neben `CoCreateInstance` enthält die COM-API zahlreiche weitere Prozeduren, mit denen z.B. alle verfügbaren COM-Komponenten auflistbar sind.

Hat ein Verwender eine Referenz auf ein Schnittstelle, kann er sich – wie anfangs geschildert – nicht sicher sein, dass die dahinterstehende COM-Komponente noch irgend-

welche zusätzlichen Schnittstellen implementiert. Er kann nur `QueryInterface` zu Prüfzwecken mit gewünschten IIDs aufrufen, muss aber stets gewappnet sein, dass die Schnittstelle nicht vorhanden ist, was unschöne, kaskadierende Laufzeittypprüfungen zur Folge hat, wie sie sich im folgenden, lediglich zur besseren Lesbarkeit umformatierten Pseudocode-Beispiel aus [MD95] zeigen:

```
BOOL SaveObject( IUnknown *pUnkObj )
 {
  pUnkObj->QueryInterface( IID_IPersistStorage )
  if( success )
   {
    create a storage element for the object
    call IPersistStorage::Save
    call IPersistStorage::Release
    return TRUE
   }

  //All other cases use a client-controlled stream
  create a stream element for the object in some storage

  //IPersistStorage not supported, try IPersistStream
  pUnkObj->QueryInterface( IID_IPersistStream )
  if( success )
   {
    call IPersistStream::Save
    call IPersistStream::Release
    return TRUE
   }

  //IPersistStream not supported, try IPersistFile
  pUnkObj->QueryInterface( IID_IPersistFile )
  if( success )
   {
    //Save to a temp file
    call IPersistFile::Save( "objdata.tmp" );
    call IPersistFile::Release
    read data from temp file
    write data to the stream
    return TRUE
   }

  //All else failed, try IDataObject
  pUnkObj->QueryInterface( IID_IDataObject )
  if( success )
   {
    call IDataObject::EnumFormatEtc
    call IEnumFORMATETC to get the first format (assume it's native)
    call IEnumFORMATETC::Release
    call IDataObject::GetData for the format, asking for global memory
    call IDataObject::Release
    Lock global memory and write to stream
    Free global memory
    return TRUE
   }

  //Everything failed, so give up
  destroy stream we created:  not using it.
  return FALSE
 }
```

B.3 Plug-Ins

Plug-Ins sind spezielle dynamische Bibliotheken (engl. *dynamic link libraries*), mit deren Hilfe Anwender die Funktionalität eines Anwendungssystems erweitern können. Im Gegensatz zu CORBA und COM gibt es für Plug-Ins keine einheitliche Technologie. Es handelt sich vielmehr um eine Gruppe ähnlicher, aber proprietärer Technologien, die das Konzept der Einsteck-Erweiterung auf individuelle Weise umsetzen. Als Beispiel dienen mir im in diesem Abschnitt Plug-Ins für den *Netscape Navigator* und das *GNU Image Manipulation Program* (GIMP), die sich hinsichtlich ihrer Funktionalität sowie Typisierung und Black-Box-Bildung recht stark unterscheiden.

Im Navigator dienen Plug-Ins ausschließlich zum Anzeigen von Hypertext-Bestandteilen, die nicht in der *Hypertext Markup Language* (HTML) abgefasst sind, und auch nicht den Graphikstandards GIF- oder JPEG entsprechen. Ein Plug-In kann dabei einen oder mehrere Arten von über ihren MIME-Typ[8] identifizierten Bestandteilen anzeigen. In GIMP werden Plug-Ins für vier verschiedene Arten von Funktionalität eingesetzt: Für Graphikfilter, Dateiimport- und Dateiexportroutinen sowie für Bildakquistionsdienste wie Screenshot- und Scanner-Tools.

Generell geht die Benutzung eines Plug-Ins in drei Schritten von Statten, die ich zunächst allgemein und dann speziell für Netscape und GIMP vorstellen werde:

1. **Bezug durch den Anwender.** Der Anwender muss sich das als ein für seine Plattform geeignetes Kompilat vorliegende Plug-In verschaffen und in einem vom Anwendungssystem vorgegebenen, speziell für Plug-Ins vorgesehenen Verzeichnis ablegen.

