Entwicklung verteilter Softwaresysteme: Integration von Verteilung, Nebenläufigkeit und Persistenz

Marko Boger

Vom Fachbereich Informatik
der Universität Hamburg
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

Betreuer und 1. Gutachter:	Prof. Dr. Winfried Lamersdorf
2. Gutachter:	Prof. Dr. Claudia Linnhoff-Popien
3. Gutachter:	Dr. Daniel Moldt
Tag der Disputation:	12. Januar 2001

Inhaltsverzeichnis

Zusammenfassung

Verteilte Systeme finden eine wachsende Bedeutung auf dem Gebiet der Softwareanwendungen. Doch die Entwicklung von Anwendungen für solche verteilten Systeme gilt als einer der komplexesten Bereiche der Softwareentwicklung. Zusätzlich zu vielen Problemen, die schon in zentralisierten Softwaresystemen bewältigt werden müssen, erfordern verteilte Systeme die Behandlung der Aspekte Verteilung, Nebenläufigkeit und Persistenz. Nach heutigem Stand der Technik werden diese Aspekte mit orthogonalen Konzepten bewältigt, so dass die Entwickler Techniken für jeden dieser Bereiche verstehen und beherrschen müssen. Daraus resultiert eine enorme Komplexität, die die Entwicklung verteilter Systeme so schwer macht.

Ziel dieser Arbeit ist, die Komplexität der Programmierung verteilter Softwareanwendungen zu reduzieren. Dazu wird in dieser Arbeit untersucht, ob eine abstraktere Sicht möglich ist, die diese Aspekte vereinigt. Es wird ein Konzept entwickelt, das alle drei Aspekte gleichermaßen behandelt. Daraus ergibt sich eine Reduktion der Komplexität in der Entwicklung und eine Erhöhung der Flexibilität im Einsatz. Dieses Konzept wird in einer verteilten Programmiersprache umgesetzt, die als Dialekt von Java implementiert wird. Am Beispiel eines praktischen Einsatzes der im Rahmen der Arbeit konzipierten und neu entwickelten Techniken und Werkzeuge wird deren Tauglichkeit für reale Anwendungsszenarien untersucht und evaluiert.

Kapitel 1

Einleitung

Die Entwicklung von Software ist eine komplexe Aufgabe. Wichtige Fortschritte auf Gebieten wie der Softwaretechnik, dem Sprachdesign und der Hardware haben den Prozess der Entwicklung vereinfacht, doch in gleichem Maße wie die Reduktion von Komplexität voranschreitet, steigen die Anforderungen, die an Software gestellt werden. Die Anwendungsmöglichkeiten wachsen und Einsatzgebiete für Software weiten sich aus. Neue Problemstellungen führen zu neuen Komplexitäten, die wiederum durch neue Lösungen reduziert werden müssen.

Ein Anwendungsgebiet der Softwareentwicklung, das als besonders komplex gilt, stellen verteilte Systeme dar. Die Bedeutung der Entwicklung von Anwendungen für verteilte Systeme wächst. Einerseits weitet sich die Durchdringung von rechnergestützten Systemen in zunehmendem Maße aus, andererseits nimmt die Vernetzung von Rechnern zu, so dass vielfach auch die Anwendung selbst verteilt realisiert wird. In diesen müssen – zusätzlich zu den Problemen nichtverteilter Systeme – unter anderem die Kommunikation über Rechnergrenzen hinweg bewältigt, die Komplexität der natürlich gegebenen Nebenläufigkeit behandelt und die Nachhaltigkeit durch die Sicherung der Daten gewährleistet werden.

Ziel dieser Arbeit ist, die Komplexität in der Programmierung verteilter Softwareanwendungen zu reduzieren. Ansatzpunkte hierfür lassen sich auf Ebene der Betriebssysteme, der Middleware, der Programmiersprachen oder bei der Modellbildung finden. Ansatzpunkt dieser Arbeit ist die Ebene der Programmiersprachen. Es werden zunächst die heutzutage zur Verfügung stehenden Konzepte auf dieser Ebene untersucht, wobei besonders die Aspekte Verteilung, Nebenläufigkeit und Persistenz betrachtet werden. Es wird herausgearbeitet, dass diese drei Aspekte heutzutage mit sehr unterschiedlichen Mechanismen behandelt werden. Die Notwendigkeit, die Details aller drei Aspekte kennen und behandeln zu müssen, führt zu einer enormen Komplexität. Dann werden die Möglichkeiten diskutiert, diese zu reduzieren. Aus der Erkenntnis einer prinzipiellen Ähnlichkeit der drei Aspekte heraus wird die Frage untersucht, ob sich für diese eine Abstraktion finden lässt, die ihre Zusammenfassung zu einem Konzept erlaubt. Ein konkreter Vorschlag wird erarbeitet, in einem Dialekt einer existierenden Sprache realisiert und das Ergebnis anhand von Beispielen evaluiert.

In diesem einleitenden Kapitel soll zunächst der Rahmen der Arbeit gesteckt

werden. Zunächst werden im Abschnitt 1.1 die zugrundeliegenden Begriffe »verteiltes System« und »verteilte Anwendung« eingeführt und deren Relevanz erläutert. In Abschnitt 1.2 erfolgt eine Erläuterung der Begriffe Nebenläufigkeit, Verteilung und Persistenz und ihrer besonderen Bedeutung in verteilten Systemen. In Abschnitt 1.3 wird auf die herausragende Rolle der Programmiersprache Java für dieses Gebiet eingegangen. Schließlich wird in Abschnitt 1.4 der Aufbau dieses Textes dargestellt.

1.1 Die Bedeutung verteilter Systeme

Unter einem verteilten System wird allgemein ein Verbund von mehreren Rechnern mit separatem Speicher, die durch ein Netzwerk miteinander gekoppelt sind und auf denen eine verteilte Anwendung lauffähig ist, verstanden. Als Rechner sind im Zusammenhang dieser Arbeit – ohne Einschränkung der Allgemeingültigkeit – nur Geräte relevant, auf denen zumindest eine Java Virtual Machine existiert und die zur Kommunikation über ein Netzwerk fähig sind. Das Netzwerk soll im allgemeinen fest sein, auch wenn einzelne Geräte (zum Beispiel durch Ausschalten) ausfallen können, und die Geräte sollen jeweils eine feste Identifikation innerhalb des Netzes haben. Eine Umkonfiguration des Systems geschieht kontrolliert und nicht spontan, wie etwa in mobilen Systemen.

Eine verteilte Anwendung ist eine Anwendung, die aus mehreren, miteinander kommunizierenden Programmteilen besteht, die kooperativ an der Erfüllung einer gemeinsamen Aufgabe arbeiten. Dabei sind die Teile der Anwendung typischerweise, aber nicht unbedingt, über mehrere Rechner verteilt. Die Verteilung kann auch auf einem Rechner simuliert sein. Information wird in diesem Fall allerdings nicht über einen gemeinsamen Speicher oder Adressraum, sondern durch Techniken der entfernten Kommunikation übermittelt.

Bis etwa Anfang der achtziger Jahre waren nahezu alle Rechnersysteme zentralisierte Mainframe-Anlagen, auf die mehrere Arbeiter über ein Textterminal zugreifen konnten. Durch das Aufkommen der Personal-Computer (PC) wurden diese zunächst vielfach durch Einplatz-Rechner verdrängt, die zwar dem einzelnen Benutzer einen billigen und leichter zu bedienenden Rechner an die Hand gaben, ihn aber von gemeinsam genutzten Ressourcen, wie etwa zentralen Druckern oder der Kommunikation über den Computer, abschnitten. Die Entwicklung von Netzwerktechnologien zur Vernetzung solcher PCs ermöglichte schließlich den Aufbau lokaler Netze. Auch im Hochleistungsbereich geht die Tendenz von großen zentralen Systemen hin zu vernetzten verteilten Systemen.

Verteilte Systeme haben gegenüber zentralisierten Systemen eine ganze Reihe von Vorteilen, von denen hier einige genannt seien:

- In verteilten Systemen können viele Rechner gleichzeitig gemeinsam an der Lösung eines Problems arbeiten. Voraussetzung hierfür ist, dass sich ein Problem in viele kleine Teilprobleme zerlegen und somit parallelisieren lässt. Natürlich gibt es auch zentralisierte Parallelrechner, die solche Aufgaben ebenfalls erledigen können, doch diese sind extrem teuer und spezialisiert. PCs dagegen stehen heutzutage an fast jedem Büroarbeitsplatz, sind vielfach auch vernetzt und bearbeiten den größten Teil des Ta-

ges gar nichts. Dieses Potential kann durch verteilte Anwendungen genutzt werden. Zum Beispiel können Bildszenen oder kryptographische Probleme elegant in Rechner-Farmen oder sogar in spontan über das Internet verknüpften Rechnerverbünden gelöst werden.

- Verteilte Systeme sind im Verhältnis zu zentralisierten Systemen sehr viel leichter zu skalieren: In zentralen Systemen, in denen große und teure Spezialrechner ihre Arbeit tun, wie es auch heute noch in vielen Rechenzentren von Banken, meteorologischen Instituten oder Industrieunternehmen häufig der Fall ist, kann eine Erhöhung der Rechenleistung bei steigender Anforderung oft nur um einen Faktor 2 gesteigert werden. In verteilten Systemen dagegen können je nach Bedarf ein paar Rechenknoten hinzugefügt oder entfernt werden.

- Teure Ressourcen, wie zum Beispiel Drucker, oder zentralisierte Ressourcen, wie etwa Datenbanken, können durch ein verteiltes System durch viele dezentrale Geräte genutzt werden.

- Verteilte Systeme können eine erhöhte Zuverlässigkeit durch Replikation ermöglichen. Rechner von wichtigen Anlagen, die auf keinen Fall ausfallen dürfen, wie etwa in Flugverkehr-Steuerungssystemen, können durch einen zweiten Rechner, der den Berechnungsstand mitverfolgt, abgesichert werden. Fällt der Hauptrechner aus, springt der Zweitrechner ein und setzt die Berechnungen unmittelbar fort.

- In vielen Anwendungen ist die Verteilung schlicht eine natürliche Gegebenheit der Anwendung. In chemischen Prozess-Steuerungsanlagen oder in Verkehrsleitsystemen etwa fallen die Daten an verschiedenen Messeinrichtungen an und werden zu zentralen Rechnern übertragen, um anschließend wieder zu verteilten Reglern gesendet zu werden, die oft in der Nähe der ursprünglichen Messstationen liegen. Durch eine verteilte *Verarbeitung* dieser Daten kann der Kommunikationsaufwand und die Komplexität solcher Anwendungen reduziert werden.

Verteilte Systeme finden eine wachsende Verbreitung. Bisherige verteilte Systeme sind oft sehr teure und hochspezialisierte Anlagen. Durch technologische Entwicklungen wie etwa Java und den Java-Chip wird es möglich sein, auch sehr einfache und billige Geräte an ein Kommunikationsnetz anzuschließen und ihnen eigenständige Verarbeitungseinheiten und Programme zuzuteilen. So könnten zum Beispiel Lichtschalter und Helligkeitsmesser im Haushalt oder im Büro mit der Heizsteuerung oder der automatischen Beleuchtungsanlage eines Gebäudes kommunizieren. Die Einsatzgebiete verteilter Systeme sind zahlreich. Verteilte Anwendungen werden zum Beispiel auf dem Gebiet des Electronic Commerce, in der Automatisierungstechnik und in der Steuerung von Arbeitsabläufen (Workflow-Systeme) und der kooperativen Zusammenarbeit (computer supported cooperative work, CSCW) entwickelt und eingesetzt.

Ebenso nimmt der Grad der Vernetzung zu, so dass die nötige Infrastruktur immer besser und billiger wird und in immer mehr Bereiche vordringt. Neben

dem Internet entstehen weitere Netze, die mit denselben Techniken arbeiten wie das Internet. Firmeninterne und von der Öffentlichkeit des Internets abgeschirmte Netze, so genannte Intranets, sorgen für eine schnelle, zuverlässige und sichere Kommunikationsinfrastruktur innerhalb von Unternehmen. Auch zwischen den Filialen globaler Organisationen oder zwischen kooperierenden Firmen werden spezielle Netze, die Extranets, eingerichtet. Aber auch in sehr kleinen Netzen finden verstärkt dieselben Techniken Verwendung. In einem modernen Auto zum Beispiel befinden sich heutzutage bis zu 50 einzelner Prozessoren, die zum Teil durch so genannte Controller Area Networks (CAN) verbunden werden und miteinander kommunizieren.

1.2 Nebenläufigkeit, Verteilung, Persistenz

Die Entwicklung und Programmierung verteilter Systeme ist erheblich komplexer als die Programmierung von lokalen Programmen, wie sie typischerweise auf normalen Arbeitsplatzrechnern im Einsatz sind. Natürlich ist schon die Entwicklung sequentieller Programme sehr komplex, doch durch die Verteilung kommen gleich mehrere Dimensionen der Komplexität hinzu. Während lokale sequentielle Programme auf genau einem Rechner ausgeführt werden, von nur einer Person bedient werden und Daten ohne größere Umstände in einer Datei abgelegt werden können, laufen verteilte Anwendungen auf mehreren, räumlich verteilten Rechnern, müssen über ein relativ langsames und potentiell unsicheres Netzwerk kommunizieren, werden von mehreren Benutzern gleichzeitig benutzt, und Zugriff und Sicherung von gemeinsam genutzten Daten muss durch Transaktionen gesichert und deren konsistente zentrale oder verteilte Datenhaltung gewährleistet werden.

Diese Arbeit behandelt insbesondere drei dieser Aspekte: Nebenläufigkeit, Verteilung und Persistenz.

Nebenläufigkeit ist die tatsächliche oder scheinbare Parallelität von Kontrollflüssen. Nebenläufigkeit kann sowohl zwischen verschiedenen Prozessoren, wie zum Beispiel in einem Mehrprozessorsystem, aber auch auf einem Prozessor auftreten. Moderne Prozessorarchitekturen erlauben es, auf einem Prozessor mehrere Prozesse gleichzeitig ablaufen zu lassen. Dabei laufen diese Prozesse natürlich nicht wirklich gleichzeitig, sondern bekommen zueinander versetzt Zeitscheiben der Rechenzeit des Prozessors zugeteilt. Für den Benutzer allerdings scheinen diese Prozesse parallel abzulaufen. Es gibt zwei Arten solcher Prozesse. Einerseits können sie voneinander völlig entkoppelt sein, so dass sie eine jeweils eigenständige Ausführungsumgebung haben. Dann bezeichnet man sie als schwergewichtige Prozesse. Andererseits können innerhalb eines Ausführungskontextes mehrere Kontrollflüsse, so genannte leichtgewichtige Prozesse, geführt werden, die durch einen gemeinsamen Adressraum miteinander kooperieren können. In Java haben die leichtgewichtigen Prozesse, die hier als Threads bezeichnet werden, eine benutzerfreundliche Integration erfahren: Der Programmierer kann sie unmittelbar aus der Sprache erzeugen und verwalten. Andererseits können durch moderne Rechnerarchitekturen und Betriebssysteme

heutzutage auch mehrere Prozessoren in einem Rechnersystem integriert sein, wodurch eine echte Parallelität entsteht. Nebenläufigkeit soll hier als ein Oberbegriff für scheinbare oder simulierte Parallelität, die man durch mehrere Prozesse erhält, und tatsächliche Parallelität durch mehrere Prozessoren verstanden werden.

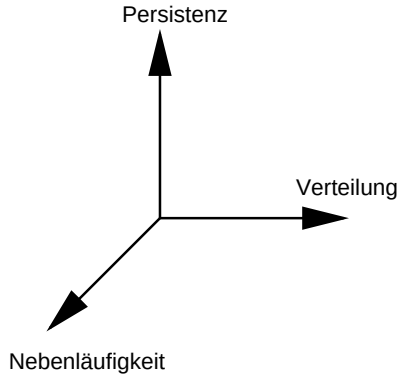
Verteilung ist die logische oder physikalische räumliche Entfernung von Objekten zueinander. Zwei Objekte, die zur gemeinsamen Kommunikation nicht den gewöhnlichen Methodenaufruf verwenden können, sondern Mechanismen der entfernten Kommunikation nutzen müssen, sind zueinander verteilt. Dies ist der Fall, wenn sie sich auf unterschiedlichen Rechnern befinden, also räumlich getrennt sind. Doch auch wenn sie auf demselben Rechner aber in unterschiedlichen Adressräumen (in verschiedenen schwergewichtigen Prozessen) liegen, sind sie (logisch) verteilt. Natürlich sind auch verschiedene Objekte, die nichts miteinander zu tun haben, aber auf unterschiedlichen Rechnern liegen, zueinander verteilt, haben in diesem Kontext aber auch keine große Bedeutung. Hier soll unter Verteilung eine Trennung von miteinander in Beziehung stehenden Objekten gemeint sein. Zwischen diesen besteht das Problem, wie sie sich gegenseitig lokalisieren, aufeinander zugreifen oder miteinander kommunizieren können.

Persistenz ist die nachhaltige Speicherung von Daten oder Objekten auf nichtflüchtigen Medien. Daten oder Objekte, die nicht explizit in irgendeiner Form gespeichert werden, existieren nach dem Ende eines Programms nicht mehr. Sie werden als transient bezeichnet. Aber viele Daten möchte man nicht verlieren, auch wenn man ein Programm beendet oder den Rechner ausschaltet. Daher müssen sie auf die eine oder andere Weise gespeichert werden, um bei einem erneuten Start wieder zur Verfügung stehen zu können. Dann spricht man von persistenten Daten oder Objekten. Durch Persistenz erreicht man quasi die Verteilung von Daten oder Objekten in der Zeit.

Diese drei Aspekte der Programmierung verteilter Anwendungen sind voneinander jeweils relativ unabhängig und werden, wie im ersten Teil der Arbeit gezeigt wird, mit völlig unterschiedlichen Techniken behandelt. Man könnte sie daher als zueinander orthogonal bezeichnen und von Dimensionen sprechen. Dennoch müssen sie, zumindest in vielen Fällen, gleichzeitig bearbeitet werden. Diese drei Dimensionen sind in Abbildung 1.1 dargestellt.

Im ersten Teil dieser Arbeit werden für alle drei Aspekte die Konzepte vorgestellt, die den State-of-the-Art darstellen. Den Schwerpunkt innerhalb dieser Aspekte bildet die Verteilung oder, besser gesagt, die Überwindung der damit verbundenen Probleme. Die Kommunikation von Programmen in verteilten Systemen kann auf sehr unterschiedliche Arten gelöst werden, denen allerdings immer eines gemein ist: Letztendlich werden über ein Netzwerk Bits und Bytes übermittelt. Doch diese einfache Datenkommunikation lässt sich auf verschiedene Ebenen abstrahieren. Auf unterster Ebene steht der Mechanismus zur Übertragung von Datenströmen, im Folgenden als unstrukturierte Datenkommunikation bezeichnet. Hier werden Daten ohne weitere Interpretation von einem Rech-

Abbildung 1.1: Aspekte verteilter Programmierung



ner zu einem anderen übertragen. Auf diesem Mechanismus bauen alle weiteren auf, verleihen den übertragenen Daten aber eine gewisse Semantik und nehmen dem Programmierer einen sehr großen Teil der Arbeit ab, so dass er sich besser auf seine eigentlichen Aufgaben konzentrieren kann. Aus Sicht des Programmierers ist es wünschenswert, nicht von einem Rechner zum anderen kommunizieren zu müssen, sondern von Objekt zu Objekt, wie er es in der objektorientierten Programmierung gewohnt ist. Diese Sicht wird durch die Kommunikation über Stellvertreter angenähert. Eine Manifestation dieses Mechanismus ist die Remote Method Invocation (RMI), die jedoch auf der Annahme basiert, dass beide Kommunikationspartner die gleiche Semantik der übertragenen Daten verwenden, was praktisch bedeutet, dass die gleiche Sprache benutzt werden muss und ferner der Ort des entfernten Objekts bekannt ist. Zur Kommunikation in heterogenen Umgebungen muss von diesen Annahmen weiter abstrahiert werden, so dass Ort und die verwendete Sprache eines entfernten Objekts verdeckt bleiben können, und man gelangt zu heterogenen Middlewareumgebungen, deren prominentester Vertreter *CORBA*, die Common Object Request Broker Architecture ist, die von der OMG standardisiert wird. Hier kann der Programmierer ein entferntes Objekt nutzen, ohne zu wissen, wo es ist und in welcher Sprache es implementiert wurde. Die nächste Stufe der Abstraktion besteht darin, dass man den Ort eines Objekts zur Laufzeit verändern kann, was man als Migration bezeichnet. Ein System, das dies ermöglicht, ist *Voyager*, das noch einige weitere für verteilte Umgebungen nützliche Eigenschaften aufweist. So kommt zum Beispiel der asynchronen Kommunikation in verteilten Umgebungen eine weitaus höhere Bedeutung zu als in nicht verteilten und wird daher, ebenfalls am Beispiel Voyager, untersucht. Aus Sicht der Programmierung wäre aber eigentlich eine völlige Transparenz der Verteilung wünschenswert, was durch eine unmittelbare Objektkommunikation prinzipiell erreichbar ist und in einigen Forschungsprojekten auch umgesetzt worden ist, was in der Praxis allerdings zu Problemen führt, die ebenfalls eingehend diskutiert werden.