2. **Registrierung des Plug-Ins beim Anwendungssystem.** Beim Start des Anwendungssystems liest dieses das besagte Verzeichnis und prüft alle darin enthaltenen Plug-Ins. Es ermittelt den Typ des Plug-Ins sowie – sofern vorhanden – weitere Informationen. Schließlich wird das Plug-In dynamisch zum Anwendungssystem hinzugebunden und dessen Bedienschnittstelle so erweitert, dass die Funktionalität des Plug-Ins genutzt werden kann.

3. **Inanspruchnahme des Plug-Ins durch das Anwendungssystem.** Wird die im Plug-In enthaltene Funktionalität benötigt, ruft das Anwendungssystem das Plug-In über einen internen Operationsaufruf auf und übermittelt ihm die notwendigen Daten. Teilweise kann das Plug-In zur Erledigung seiner Aufgabe auch weitere Daten vom Anwendungssystem oder anderen Plug-Ins beziehen. Wie bei einem regulären Operationsaufruf kehrt der Kontrollfluss am Ende wieder zum Anwendungssystem zurück.

---

[8] Das Format der *Multipurpose Internet Mail Extensions* (MIME) wurde ersonnen, um auch
1. Nicht-Standard-ASCII-Zeichen (wie z.B. deutsche Umlaute) in E-Mails übertragen zu können und
2. Nicht-Textdateien als Anhänge von E-Mails versenden zu können.

Dazu geht jedem Nachrichtenteil eine Zeile voraus, die dessen *content type* angibt und vom Empfänger genutzt werden muss, um die nachfolgenden Daten zu interpretieren. Der *content type* ist zweiteilig, wobei der erste Teil die generelle Datenart (`text`, `image`, `audio`, `video`, etc.) und der zweite Teil die konkrete Unterart angibt (`text/html`, `image/tiff`, `audio/aiff`, `video/mpeg`, etc.). Durch diese *content types*, die weltweit zentral bei der *Internet Engineering Task Force* (IETF) registriert werden müssen, sind eindeutigere Formatangaben möglich als mit den von Plattform zu Plattform unterschiedlichen Identifikationsschemata wie z.B. Suffixen von Dateinamen.

Für Netscape stellt sich dieser Prozess wie folgt dar:

1. **Bezug durch den Anwender.** Je nach Plattform muss das Plug-In in einem besonderen Verzeichnis abgelegt werden. Die Informationen über die darstellbaren MIME-Typen sind plattformabhängig auf unterschiedliche Arten codiert (vgl. [Net98a, S. 23ff]):

   - *Windows*: in der Versionsinformation der `.DLL`-Datei;
   - *MacOS*: in den dazugehörigen Ressourcen `'STR#' 126` bis `128;`
   - *UNIX*: in der Plug-In-internen Prozedur `NPP_GetMimeDescription`[9]).

2. **Registrierung des Plug-Ins beim Anwendungssystem.** Beim Start liest der *Navigator* die Informationen über die darstellbaren MIME-Typen des Plug-Ins und registriert es in der selben Tabelle, in der auch die externen *helper applications* eingetragen sind. Unter UNIX muss das Plug-In dazu bereits geladen und `NPP_GetMimeDescription` aufgerufen werden.

3. **Inanspruchnahme des Plug-Ins durch das Anwendungssystem.** Lädt der Anwender ein HTML-Dokument, das Daten in einem der MIME-Typen enthält, für die das Plug-In registriert ist, lädt der *Navigator* – sofern noch nicht geschehen – das Plug-In und ruft `NPP_Initialize` auf. Für jedes Nicht-HTML-Element des Hypertext-Dokuments erzeugt der Navigator eine auf externe Dateien, Datenblöcke oder einen Ströme gestützte Datenquelle (vgl. [Net98a, S. 65f]). Zu jeder Datenquelle wird dann mit `NPP_New` ein eigenes Laufzeitexemplar des passenden Plug-Ins erzeugt (vgl. [Net98a, S. 7f]) und ihm die Daten mit `NPP_Write` übermittelt. Jedes Plug-In muss eine aus 16 Prozeduren bestehende, einheitliche Schnittstelle anbieten, über die es Anwenderereignisse (Mausklicks, Zeicheneingaben, etc.) und Meldungen über verfügbare URLs entgegennehmen kann (vgl. [Net98a, S. 102f]). Umgekehrt kann jedes Plug-In die 22 Prozeduren umfassende Plug-In-API des *Navigators* nutzen, um Daten zurückübermitteln, Versionsinformationen auszulesen, Speicher zu beantragen sowie Daten von URLs zu erbitten und an sie zu versenden (vgl. [Net98a, S. 103f]).