Im Gegensatz zur Verteilung, die die Ausführung von Programmen deutlich langsamer macht, kann durch die Nebenläufigkeit eine deutliche Beschleunigung der Ausführungsgeschwindigkeit erreicht werden. Die grundlegendste Form der Nebenläufigkeit wird durch die Nutzung von physikalisch verteilten Prozesso-

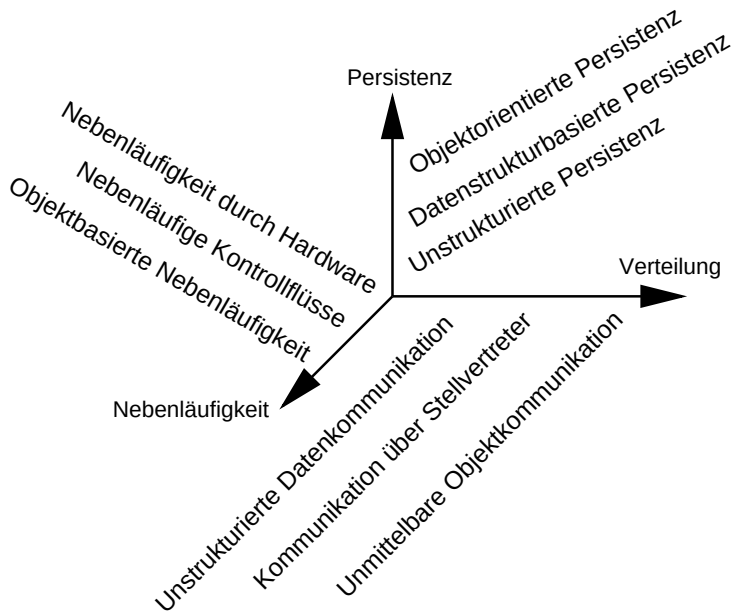
ren erreicht, im Folgenden als Nebenläufigkeit durch Hardware bezeichnet. Um diese allerdings ausnutzten zu können ist meist auch entfernte Kommunikation nötig. Moderne Prozessoren lassen eine Variante hiervon zu, indem auf einem Prozessor die Einrichtung von getrennten Ausführungsräumen, die als Prozesse bezeichnet werden, möglich ist, die nebenläufig und unabhängig voneinander ausgeführt werden können. Für die Nebenläufigkeit innerhalb eines Prozesses wurde das Konzept der leichtgewichtigen Prozesse eingeführt, das in Java durch den Mechanismus der Threads umgesetzt wurde. Grundlegende Probleme und Techniken der Synchronisation sowie Muster des Einsatzes dieser Art der Nebenläufigkeit, wie sie in verteilten Systemen häufig auftreten, werden behandelt. In einigen Forschungsarbeiten wird als Grundlage für die Granularität der Nebenläufigkeit nicht mehr ein Prozess, sondern ein Objekt, oft als aktives Objekt bezeichnet, oder eine Komponente herangezogen, hier als objekt- oder komponentenbasierte Nebenläufigkeit bezeichnet.

Für die als drittes genannte Dimension in der Entwicklung verteilter Anwendungen, die Persistenz, werden ebenfalls drei grundsätzliche Ansätze unterschieden. Der fundamentalste, die einfache Speicherung von serialisierten Datenströmen in Dateien, wird hier als unstrukturierte Persistenz bezeichnet, da sie, ähnlich wie die unstrukturierte Datenkommunikation, aus Sicht des Speichermechanismus Daten unabhängig von ihrer Struktur abspeichert. Die in der Praxis am weitesten verbreitete und bewährteste Technik stellt die datenstrukturbasierte Persistenz dar, bei der die Speicherung der Daten nach deren inhärenter interner Struktur organisiert ist und sich auf Tabellen in relationalen Datenbanken abbilden lässt. Diese ermöglichen die Suche nach Daten durch strukturierte Anfragen mit Anfragesprachen wie SQL. Sie erlauben auch die Koordination von Zugriffen mehrerer Benutzer und die Einhaltung der so genannten ACID-Prinzipien (Atomicity, Consistency, Isolation, Durability). Für Java ist der Zugriff auf relationale Datenbanken durch die Java Database Connectivity (JDBC) standardisiert. Die Speicherung von Daten in Tabellen bedeutet allerdings einen Bruch zu objektorientierten Programmiersprachen, in denen nicht mehr mit Daten, sondern mit gekapselten Objekten umgegangen wird. Um dies zu vermeiden, wurden Mechanismen für eine objektorientierte Persistenz entwickelt, die sich sehr viel nahtloser in Sprachen wie Java einfügen lässt. Schließlich wird untersucht, ob Persistenz als fester Bestandteil in eine Programmierumgebung wie Java integrierbar ist. Wenn dies für alle Datenelemente gelingt, wird dies als orthogonale Persistenz bezeichnet, was derzeit in dem Projekt PJama [Atkinson und Jordan 1998] verwirklicht wird.

Diese drei Aspekte und die jeweils diskutierten Konzepte sind in Abbildung 1.2 gezeigt und bilden das Grundgerüst für die Struktur des ersten Teils der Arbeit. Dabei sind die Konzepte nach Abstraktionsgrad geordnet, wobei fundamentalere Konzepte mit einem geringeren Abstraktiongrad näher am Ursprung liegen und solche mit höherem Abstraktionsgrad weiter davon entfernt. Typischerweiser werden die abstrakteren Konzepte praktisch mit Mitteln des in dieser Darstellung jeweils näher am Ursprung liegenden Konzepts realisiert.

Ein Großteil der Komplexität der Entwicklung verteilter Anwendungen liegt darin begründet, dass in fast allen praktischen Anwendungen alle drei Aspekte behandelt werden müssen, so dass die Konzepte aller drei Aspekte gut verstan-

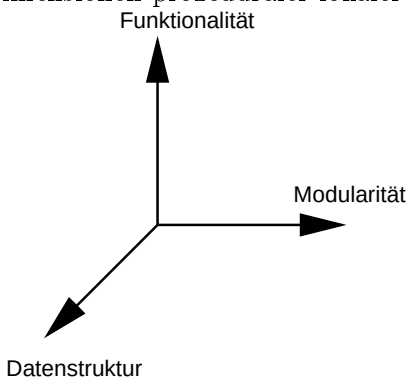
Abbildung 1.2: Die im ersten Teil betrachteten Konzepte



den und beherrscht werden müssen.

Ziel dieser Arbeit ist, die Komplexität dadurch zu vereinfachen, alle drei Aspekte in einem Konzept zusammenzufassen. Um dies zu verdeutlichen, soll hier eine Analogie aus der Entstehung der objektorientierten Programmiersprachen angeführt werden. In der prozeduralen Programmierung, etwa mit Sprachen wie Pascal oder C in ihren ursprünglichen Formen, in denen zunächst rein sequentielle und lokale Programme entwickelt wurden, lassen sich ebenfalls drei Dimensionen erkennen, die es genauso mit unterschiedlichen Techniken zu bewältigen galt. Diese sind

Abbildung 1.3: Dimensionen prozeduraler lokaler Programmierung



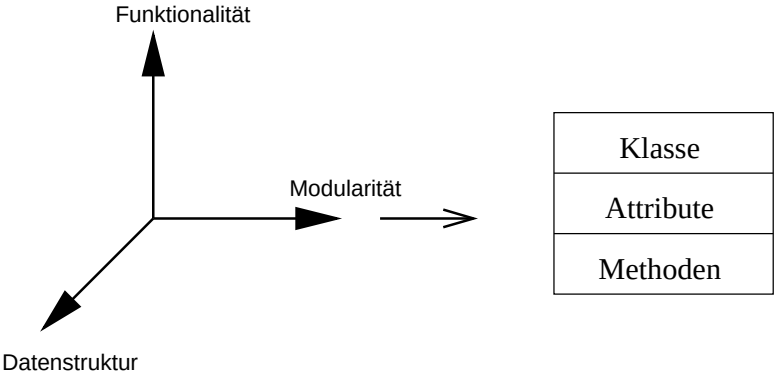
Datenstruktur, die durch Basistypen, Arrays, Structs und Records aufgebaut wird,

Funktionalität, die in Prozeduren festgelegt wird, und

Modularität, die in Dateien, Programmpaketen und Bibliotheken ausgedrückt wird.

Diese drei Dimensionen wurden durch die Objektorientierung zu einem einzigen Konzept zusammengefasst, dem Konzept der Klasse. Eine Klasse (genau genommen deren Instanz, das Objekt) birgt in sich eine Datenstruktur, die meist als Attribute einer Klasse bezeichnet werden, und bindet sie untrennbar mit ihrer Funktionalität, manifestiert in Methoden einer Klasse, zusammen, was auch als Kapselung bezeichnet wird. Gleichzeitig stellt sie einen Mechanismus zur Modularisierung dar, da eine Klasse eine als solche wiederverwendbare Einheit darstellt. Dies wird in Abbildung 1.4 dargestellt.

Abbildung 1.4: Zusammenfassung von Konzepten beim Wechsel von prozeduraler zu objektorienterter Programmierung

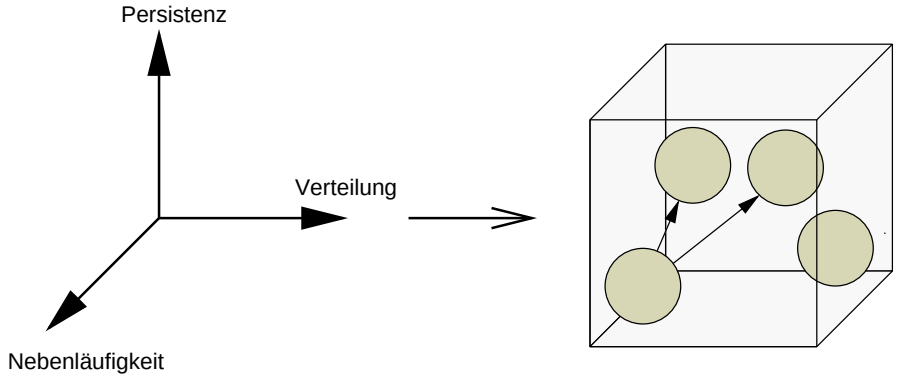


a)Prozedurale Programmierung b)objektorientierte Programmierung

Diese Vereinheitlichung von drei zuvor getrennten Aspekten zu einem Konzept vereinfachte die Programmierung beträchtlich und setzte Kapazitäten für die Bearbeitung neuer Aspekte frei, etwa die Nebenläufigkeit, die Verteilung und die Persistenz. Nicht dass dies nicht auch mit prozeduralen Sprachen möglich wäre, doch objektorientierte Sprachen haben sich für derart komplexe Gebiete als produktiver und wartbarer erwiesen. Wenn es nun gelingt, auch diese drei – Nebenläufigkeit, Verteilung und Persistenz – in einem Konzept zu vereinen, könnte dies eine weitere beträchtliche Vereinfachung der Programmierung bedeuten, insbesondere für verteilte Anwendungen, die als ausgesprochen schwierig zu programmieren gelten. Ein Konzept, das dies ermöglicht, wird im zweiten Teil dieser Arbeit entwickelt.