Bei GIMP kommt dem Anwender eine aktivere Rolle zu (vgl. [Tur99]):

1. **Bezug durch den Anwender.** Aufgrund der zahlreichen UNIX-Varianten müssen Plug-Ins meist im Quelltext bezogen und vom Anwender übersetzt werden.

2. **Registrierung des Plug-Ins beim Anwendungssystem.** Im Gegensatz zu Netscape-Plug-Ins fußen *GIMP*-Plug-Ins stark auf statisch nicht prüfbaren Konventionen:

   - Zum einen sind sie ausschließlich dynamisch typisiert und besitzen allesamt die selbe durch die Struktur `_GplugInInfo` definierte statische Schnittstelle, obwohl es *GIMP*-Plug-Ins für vier verschiedene Arten von Funktionalität gibt.

[9] Alle Prozeduren der Netscape Plug-In-API beginnen mit `NP`. Der dritte Buchstabe ist entweder ein `P` für eine Prozedur an der Schnittstelle des Plug-Ins oder `N` für eine Prozedur an der Schnittstelle des Navigators.

```
struct _GplugInInfo
 {
  void ( *init_proc )( void ); // Anfaengliche Initialisierung.
  void ( *quit_proc )( void ); // Letztmaliges Aufraeumen.
  void ( *query_proc )( void ); // Aufforderung zum Selbsteintrag
                                // in die Prozedurdatenbank.
  void ( *run_proc )( char   *name,   // Starten der eigentlichen
                      int    nparams, // Funktionalität.
                      GParam *param,
                      int    *nreturn_vals,
                      GParam **return_vals);
 };
```

Die Signatur der zentralen Operation „`run_proc`" ist dabei weder hinsichtlich ihres Namens noch der Anzahl oder des Typs ihrer Parameter bestimmt. Da die Schnittstelle dynamisch aus einer Folge von `GparamDef`-Einträgen herausgelesen werden muss, sind keinerlei Passformprüfungen zur Konfigurationszeit möglich.

```
struct GparamDef
 {
  GparamType type;
  char *name;
  char *description;
 };
```

```
static GParamDef params[] = // Beispieldefi-
 {                          // nition einer
  { PARAM_INT32,            // Prozedur mit
    "run_mode",             // 4 Parametern
    "Interactive, non-interactive" },
  { PARAM_IMAGE,
    "image_id",
    "(unused)" },
  { PARAM_DRAWABLE,
    "drawable_id",
    "Drawable to draw on" },
  { PARAM_COLOR,
    "fgcolor",
    "Color to draw with"}
 };
```

- Zum anderen gründet sich die Kommunikation zwischen Anwendungssystem und Plug-Ins wesentlich stärker als bei Netscape-Plug-Ins auf Konventionen anstatt auf feste Protokolle. Das *GIMP*-System erwartet beispielsweise, dass die Prozedur „`query_proc`" während der Registrierung ihrerseits wiederum die Prozedur `gimp_install_procedure` der GIMP-API aufruft und sich selbst in die Plug-In-Prozedur-Datenbank einträgt und dabei unter anderem auch über das Präfix des siebten Parameters (`char *menu_path`) angibt, welchen Typ es hat („`<Image>`", „`<Load>`", „`<Save>`", „`<Toolbox>`"). Im Gegenzug stehen *GIMP*-Plug-Ins nicht nur wie Netscape-Plug-Ins lediglich 22 ausgewählte, sondern sämtliche (über 200) Operationen der *GIMP*-API zur Verfügung, mit der das System sogar heruntergefahren werden könnte.