Dazu werden zunächst die im ersten Teil betrachteten Konzepte diskutiert und insbesondere die Konzepte mit hohem Abstraktionsniveau kritisch betrachtet. Daraus werden die Grenzen des Machbaren und die offen bleibenden Möglichkeiten hergeleitet und ein geeignetes Konzept entwickelt. Grundlage bildet ein neu entwickeltes Konzept, das als virtueller Prozessor bezeichnet wird. Ein virtueller Prozessor dient als eine Art Objektkontainer, der für die in ihm gekapselten Objekte eine gemeinsame Funktionalität bietet und unter den Aspekten Verteilung, Nebenläufigkeit und Persistenz für die Konsistenz aller Beziehungen der Objekte untereinander sorgt. Dadurch wird eine Vereinigung dieser Aspekte zu einem Konzept erreicht, was schematisch in Abbildung 1.5 dargestellt wird.

Abbildung 1.5: Zusammenfassung Verteilung, Nebenläufigkeit und Persistenz mittels eines neuen Konzeptes



a) Aspekte nach Stand der Technik b) virtueller Prozessor

Eine konkrete Umsetzung dieses Konzeptes in eine Programmiersprache wird vorgestellt, für die als Grundlage die Sprache Java dient, was im folgenden Abschnitt motiviert werden soll.

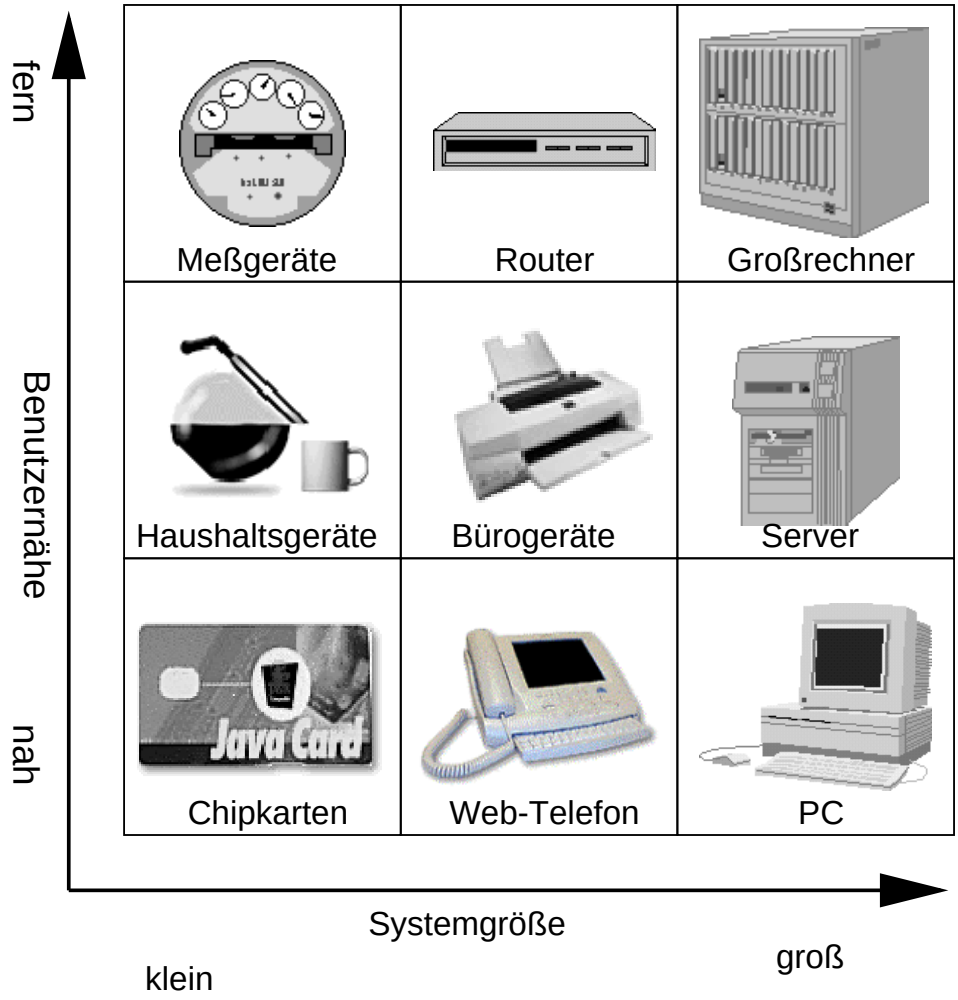
1.3 Die Bedeutung von Java

Das im zweiten Teil entwickelte Konzept soll, um den Nachweis der Realisierbarkeit zu führen, als Erweiterung einer konkreten, akzeptierten und weit verbreiteten Sprache umgesetzt werden. Hierfür wurde Java gewählt. Um dies zu begründen, erfolgt hier eine kurze Betrachtung der Bedeutung von Java im Kontext verteilter Systeme.

Java setzt sich als universelle Programmiersprache in allen Einsatzbereichen mehr und mehr durch. Die Bezeichnung »plattformunabhängig« bekommt durch Java eine neue Bedeutung. Oft wird darunter verstanden, dass ein Programm unter Unix entwickelt und auf Windows- und MacOS-Rechnern gestartet werden kann – sicherlich eine wichtige Eigenschaft von Java. Doch inzwischen wird Java nicht mehr nur auf PCs eingesetzt, sondern deckt ein Spektrum von Kleinstgeräten bis hin zu Hochleistungsrechnern ab. Dieses Spektrum, das von sehr kleinen Systemgrößen wie Chipkarten bis hin zu Massiv-Parallelrechnern und von sehr benutzernahen Systemen wie PCs, Telefonen oder PDAs bis zu benutzerfernen Systemen wie Reglern oder Routern reicht, wird in Abbildung 1.6 dargestellt.

Ursprünglich war Java für den Einsatz in Kleingeräten wie Set-Top-Boxen, Mess- und Steuerungsanlagen und Routern entwickelt worden. Sun Microsystems startete 1990 ein Projekt mit dem Namen »Green Project« unter der Führung von James Gosling. Das Projekt beschäftigte sich zunächst mit Consumer Electronics, Interactive TV und Set-Top-Boxen, für die die Programmiersprache OAK entwickelt wurde, und die das Problem der Heterogenität dieser Geräte lösen und aus einem Netz heraus auf diese Geräte geladen werden sollte. Aus diesem Projekt ging schließlich die Sprache Java hervor.

Abbildung 1.6: Verschiedene rechnergesteuerte Systeme, in denen Java Einsatz findet



Den Durchbruch erfuhr Java auf dem Gebiet der PCs, also nach der Darstellung in Abbildung 1.6 auf dem diametral gegenüberliegenden Gebiet. Die eigentlich für Kleinstgeräte entwickelte Plattformunabhängigkeit machte Java zu *der* Sprache im Internet. Denn hier wurden PCs mit unterschiedlichen Architekturen und kaum überwindbaren Systemunterschieden eingesetzt, für die eine gemeinsame Programmiersprache fehlte. Durch das Konzept der virtuellen Maschine (Java Virtual Machine), die für jede Plattform einmal entwickelt werden muss, anschließend aber ermöglicht, dass Anwendungsprogramme nur noch einmal für eben diese entwickelt werden müssen, werden die tiefen Schluchten zwischen den verschiedenen Hardwaresystemen überwunden. Durch die Integration dieser virtuellen Maschine in Internetbrowser können Programme über das Internet geladen und innerhalb einer Webseite gestartet werden. Damit konnten zunächst grafische Effekte erzielt werden, doch die eigentliche Bedeutung liegt darin, dass damit eine Möglichkeit zur Programmierung von Clients zur Verfügung stand.

Nachdem Java die Clientseite im Sturmlauf genommen hatte, beginnt es nun langsam auch die Serverseite zu erobern. Hier zählt der Vorteil der Plattformunabhängigkeit nicht mehr so sehr, da serverseitige Programme eher für eine dedizierte Plattform entwickelt oder portiert werden können. Dagegen wiegt hier der Nachteil der Langsamkeit schwer. Doch durch Just-In-Time-Compiler oder plattformspezifische Compiler kann Java diesen Nachteil langsam wett machen. In Form von Servlets, die von einem HTTP-Server geladen und aufgerufen werden können, ehe ein entsprechendes Tag in einer HTML-Seite gesendet wird, kann Java seine volle Stärke sogar wieder zur Geltung bringen: Servlets können, wie Applets, mittels URL über das Internet geladen und dynamisch eingebunden werden, nur eben auf Serverseite, und können so zum Beispiel Webinformationen dynamisch erzeugen.

Doch Java gibt es sogar auf so kleinen Geräten wie Chipkarten. Auf einer Chipkarte kann eine abgespeckte Version der Java Virtual Machine laufen und dann herstellerunabhängig mit Java programmiert werden. Die Einsatzmöglichkeiten des JavaCard-Konzepts sind heute noch durch die langsame Ausführungszeit von Java-Bytecode beschränkt. So lassen sich zum Beispiel kryptographische Algorithmen noch nicht effizient auf der JavaCard programmieren. Doch in absehbarer Zukunft werden dieser Technologie große Möglichkeiten eingeräumt.

Für etwas größere Geräte, die einen vollwertigen Prozessor beherbergen können, aber dennoch nicht die Notwendigkeit des vollen Funktionsumfanges von Java benötigen, werden Embedded- und PersonalJava entwickelt. Durch den Java-Chip können künftig auch kleine Geräte wie Haushaltsgeräte oder Büromaschinen, die bisher keine oder hochspezialisierte Hardware enthielten, mit Java programmiert werden. Einsatzgebiete für diese Techniken sind Web-Telefone, Büro- und Haushaltsgeräte sowie mobile Geräte.

Inzwischen gibt es sogar Projekte, um Großrechner in die Java-Welt einzubinden oder Java selbst auf Großrechnern laufen zu lassen. Obwohl Java heutzutage noch als langsam in der Ausführung gilt und zunächst anscheinend nichts mit »High Performance Computing« zu tun hat, bietet es dennoch ein sehr großes Potential für das Hochleistungsrechnen. Das Java Grande Forum, ein Konsortium, das die Weiterentwicklung von Java zu einer Hochleistungssprache vorantreibt, hält Java sogar für die potentiell beste bisher existierende Sprache für dieses Gebiet, auch wenn es noch einige Probleme zu überwinden gilt.

Java wird also auf einem sehr großen Spektrum von Hardwareplattformen eingesetzt und findet auf allen Gebieten eine immer größere Verbreitung und Akzeptanz. Doch in dieser Arbeit geht es nicht nur um die Programmierung von einzelnen Systemen, die abgesondert auf einer der vielen möglichen Plattformen laufen, sondern um das Zusammenarbeiten von mehreren Programmteilen über ein Netzwerk: Es geht um die Programmierung von Anwendungen in verteilten Systemen.

1.4 Aufbau der Arbeit

Der vorliegende Text gliedert sich in drei Hauptteile. Im ersten Teil werden die heutzutage zur Bewältigung der Aspekte Verteilung, Nebenläufigkeit und Persi-

stenz eingesetzten Konzepte vorgestellt. Im zweiten Teil werden die Möglichkeiten der Reduktion der Komplexität dieser Aspekte diskutiert und ein konkreter Vorschlag erarbeitet. Im dritten Teil schließlich wird das entwickelte Konzept evaluiert. Den Abschluss bilden eine Zusammenfassung und ein Ausblick.

Im ersten Teil dieser Arbeit soll zunächst erarbeitet werden, welche Konzepte in der Programmierung verteilter Systeme den heutigen Stand der Technik darstellen. Besonderes Augenmerk wird dabei auf Aspekte gelegt, die in verteilten Systemen zusätzlich beachtet werden müssen und damit die eigentliche Herausforderung solcher Systeme darstellen. Dies sind vor allem die Überwindung der physikalischen Verteilung durch entfernte Kommunikation (Kapitel 2), die Behandlung der natürlich gegebenen oder zusätzlich erforderlichen Nebenläufigkeit (Kapitel 3) und die Sicherstellung der Nachhaltigkeit durch Persistenz (Kapitel 4). Als Ergebnis wird herausgearbeitet, dass die Behandlung von Verteilung, Nebenläufigkeit und Persistenz mit jeweils sehr unterschiedlichen, zueinander orthogonalen Techniken behandelt werden. Die Notwendigkeit, die Details aller drei Aspekte bei der Entwicklung gleichzeitig beherrschen zu müssen, macht die Entwicklung verteilter Anwendungen sehr komplex.

Im zweiten Teil wird untersucht, ob sich die Komplexität, die aus der getrennten Betrachtung von Verteilung, Nebenläufigkeit und Persistenz resultiert, reduzieren lässt. Für jeden einzelnen dieser Aspekte werden im ersten Teil Konzepte diskutiert, die eine weitgehende Vereinfachung bis hin zu einer fast vollständigen Transparenz ermöglichen. Für den Aspekt der Verteilung wird die Transparenz durch das Konzept der unmittelbaren Objektkommunikation erreicht. Doch obgleich eine völlige Transparenz bezüglich der Verteilung technisch möglich ist, ist ein wichtiges Ergebnis der Forschung, dass für praktische Anwendungsfälle bei lose gekoppelten Systemen eine explizite Behandlung der Verteilung sinnvoll ist. Dies wird ausführlich in Kapitel 5 dargestellt. Für Nebenläufigkeit und Persistenz gelingt zwar keine vollständige Transparenz, aber es stehen immerhin Mechanismen zur Verfügung, die eine gute Integration in das Paradigma der Objektorientierung erlauben, was anhand von komponentenbasierter Nebenläufigkeit und objektorientierter bzw. orthogonaler Persistenz gezeigt wird.

Daraus wird geschlossen, dass der explizite Umgang der mit den einzelnen Aspekten verbundenen Problematiken notwendig ist und die Möglichkeiten der weiteren Vereinfachung für jeden einzelnen Aspekt begrenzt sind.

Der Ansatz dieser Arbeit liegt daher darin, eine geeignete Abstraktion zu finden, die alle drei Aspekte in einem Konzept vereinigt. Dadurch kann eine weitergehende Vereinfachung der Programmierung erreicht werden. In Kapitel 6 wird ein geeignetes Konzept entwickelt, so dass die Komplexität auf das Beherrschen eines Mechanismus reduziert wird und das darüber hinaus eine Vereinfachung für jeden einzelnen Aspekt gegenüber den im ersten Teil vorgestellten Techniken darstellt. Die Umsetzbarkeit wird durch eine praktische Realisierung als Dialekt von Java nachgewiesen und in Kapitel 7 vorgestellt. Sich daraus ergebende verbesserte Möglichkeiten der Modellierung und Visualisierung verteilter Systeme werden in Kapitel 8 vorgestellt. Aspekte der technischen Umsetzbarkeit werden in Kapitel 9 diskutiert. Ein Vergleich zu anderen Forschungsarbeiten auf naheliegenden Themengebieten wird in Kapitel 10 angestellt.

Im dritten Teil werden die im zweiten Teil entwickelten Konzepte anhand praktischer Beispiele evaluiert. In Kapitel 11 werden zunächst die Aspekte Verteilung, Nebenläufigkeit und Persistenz einzeln betrachtet und mit den im ersten Teil vorgestellten, heute üblichen Mechanismen, verglichen. In Kapitel 12 wird dann in einem umfassenden Beispiel die Anwendbarkeit der entwickelten Konzepte in der Praxis anhand einer Technologiestudie im Kontext einer industriellen Anwendung überprüft. Schließlich bietet das letzte Kapitel eine Zusammenfassung und einen Ausblick.

Teil I

Konzepte zur Programmierung verteilter Systeme

Kapitel 2

Verteilung

Das Wesen aller Softwareanwendungen ist die Verarbeitung von Daten. In verteilten wie nicht-verteilten Anwendungen müssen diese meist zwischen mehreren Anwendungen oder zumindest mehreren Teilen einer Anwendung ausgetauscht werden. In nicht-verteilten oder eng gekoppelten Systemen stehen dem Entwickler die Möglichkeiten der unmittelbaren Kommunikation durch Zugriff auf gemeinsame Daten oder durch Prozeduraufrufe zur Verfügung. Doch verteilte Systeme sind lose gekoppelt und haben keinen gemeinsamen Speicher, so dass die Übermittlung von Daten durch entfernte Kommunikation geleistet werden muss. Diese ist, in Relation zu eng gekoppelten Systemen oder prozessorinterner Kommunikation, um ein vielfaches teurer und mit erheblichen Sicherheitsrisiken behaftet. Daher können hier nicht die selben Konzepte eingesetzt werden.

In diesem Kapitel werden die Konzepte, die zur entfernten Kommunikation Einsatz finden und den State-of-the-Art auf diesem Gebiet darstellen, vorgestellt. Dabei werden die zur Verfügung stehenden Mechanismen in drei Kategorien eingeteilt, die sich in ihrem Abstraktionsgrad unterscheiden. Das fundamentalste Konzept stellt die einfache Übermittlung von – aus Sicht des Mechanismus – unstrukturierten Daten dar. Es wird im Abschnitt 2.1 am Beispiel der Java-Sockets erläutert. Dieses Konzept findet breite Verwendung und liegt allen weiteren zugrunde. Doch aus Sicht der Programmierung ist eine Annäherung der Mechanismen an die der lokalen Kommunikation erstrebenswert. Der heute übliche Schritt dorthin wird durch das Konzept des Stellvertreters oder Proxies realisiert, der auf Seite des Aufrufers das aufgerufene Programm repräsentiert und die darunterliegende Kommunikation weitgehend verdeckt. Dieses Konzept wird in Abschnitt 2.2 anhand des Java RMI, CORBA und Voyager untersucht. Eine völlige Angleichung an die lokale Kommunikation kann damit allerdings nicht erreicht werden. Konzepte, durch die eine Transparenz der Verteilung erreicht wird, werden in Abschnitt 2.3 vorgestellt und diskutiert.

2.1 Unstrukturierte Datenkommunikation

Die Grundvoraussetzung zur Programmierung in verteilten Systemen ist die Möglichkeit, Daten von einem Teil eines Systems zu einem anderen übertragen zu können. Aus Sicht des Programmierers stehen diese Daten als Bits und Bytes,

als ASCII-Zeichen oder sogar als Objekte zur Verfügung und sollen von einem Computer zu einem anderen übermittelt werden. Zur Übertragung stehen physikalische Träger wie Kupferkabel oder Glasfaserleitungen zur Verfügung, über die elektrische Signale oder Lichtwellen ausgesendet und empfangen werden. Die Kluft zwischen der Darstellung von Daten auf Programmierebene und der physikalischen Ebene wird von Protokollen geschlossen. Das Internet ist ein Netz von Netzen, in dem verschiedene Arten von physikalischen Signalträgern, Netzwerktopologien und -protokollen eingesetzt werden. Die Protokollfamilie, die im Internet Verwendung findet und die verschiedenen Protokolle der Subnetze, aus denen es besteht, vereint, ist TCP/IP.

2.1.1 Punkt-zu-Punkt-Kommunikation

Eine Verbindung über ein TCP/IP-Netzwerk wird durch zwei Sockets repräsentiert, die jeweils durch eine IP-Adresse und eine Portnummer identifiziert sind [Stevens und Wright 1994]. Sockets stellen damit eine der grundlegendsten Techniken zur Kommunikation in verteilten Systemen dar. Alle im weiteren behandelten Verfahren zur Kommunikation setzen intern auf diesem Mechanismus auf, verdecken diesen aber meist vollständig.

Ports

Rechner, die an ein TCP/IP-Netz angeschlossen sind, werden durch ihre IP-Adresse identifiziert. Diese Adressierung ist jedoch zu grob, wenn es darum geht, Prozesse anzusprechen, die über das Netz miteinander kommunizieren. Um nun einen solchen Netzdienst auf einem bestimmten Rechner ansprechen zu können, wird zusätzlich zur IP-Nummer noch eine Portnummer benötigt. Soll von einem Rechner aus eine Verbindung zu einem Dienst auf einem anderen Computer geöffnet werden, beispielsweise zum Datentransfer via FTP, bekommt der lokale Prozess eine Portnummer zugewiesen. Für den entfernten Dienst ist er dann über diese Nummer in Verbindung mit der lokalen IP-Adresse im gesamten Netz eindeutig zu identifizieren.

Wird hingegen auf einem Rechner ein Serverprozess wie etwa ein Webserver gestartet, so bindet ihn das System an einen festgelegten Port. Dann wartet dieser Server als Hintergrundprozess auf eingehende Verbindungen, die an diese Portnummer adressiert sind.

Sockets

Ein Socket ist ein Endpunkt einer Kommunikationsverbindung zweier Rechner. Er wird durch die IP-Adresse und die Portnummer identifiziert. Aus Sicht des Programmierers stellt ein Socket den Mechanismus dar, um Daten von einem Rechner zu einem anderen zu übertragen. Sockets wurden ursprünglich für das BSD-Unix entwickelt, sind heutzutage aber auf allen Plattformen vorhanden und stellen, nach heutigem Programmierstandard, den fundamentalsten Kommunikationsmechanismus dar.