3. **Inanspruchnahme des Plug-Ins durch das Anwendungssystem.** Wählt der Anwender den im Rahmen der Registrierung erzeugten Menüeintrag aus, wird die „`run_proc`" des jeweiligen Plug-Ins mit den bei der Selbstregistrierung angegebenen Parametern aufgerufen. Je nach *runmode* (1. Parameter), läuft das Plug-In entweder ohne weitere Nachfragen (`RUN_NONINTERACTIVE` – z.B. zum Laden von Graphiken) oder baut in Eigenregie unter Rückgriff auf die GIMP-API ein Fenster auf, über das es dem Anwender weitere Einstellungen abverlangt (`RUN_INTERACTIVE` – z.B. bei Filtern). Ist die Aufgabe des Plug-Ins erledigt, gibt es die bei der Registrierung angegebenen Parameter zurück und gibt den Kontrollfluss an das GIMP-Anwendungssystem zurück.

B.4 Java

Die hybrid objektorientierte Programmiersprache Java wurde von Sun Microsystems entwickelt und 1996 in der Version 1.0 veröffentlicht. Java führt zwar keine softwaretechnischen Neuerungen ein, kombiniert aber Verbesserungen auf dem Gebiet (1) der Sprachdefinition, (2) der Plattformunabhängigkeit sowie der (3) binären Auslieferung und des dynamischen Bindens, so dass die objektorientierte Softwareentwicklung beträchtlich vereinfacht wird. Aufgrund der unter (3) genannten Punkte kann Java als Laufzeitkomponententechnologie gelten, da dynamisches Binden nicht wie bei herkömmlichen Programmiersprachen dem Betriebssystem überlassen, sondern von der Java-Laufzeitumgebung realisiert wird. Da dieser Prozess teilweise auf Besonderheiten aus den Bereichen (1) und (2) fußt, stelle ich diese anfangs kurz vor.

(1) Sprachdefinition. Java erlaubt es, Typ- und Klassenhierarchien getrennt aufzubauen, so dass aus Implementationsvererbung resultierende Schwierigkeiten wie das *Problem der änderungsempfindlichen Oberklasse* (siehe Abschnitt 5.1) vermeidbar sind. Während dies auch in anderen objektorientierten Sprachen wie C++ möglich ist, verfügt Java mit `interface` über ein explizites Schlüsselwort, mit dem reine Schnittstellenbeschreibungen gekennzeichnet werden können. Mehrfachimplementationsvererbung ist nicht möglich, aber jede Klasse kann beliebig viele Schnittstellen implementieren (vgl. [GJS96, S. 183ff]) Sowohl Schnittstellen als auch Klassen können unter semantischen Gesichtspunkten in *packages* gruppiert werden, die gleichzeitig auch einen eigenen Namensraum bilden (vgl. [GJS, S. 113f]). Neben der Klasse `de.jwam.lang.contract.Contract`, die das Meyer'sche Vertragsmodell implementiert (vgl. [Mey97, S. 311ff]), könnte es daher noch weitere Klassen mit dem Namen `Contract` geben, sofern sie sich in einem anderen Paket befinden.

(2) Plattformunabhängigkeit. Der Kern der Plattformunabhängigkeit Javas besteht darin, dass Java-Übersetzer keine plattformabhängige Maschinensprache, sondern eine *bytecode* genannte Maschinensprache für eine stets einheitliche virtuelle Maschine (VM) erzeugt (vgl. [LY99, S. 3]). Die VM kann als Hardware realisiert werden, ist aber meistens eine reine Softwarelösung. Java-Programme sind auf jeder Plattform ausführbar, auf der es eine VM gibt. Die zwei weiteren Elemente der Plattformunabhängigkeit sind das einheitliche *application programming interface* (API) und Art des Aufrufs der VM aus dem jeweiligen Betriebssystem heraus:

- Die VM kapselt das zugrundeliegende Betriebssystem vollkommen, so dass im Gegensatz zu anderen Programmiersprachen keine plattformlokalen Bibliotheken vorhanden sind, um spezielle Funktionalität zu nutzen. Stattdessen gibt es eine von Sun Microsystems ständig weiterentwickelte Java API, die von Mathematik bis MIDI einen Großteil der von den meisten Entwicklern benötigten Funktionalität auf allen VM-Plattformen zur Verfügung stellt. Besonders hilfreich ist die Möglichkeit, mit dem *Abstract Window Toolkit* (`java.awt`) Bedienschnittstellen für alle Plattformen nur einmal entwickeln zu müssen und auf plattformunabhängige Weise auf das Dateisystem zuzugreifen (`java.io.File`), ohne plattformlokale Konventionen z.B. für Pfadtrennzeichen be-

rücksichtigen zu müssen („/“ unter UNIX, „\“ unter MS-DOS bzw. Windows und „:“ unter MacOS).