Sockets waren ursprünglich Teil des Betriebssystems und über betriebssystemspezifische Bibliotheken für C oder C++ anzusprechen, so dass der Umgang

mit ihnen oft kompliziert und nicht plattformunabhängig war. In Java hingegen wird dieses Konzept im Kern der Sprache und damit plattformunabhängig angeboten, weshalb dieses Konzept hier am Beispiel der Java-Sockets erläutert werden soll. Eine ausführliche Beschreibung findet sich in [Gosling et al. 1996] und [Neimeyer 1998]. Die Java-Programmierschnittstelle für Sockets abstrahiert vollständig vom darunterliegenden Betriebssystem und vereinfacht deutlich deren Benutzung. Die Klasse Socket ist im Paket java.net enthalten und lässt sich einfach über einen Aufruf des Konstruktors instantiieren. Als Parameter sind die IP-Adresse und die Portnummer des Sockets auf der Gegenseite anzugeben. Auf der Gegenseite muss allerdings schon ein Socket eingerichtet sein, der auf die Verbindungsanforderung reagieren kann. Für eine Reihe von wichtigen Diensten stellen Server solche Sockets auf standardisierten Portnummern zur Verfügung. Um einen eigenen Socket auf einem Server einzurichten, benötigt man einen ServerSocket, der ebenfalls in java.net zur Verfügung steht.

```
int port = 1234;
ServerSocket server = new ServerSocket (port);
```

Wenn eine Verbindung zu diesem Server aufgebaut wird, muss der Server darauf reagieren, indem er diesen Verbindungsaufbau akzeptiert (accept()) und einen neuen Socket speziell für diese Verbindung zur Verfügung stellt. Vielfach wird der Server in einer Endlosschleife auf weitere Verbindungen warten. Die Methode accept() blockiert, bis ein neuer Verbindungswunsch eintrifft.

```
//Server
int port = 1234;
ServerSocket server = new ServerSocket (port);
while (true) {
  System.out.println("Waiting for client...");
  Socket client = server.accept();
  System.out.println("Client "+client.getInetAddress()+" connected.");
}
```

Nachdem ein ServerSocket auf dem Serverrechner eingerichtet ist, kann sich ein Client an diesen wenden und eine Verbindung zu ihm aufbauen.

```
// Client
Socket server = new Socket("sun",1234);
System.out.println("Connected to "+server.getInetAddress());
```

Um nun auch Daten über diese Sockets austauschen zu können, werden in Java Streams verwendet, die im nächsten Abschnitt erläutert werden.

Streams

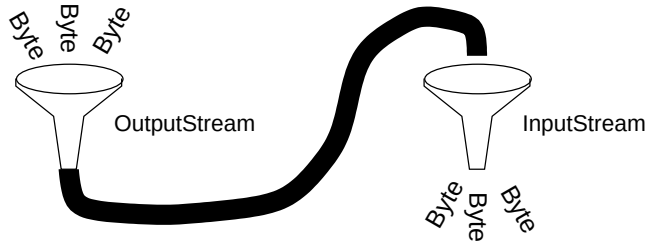
Streams sind eine Abstraktion für beliebige Datenströme, ob von oder zu einem Socket, der Konsole, dem Filesystem oder einem Speichermedium. Streams kommen in Java daher eine sehr wichtige Rolle zu. Es gibt Streams für die Eingabe,

die InputStreams, und für die Ausgabe, die OutputStreams. Von diesen beiden Klassen, die im Package java.io enthalten sind, werden alle weiteren Streams abgeleitet. InputStreams und OutputStreams lassen sich nicht direkt instantiieren (sie sind abstrakte Klassen), sondern immer nur über Erben von diesen, die dann einen Datenstrom in oder aus etwas Konkretem beschreiben, etwa für das Filesystem oder die Konsole.

Die Programmstücke aus dem letzten Abschnitt werden nun erweitert, um tatsächlich ein paar Byte Information zu übertragen. Dafür lässt der Client sich einen OutputStream vom Socket übergeben und schickt ein byte-Array in diesen Stream. Der Server seinerseits verlangt nach einem InputStream, liest diesen in ein byte-Array und gibt den Inhalt aus.

Als einfache Anwendung für Sockets und Streams soll nun ein Programm gezeigt werden, das die aktuelle Zeit auf Anfrage übermittelt. Es erzeugt einen ServerSocket mit der festen Portnummer 1234 und wartet dann durch die Methode accept() auf eingehende Anfragen. Wenn eine eintrifft, wird ein Socket und damit eine Verbindung zum Client erzeugt (von dem zur Kontrolle die IP-Adresse ausgegeben wird). Dann wird die aktuelle Zeit abgefragt, in einen String und dann in ein byte-Array umgewandelt und durch einen OutputStream über die Socketverbindung übertragen.

Abbildung 2.1: Streams




```
// Time Server
import java.net.*;
import java.io.*;
import java.util.*;

public class TimeServer {
  public static void main (String args[]) throws IOException {
    int port = 1234;
    ServerSocket server = new ServerSocket(port);
    while (true) {
      System.out.println("Waiting for client...");
      Socket client = server.accept();
      System.out.println("Client from "+client.getInetAddress()+" connected.");
      OutputStream out = client.getOutputStream();
      Date date = new Date();
      byte b[] = date.toString().getBytes();
      out.write(b);
```

```
    }
  }
}
```

Ein hierzu passender Client könnte wie folgt aussehen: Der TimeClient verbindet sich mit dem TimeServer, indem er einen Socket mit dessen IP-Adresse und Portnummer erzeugt. Aus einem InputStream liest er einen byte-Array, wandelt ihn in einen String um und gibt ihn aus.

```
// TimeClient
import java.net.*;
import java.io.*;

public class TimeClient {

  public static void main (String args[]) throws IOException {
    Socket server = new Socket("sun",1234);
    System.out.println("Connected to "+server.getInetAddress());
    InputStream in = server.getInputStream();

    byte b[] = new byte[100];
    int num = in.read(b);
    String date = new String(b);
    System.out.println("Server said: "+date);
  }
}
```

Die obigen Programme erzeugen die Ausgabe:

```
sun> java TimeServer
sun| Waiting for client...

lin> java TimeClient
lin| Connected to 134.100.11.1

sun| Client from 134.100.11.2 connected.
sun| Waiting for client...

lin| Server said: Fri Jul 31 13:49:05 GMT+03:30 1998
```

Mit Streams steht also ein einfacher Mechanismus zur Übertragung von Daten zur Verfügung. Allerdings lassen sich über Streams nur bytes und byte-Arrays transportieren. Um ihre Funktionalität zu erweitern und zum Beispiel Strings übertragen zu können, ohne sie selbst in einen byte-Array umsetzen zu müssen, bietet Java das Konzept der Filter, auf die hier nicht weiter eingegangen wird.

2.1.2 Multicast

Dem Multicast wird eine wachsende Bedeutung in der Programmierung verteilter Anwendungen zugemessen, auch und gerade in den erst entstehenden Intranets [Wittmann und Zitterbart 1999]. Doch in der Form, wie er im letzten Kapitel vorgestellt wurde, erfüllt er insbesondere die Anforderungen der Zuverlässigkeit und Benutzbarkeit nicht. Daher ist es nicht verwunderlich, dass in Forschungs- und Entwicklungsprojekten nach Erweiterungen und Verbesserungen des Multicasts gesucht wird.

Eine Erweiterung, die auf dem IP-Multicast beruht, ihn aber um fehlende Qualitäten erweitert, ist iBus [Maffeis 1997][Softwired 1998], ein Softwarepaket von der Firma Softwired, entwickelt von Silvano Maffeis. Seit Mai 1999 ist iBus in der Version 2.0 erhältlich und ist für kommerzielle Projekte kostenpflichtig. Für Lehre und Forschung besteht allerdings die Möglichkeit, eine kostenlose Lizenz zu bekommen. Die hier verwendete API basiert noch auf der letzten völlig freien Version (Version 0.5), die sich vom Prinzip her gering von der Version 2.0 unterscheidet. iBus bietet eine ganze Reihe von zusätzlichen Funktionalitäten und versteht sich eher als Kommunikationsmiddleware, basierend auf IP, denn als bloße Erweiterung des Multicasts. Es bietet ein konfigurierbares Protokoll, bei dem der Programmierer abwägen kann, ob ihm die Qualität des Übertragungsdienstes, die Effizienz oder die Sicherheit wichtiger ist. Es lassen sich aus Protokollmodulen unterschiedliche, für den Bedarf anpassbare Protokollstacks aufbauen, die von einer einfachen, unzuverlässigen Multicast-Übertragung, wie im letzten Kapitel behandelt, bis zu zuverlässigen, komprimierten und verschlüsselten Übertragungen reichen. Außerdem lassen sich nicht nur Byteströme, sondern auch beliebige (serialisierbare) Java-Objekte übertragen. Die Socket-Schnittstelle von Java, wie sie in den vorherigen Abschnitten besprochen wurde, wird dabei vollständig überdeckt, die Übertragung beruht allerdings in sehr direkter Weise auf eben dieser. Auch dient iBus, wie Sockets, zur Übertragung von Daten und nicht, wie viele der folgenden Techniken, zum entfernten Methodenaufruf.

In iBus werden alle zu übertragenden Daten in ein Posting gekapselt, einer Klasse, die einen dynamisch erweiterbaren Container von serialisierbaren Java-Objekten darstellt. Ein Posting kann erzeugt, mit beliebigen Java-Objekten gefüllt und dann über das Netz verschickt werden.

```
import iBus.*;

Posting posting = new Posting();
posting.setLength(3);
posting.setObject(0, new Integer(1));
posting.setObject(1, new String("Hi, I am an iBus Message!"));
posting.setObject(2, new Date());
```

Um solch ein Posting abzuschicken, benötigt man einerseits einen Protokollstack und andererseits eine spezielle iBus-URL, an die es geschickt werden kann. Ein Stack lässt sich aus verschiedenen vorhandenen Protokollobjekten zusammenstellen und sich sogar um eigene erweitern. Es gibt eine Reihe solcher Protokoll-

objekte. Das einfachste von diesen ist das IPMCAST, das die Funktionalität von java.net.MulticastSockets bereitstellt. Erweiterungen werden später behandelt, hier soll zunächst die Erzeugung eines solchen einfachen Stacks gezeigt werden:

```
Stack stack = new iBus.Stack("IPMCAST");
```

Während bei MulticastSockets Nachrichten einfach an eine Multicast-IP-Adresse geschickt werden, gibt es bei iBus einen feineren Mechanismus; es werden Kanäle erzeugt, die durch eine URL (*universal resource locator*) gekennzeichnet sind und sich aus einer IP-Adresse, einem Port und einem Themenbezeichner zusammensetzen. Sie haben die Form ibus://<ip address>[:port]/<subject> und werden folgendermaßen erzeugt:

```
iBusURL url = new iBusURL("ibus://226.1.3.5/Talk");
```