- Programme erhalten ihre Parameter nicht über plattformabhängige Mechanismen (z.B. mit „-“ beginnende Optionen unter UNIX und mit „/“ beginnende Optionen unter MS-DOS), sondern einheitlich über *Properties*, die der VM beim Start übergeben werden.

Plattform	**Shell-Kommando**
MS-DOS	`caldendar.exe /Y:2003`
UNIX	`calendar –y 2003`
Java *alle Plattformen*	`java –Dyear=2003 calendar`

(3) Binäre Auslieferung. Die kleinsten Java-Auslieferungseinheiten sind die in *bytecode* vorliegenden *class files*, die jeweils genau eine kompilierte Klasse oder Schnittstelle enthalten. Aufgrund ihrer beschränkten Größe lassen sich diese Class-Files nicht direkt mit herkömmlichen Objektdateien vergleichen, die ausführbaren Code für beliebig viele Klassen oder Prozeduren enthalten können (vgl. [Mös99, S. 728]). Zur gebündelten Auslieferung von Class-Files dienen in Java sogenannte *Java Archives* (JARs), die eine Weiterentwicklung von herkömmlichen Bibliotheken darstellen (vgl. [Sun98a]). Ähnlich wie eine Bibliothek besitzt ein JAR zunächst einmal eine Reihe kleinerer Auslieferungseinheiten – in diesem Fall Class-Files. Zusätzlich kann es aber auch beliebig viele, ggf. hierarchisch angeordnete, sonstige Dateien enthalten, die zur Laufzeit von den Class-Files benötigt werden – beispielsweise Graphiken, Texte oder Akustik-Dateien. Alle Bestandteile können über die Java-API auch einzeln ausgelesen und verwendet werden (`java.util.jar.JarFile`). Zwei Eigenschaften von JARs sind speziell auf die Auslieferung über das Internet zugeschnitten:

- JARs enthalten ihre Bestandteile stets in komprimierter Form, um Übertragungszeiten so gering wie möglich zu halten.

- JARs können digital signiert werden,[10] um ihren Hersteller zweifelsfrei zu identifizieren (vgl. [Sun98b]). Da über das Internet bezogene JARs sich potentiell auch bösartig verhalten können, besteht die Möglichkeit, in Abhängigkeit von der Bezugsquelle und dem Hersteller relativ fein gegliedert festzulegen, welche Operationen für Klassen aus dem JAR gestattet sind und welche nicht (vgl. [Sun98c] und [Sun98d]). Beispielsweise kann allen JARs, die von `http://www.gefaehrlich.de/` stammen, der schreibende Zugriff auf Speichermedien verwehrt werden, außer sie tragen die digitale Signatur von „Hans Sicher“.

Ein JAR, das mehrere Class-Files bündelt, stellt eine Laufzeitkomponente dar, weil sie sitzungsweise zu einem Java-Programm zusammengestellt bzw. zur Laufzeit zu einem bereits aktiven Programm hinzugebunden werden können. Im ersten Fall

[10] Digitale Signaturen sind ein unter anderem in Deutschland und den USA gesetzlich anerkanntes Mittel zum Ausweis der Identität einer Person (vgl. [San01]).

reicht es aus, wenn die JARs beim Start der VM spezifiziert werden. Im zweiten Fall muss innerhalb des Programms ein Klassenlader erzeugt werden, der das JAR zunächst prüft und dann eingliedert (`java.lang.ClassLoader`). In beiden Fällen ist das genaue Verfahren, nach dem die VM die Class-Files aus dem JAR eingliedert, nicht vorgeschrieben. Es muss lediglich sichergestellt sein, dass eine Klasse bzw. Schnittstelle vollständig geprüft ist, bevor sie instantiiert wird (vgl. [LY99, S. 50]). Dies umfasst die folgenden drei Prüfstufen (vgl. [LY99, S. 158ff]):