Die IP-Adresse kann dabei eine Multicast-Adresse sein, muss es aber nicht, so dass sich sowohl eine Multicast- als auch eine Punkt-zu-Punkt-Kommunikation durch iBus realisieren lässt. Bei solch einem Kanal kann sich eine Anwendung anmelden und Nachrichten empfangen oder senden. Sendende und empfangende Parteien werden von iBus intern unterschieden, so dass es die Methoden registerTalker() und registerListener() gibt. Wenn sowohl der Stack als auch die URL vorhanden sind, kann die Anmeldung erfolgen und das Posting abgeschickt werden. Da in iBus zwischen einer unidirektionalen und einer bidirektionalen Kommunikation unterschieden wird, die auch als Push- und Pull-Technologie bezeichnet werden, wird für das Abschicken eines Postings die Methode push() verwendet. Über push werden Informationen ohne Aufforderung vom Sender versandt, wohingegen über pull() die Möglichkeit besteht, Informationen explizit nachzufragen.

```
stack.registerTalker(url);
stack.push(url, posting);
```

Interessierte Empfänger können sich mittels einer subscribe()-Methode für einen Kanal anmelden. Dafür muss dem Empfänger die URL des Kanals bekannt sein, und er muss eine Klasse zur Verfügung stellen, die ein Interface Receiver implementiert. Dieser Receiver stellt je eine Methode zur Behandlung von Pull- und Push-Nachrichten zur Verfügung (dispatchPull() und dispatchPush()) sowie eine Methode error() zur Fehlerbehandlung. Ein Receiver kann für mehrere Kanäle angemeldet werden, ebenso wie mehrere Receiver für einen Kanal angemeldet werden können. Der Receiver muss die Datenstruktur eines Postings, das er erwartet, kennen, um es zerlegen und geeignet reagieren zu können. Für die oben erzeugten Postings sieht der Receiver etwa so aus:

```
class MyReceiver implements Receiver {

    public void dispatchPush (iBusURL channel, Posting posting) {
        int seq = ((Integer)posting.getObject(0)).intValue();
        String message = (String)posting.getObject(1);
```

```
        Date date = (Date)posting.getObject(2);
        System.out.println("*** Received Message: "+message+" (Seq: "+seq+"
            at "+date.toString()+")");
    }

    public Posting dispatchPull (iBusURL channel, Posting request) {
        //Pull-Nachrichten werden in dieser Applikation nicht erwartet
        return new Posting();
    }

    public void error (iBusURL channel, String details) {
        System.out.println(details);
    }
}
```

Für die Anmeldung an einem Kanal sind also ein Stack, der bei Sender und Empfänger denselben Protokollstack realisiert, eine URL des Kanals und eine Implementierung eines Receivers notwendig.

```
Stack stack = new iBus.Stack("IPMCAST");
iBusURL url = new iBusURL("ibus://226.1.3.5/Talk");
MyReceiver receiver = new MyReceiver();
stack.subscribe(url,receiver);
```

Der Protokollstack

Im obigen Beispiel wurde ein sehr einfacher Protokollstack verwendet, der in etwa die gleiche Funktionalität bietet wie ein MulticastSocket. Einer der wichtigsten Vorteile von iBus ist jedoch, dass man diesen Protokollstack je nach Bedarf aus einer Sammlung von Protokollobjekten zusammenstellen kann. An die Qualität einer Übertragung können sehr unterschiedliche Anforderungen gestellt werden. Hier ein paar Beispiele.

- Für die Übertragung von Audiodaten, die bei Eintreffen der Daten sofort wiedergegeben werden, ist es wichtig, einen einfachen Stack zu verwenden, der die Wiedergabe nicht unnötig verzögert. Wenn einzelne Datenpakete verloren gehen, ist dies meist kaum hörbar oder kann vom Hörer interpoliert werden.

- Bei der Übertragung von Software oder Anwendungsdaten, zum Beispiel einem Update von Java-Anwendungen, die auf mehreren Rechnern innerhalb eines Intranets installiert sind, ist zwar die Zeit für solch eine Übertragung zweitrangig, aber der Verlust auch nur geringster Datenmengen verheerend und nicht zu tolerieren. Hierfür benötigt man eine gesicherte Übertragung.

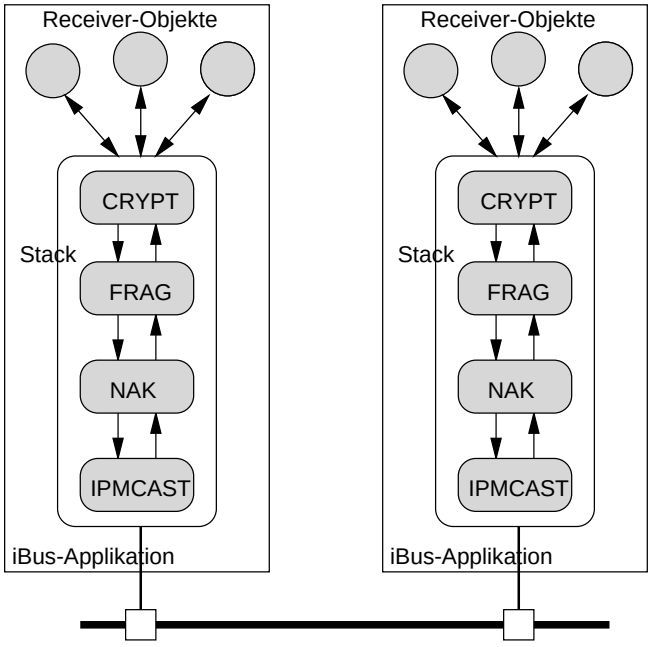
- Sind die zu übertragenden Daten sehr groß, zu groß, als dass sie in ein Datagrammpaket passen, müssen sie zerlegt und wieder reassembliert werden.

Datagramme sind üblicherweise bis zu 8192 Byte groß, was zum Beispiel für die Übertragung von Nachrichten in einem Newsticker oder in einem Blackboardsystem völlig ausreichend ist, aber für die Übertragung von Soundtracks oder von Softwarepaketen nicht ausreicht.

- Daten, die für ein Unternehmen schützenswert sind, zum Beispiel Daten über Kunden oder Umsätze, erfordern eine Verschlüsselung.
- Wenn die Übertragungszeit relativ teuer ist, kann es sich lohnen, die zu übertragenden Daten zu komprimieren. Je nach Beschaffenheit der Daten kann dies eine Verringerung der Übertragungsmenge um einen Faktor 2 bis 20 bewirken.
- Anwendungsserver, die eine extrem hohe Verfügbarkeit bieten müssen, werden häufig repliziert. Das heißt, ihr Datenbestand wird vollständig auf einem oder mehreren gleichen Servern gespiegelt, so dass diese, falls der eigentliche Server ausfällt, einspringen können und das Gesamtsystem weiterlaufen kann. Dies wird zum Beispiel in Banken oder Kernkraftwerken gemacht.

Für diese und andere Fälle bietet iBus die Möglichkeit, spezielle Protokollstacks, die solche Anforderungen erfüllen, zu erzeugen. Diese sind modular aus Protokollobjekten aufgebaut und lassen sich auf unterschiedliche Weise zusammenstellen.

Abbildung 2.2: Beispiel eines iBus-Protokollstacks



Die wichtigsten dieser Protokollobjekte sind die folgenden:

- IPMCAST stellt die Funktionalität der Klassen java.net.MulticastSocket und java.net.DatagramSocket zur Verfügung und bildet damit als unter-

stes Element die Basis der meisten Protokollstacks. IPMCAST kann auch allein eingesetzt werden und verhält sich dann identisch zum normalen Multicast.

- LOCALBUS ist eine lokale Alternative zu IPMCAST, die genau dann zum Einsatz kommt, wenn der Multicast nur innerhalb einer virtuellen Maschine eingesetzt werden soll. So können zum Beispiel mehrere Applets, die alle innerhalb eines Browsers (zum Beispiel innerhalb einer HTLM-Seite) ablaufen oder mehrere Applikationen, die auf einem Rechner miteinander kommunizieren sollen, iBus als eine Art Ereignisbus verwenden.
- TCP stellt TCP/IP-Verbindungen unter iBus zur Verfügung.
- REACH steuert und kontrolliert die Mitgliedschaft in einem Kanal. Diese Schicht sendet in regelmäßigen Abständen ein kurzes Signal, einen so genannten *Herzschlag*, über den neue Mitglieder oder das Fehlen bisheriger Mitglieder entdeckt werden können und erstellt so ein Bild der erreichbaren Mitglieder. Außerdem ist dadurch eine Ausfallerkennung möglich.
- NAK sorgt durch negative Bestätigung für die Nachlieferung von verloren gegangenen Nachrichten. Bemerkt der Empfänger, dass ein Paket in einer Sequenz von Paketen fehlt, meldet er dies dem Sender und bekommt das Paket nachgeliefert.
- FRAG bietet die Möglichkeit, große Nachrichten in kleine, dem Kommunikationsprotokoll angepasste Paketgrößen zu fragmentieren und reihenfolgegetreu wieder zusammenzubauen. Dies ist bei allen Nachrichten erforderlich, die zum Beispiel die UDP-Datagrammgröße überschreiten, wie Files, Audio- oder Videoströme oder Bilder.
- PULL ermöglicht das explizite Anfordern einer Information. Auf diese Weise kann die Übermittlung auch vom Empfänger angestoßen werden.
- CRYPT erlaubt die Verschlüsselung und Entschlüsselung von gesendeten Daten.

Bei der Erzeugung eines neuen Stacks wird der Erzeugungsmethode von Stack eine durch Doppelpunkte getrennte Liste von Protokollobjekten als String übergeben.

```
Stack s1 = new Stack("FRAG:FIFO:REACH:IPMCAST");
Stack s2 = new Stack("FRAG:FIFO:NAK:REACH:IPMCAST");
Stack s3 = new Stack("NAK:REACH:IPMCAST");
```

Für häufig gebrauchte Protokollstacks kann ein Alias definiert werden, der diesen String ersetzen kann. So steht zum Beispiel der Stack "PULL:FRAG:FIFO:NAK:-REACH:IPMCAST" unter dem Alias Reliable zur Verfügung. Es lassen sich auch eigene neue Protokollobjekte erzeugen. Zum Beispiel ist für die Kompression von zu übertragenden Daten eine genaue Kenntnis der Datenstruktur wichtig, um eine möglichst hohe Kompressionsrate zu erreichen. Daher kann es sinnvoll sein,

sich ein eigenes Protokollobjekt COMPR zu definieren und mit den passenden Protokollobjekten zu kombinieren.

```
Stack s4 = new Stack("CRYPT:Reliable");
Stack s5 = new Stack("COMPR:Reliable");
Stack s6 = new Stack("Vsync");
```

Diese Protokollstacks können für die unterschiedlichen Bedürfnisse in den verschiedenen Anwendungen eingesetzt werden. Dabei kann eine Anwendung durchaus mehrere Protokollstacks verwalten, muss aber sicherstellen, dass bei Sender und Empfänger der gleiche Stack verwendet wird. Der Stack s1 kann für die Übertragung von Audioströmen, s2 für die Verschickung von Softwareupdates, s3 für ein Blackboardsystem, s4 für die Übertragung unternehmenskritischer Daten und s5 für zu komprimierende Daten eingesetzt werden. Ein Protokollstack, der für die Replikation von Serverdaten geeignet ist, ein Konzept, das als virtuelle Synchronität bekannt ist, soll unter dem Alias Vsync wie im Stack s6 noch entwickelt werden.