1. Die VM muss überprüfen, ob sie das benötigte Class-File an den Orten finden kann, die ihr als Aufbewahrungsort für Class-Files genannt worden sind,. Diese Orte können unter anderem sowohl JARs, Verzeichnisse innerhalb des lokalen Dateisystems sein als auch durch URLs[11] spezifizierte Verzeichnisse auf entfernten Rechnern aus dem Intra- oder Internet.

2. Die VM muss den *bytecode* des Class-Files auf seine technische Integrität hin untersuchen. Dies umfasst unter anderem Übertragungsfehler, Versionskompatibilität zur VM, Übereinstimmung zwischen dem Namen des Class-Files und dem im *bytecode* enthaltenen Namen der Klasse bzw. Schnittstelle (vgl. [LY99, S. 141]).

3. Die VM muss sicherstellen, dass ggf. vorhandene Oberklassen oder -schnittstellen der zu ladenden Klasse bzw. Schnittstelle ebenfalls den Schritten 1 bis 3 geladen und geprüft worden sind. Erst nachdem dieser rekursive Prozess abgeschlossen ist, ist die Klasse oder Schnittstelle selbst vollkommen geladen und steht dem Programm zur Verfügung.

[11] Im Gegensatz zu älteren Internet-Dienstprogrammen, die nur ein Internet-Protokoll wie z.B. `ftp`, `telnet` oder `news` beherrschen, ist es mit Web-Browsern möglich, verschiedene Internet-Dienste zu nutzen. Um beim Verbindungsaufbau angeben zu können, über welches Protokoll mit einem Knoten kommuniziert werden soll und die wesentlichen Einzelparameter der individuellen Dienste einheitlich zu bündeln, existieren *Uniform Resource Locators*. Ihr allgemeines Format ist dabei `protokoll://benutzer:passwort@knoten:port/pfad#sprungmarke`. [ST00, S. 847] Class-Files können dementsprechend über alle auf Dateiübertragung ausgelegten Protokolle wie z.B. `ftp`, `http` und `gopher` bezogen werden

Lebenslauf

12. Februar 1969	Geburt in Bensberg als einziges Kind der Eheleute Horst und Sylva Kornstädt
1978 – 1988	Besuch des Dietrich-Bonhoeffer-Gymnasiums in Quickborn. Abitur im Juni 1988
Juli 1988 – September 1989	Grundwehrdienst als Feuerleitsoldat bei der Raketenartillerie in Kellinghusen
Oktober 1989 – August 1996	Studium der Informatik an der Universität Hamburg. Verleihung des Grades Diplom-Informatiker im August 1996
September 1996 – November 1997	Forschungsaufenthalt an der Stanford University auf Einladung des dortigen Center for Computer Assisted Research in the Humanities (CCARH)
Dezember 1998 – Juni 2000	Forschungsaufenthalt am Arbeitsbereich Softwaretechnik des Fachbereichs Informatik der Universität Hamburg
Juli 2000 – Januar 2001	Erneuter Forschungsaufenthalt an der Stanford University auf Einladung des CCARH
seit 1991	Selbständige Tätigkeit für die Firma SW Datentechnik (Quickborn), die Körber-Stiftung, Apcon WPS (beide Hamburg) sowie das CCARH. Zunächst als Programmierer im Bereich Echtzeitsysteme, dann Erstellung Web-basierter Dienste und Web-Administration. Schließlich Weiterentwicklung des *JWAM*-Rahmenwerks
seit Februar 2001	Software-Architekt bei der Firma Apcon WPS in Hamburg

Hiermit versichere ich an Eides statt, dass ich die vorliegende Dissertationsschrift eigenhändig und ohne Zuhilfenahme weiterer als der angegebenen Hilfsmittel angefertigt habe und dass ich an keinem anderen Fachbereich einen Antrag auf Eröffnung eines Promotionsprüfungsverfahrens gestellt habe.

Hamburg, im Oktober 2001