2.2 Kommunikation über Stellvertreter

Für die Datenkommunikation in verteilten Anwendungen sind die vorgestellten Sockets eine einfache, solide und flexible Technik, die für viele Anwendungsfälle eine ausreichende Lösung darstellen. Doch da sich Sockets auf die Übertragung von Daten beschränken und die Semantik dieser Daten unberücksichtigt lassen, müssen auf Anwendungsebene Protokolle entwickelt werden, die für die semantische Interpretation dieser Daten sorgen. Die Entwicklung solcher Protokolle ist oft zeitaufwendig und fehleranfällig. Darüber hinaus steht in objektorientierten Programmiersprachen ein Rahmenwerk für die Semantik von Daten zur Verfügung – die Objekte. In lokalen Anwendungen kommunizieren Objekte mit anderen Objekten durch die (mit Semantik belegten) Methoden. Es wäre daher wünschenswert, wenn man für den verteilten Fall ein hierzu konsistentes Kommunikationsparadigma zur Verfügung hätte, das den entfernten Aufruf von Methoden erlauben würde. Ein hierfür geeignetes Konzept stellt der Remote Method Invocation, kurz RMI, dar.

2.2.1 Das Konzept RMI

In den 80er Jahren wurde für das prozedurale Programmierparadigma eine Technologie zum Aufruf von Prozeduren auf entfernten Rechnern entwickelt, der Remote Procedure Call (RPC). Er erlaubt den Aufruf von Prozeduren, die sich in einem anderen Prozessraum auf demselben oder auf einem entfernten Rechner befinden. Schon hierbei sind mehrere technische Hürden zu überwinden: Im lokalen Fall werden Daten einfach als Referenz übergeben (häufig als Pointer bezeichnet), die auf die physikalische Speicheradresse verweisen. In einem anderen Adressraum haben solche Referenzen keine (zumindest nicht die richtige) Bedeutung mehr. Daher müssen die referenzierten Daten im verteilten Fall als Kopie übergeben werden. Zum anderen kann man sich bei der Kommunikation

zwischen heterogenen Rechnerarchitekturen nicht mehr darauf verlassen, dass die interne Darstellung von Daten auf einem anderen Rechner die gleiche ist wie auf dem Ursprungsrechner. Die zu kopierenden Daten müssen zusätzlich in ein plattformunabhängiges Datenformat wie zum Beispiel das XDR (External Data Representation), das zu diesem Zweck von Sun entwickelt wurde, übersetzt und auf der Empfängerseite wieder in eine interne Darstellung zurückübersetzt werden.

Diese Technik war durchaus erfolgreich und wurde (und wird) in vielen verteilten Systemen eingesetzt. Das Distributed Computing Environment (DCE) von der Open Software Foundation (OSF) basiert zum Beispiel auf dieser Technologie und bietet darauf aufbauend eine Middlewareplattform für verteilte Systeme, die ein sehr ausgereiftes Sicherheitskonzept, die Verwaltung von Netzstrukturen in logischen Zellen, einen Zeit-, Datei- und Verzeichnisdienst und weitere Basisdienste zur Verfügung stellt.

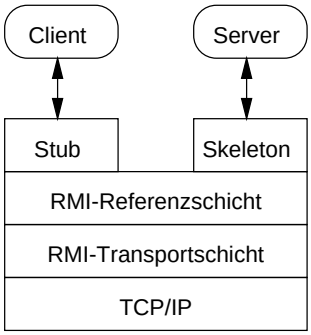
Die Übertragung dieses Konzeptes auf die Objektorientierung ist komplizierter, als man zunächst vermuten mag, da sich weitere technische Schwierigkeiten in den Weg stellen. In objektorientierten Sprachen werden Referenzen nicht als physikalische Speicheradressen, sondern als logische Adressen behandelt, so dass es wünschenswert ist, dieses Konzept auch in einem verteilten Szenario nahtlos aufrechtzuerhalten. Ebenso muss in einem Objektsystem wie dem von Java die Speicherverwaltung durch Garbage Collection bestehen bleiben, wenn die Objekte von entfernten Systemen aus referenziert werden. Mit dem Remote Method Invocation (RMI) steht ein solcher objektorientierter Mechanismus zur Verfügung, der hier anhand des Java-RMI erläutert werden soll, für weitere Informationen siehe [Downing 1998] oder [Sun 1999].

RMI ist ein Mechanismus, um Methoden von entfernten Objekten aufrufen zu können. Dabei bietet der RMI eine weitgehende Transparenz, so dass, nach einer anfänglichen Initialisierung, ein Aufruf genauso verwendet wird wie im lokalen Fall. Der RMI ist in mehreren Schichten aufgebaut, wobei die oberste Schicht die darunterliegenden Schichten, wie die Socketkommunikation und das Serialisieren und Deserialisieren von Parametern und Ergebniswerten, verdeckt.

Beim RMI gibt es jeweils einen Dienstanbieter, den *Server*, und einen Dienstnehmer, den *Client*. Zwar sind beide normale Objekte, die in Java implementiert sind, doch zumindest der Server muss als solcher gekennzeichnet sein und für den entfernten Zugriff vorbereitet werden, ehe er vom Client genutzt werden kann. Während im nicht-verteilten Fall das Verhältnis zwischen zwei Objekten symmetrisch ist (jedes Objekt kann ein anderes aufrufen, solange es eine Referenz darauf hat), ist dies bei RMI asymmetrisch.

Der Server muss die Schnittstelle, die er für den entfernten Zugriff zur Verfügung stellt, in einer Interface-Beschreibung, die vom Interface Remote abgeleitet ist, dokumentieren. Aus dieser Beschreibung werden dann durch einen speziellen Compiler zusätzliche Klassen erzeugt, die sich intern um die Kommunikationsabwicklung zwischen Client und Server kümmern, der Stub und das Skeleton. Der Stub ist ein Stellvertreterobjekt, das die gleiche Schnittstelle wie das Serverobjekt anbietet und einen Aufruf zu seinem Serverobjekt weiterleitet. Er muss entweder auf dem Rechner des Clients hinterlegt oder zur Laufzeit vom Serverrechner durch den RMIClassloader geladen werden. Das Skeleton verbleibt auf

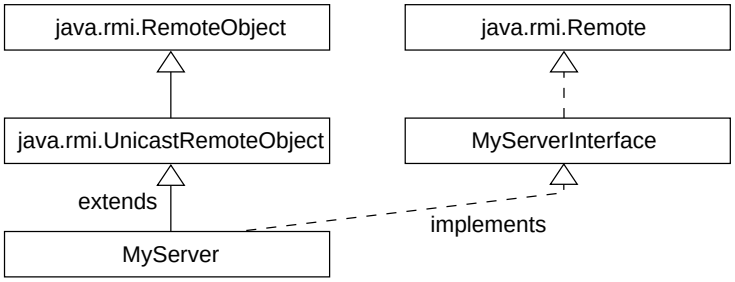
Abbildung 2.3: Die Schichten der RMI-Architektur



Serverseite und nimmt die Aufrufe des Stubs entgegen, bereitet sie auf, übermittelt den Aufruf an das Serverobjekt, erwartet das Ergebnis und sendet dieses zurück an den Stub. Stub und Skeleton bilden zusammen eine Protokollschicht in der RMI-Architektur. Die darunterliegende Referenzschicht (siehe Abbildung 2.3) dient dieser, um den jeweiligen Kommunikationspartner zu finden. Teil dieser Schicht ist auch der Namensdienst, die Registry. In der Transportschicht, die nicht mit der Transportschicht aus dem ISO/OSI-Modell verwechselt werden darf, werden Kommunikationsverbindungen verwaltet und die Kommunikation abgewickelt. Sie stützt sich selbst dabei auf TCP/IP.

Da sich verteilte Objekte nicht genauso verhalten können wie lokale Objekte, erben RMI-Objekte nicht direkt von der Klasse java.lang.Object, sondern von der Klasse java.rmi.RemoteObject. Dadurch werden einige Methoden von java.lang.Object überladen, um diese an die Bedingungen in verteilten Systemen anzupassen. Von dieser ist wiederum die Klasse java.rmi.UnicastRemoteObject abgeleitet, die einen Dienst mit Punkt-zu-Punkt-Verbindungen anbietet. Ein Objekt, auf das über RMI zugegriffen werden soll, muss von dieser Klasse erben. Die Schnittstelle, die dieses Objekt für den entfernten Zugriff anbieten möchte, muss extra in einer Schnittstellendefinition angegeben werden. Diese Interface-Definition ist Grundlage für die Erzeugung der Stubs und Skeletons, so dass nur Methoden, die hier angegeben werden, auch auf einem entfernten Rechner zur Verfügung stehen. Damit alle solchen RMI-Dienste generisch behandelt werden können, muss dieses Interface das Interface java.rmi.Remote erweitern. Insgesamt kommt es also zu der in Abbildung 2.4 dargestellten Vererbungshierarchie.

Abbildung 2.4: Die RMI-Vererbungshierarchie für Serverobjekte



Auch der Client muss, im Vergleich zum lokalen Aufruf, modifiziert wer-

den. Einerseits muss er die zusätzlichen Ausnahmen, die durch die verteilte Kommunikation auftreten können, abfangen. Andererseits muss er eine initiale Verbindung zu einem entfernten Objekt herstellen.

Das Prinzip des RMI und die nötigen Schritte sollen hier anhand eines einfachen Beispiels verdeutlicht werden. In diesem Programm gibt es zwei Klassen: einen Schläger (*Bat*) und einen Ball. Der Schläger schlägt den Ball, d.h., der Schläger ruft die Methode hit() vom Ball auf, und der Ball gibt aus, dass er geschlagen wurde.

Als erstes muss der Server seine Dienste, die er für den entfernten Zugriff zur Verfügung stellen möchte, in einem Interface bekannt geben, das hier RemoteBall heißen soll. Dieses muss von der Klasse Remote abgeleitet werden und die Methoden enthalten, auf die entfernt zugegriffen werden soll, in diesem Fall hit(). Da bei dem entfernten Aufruf dieser Methode zusätzliche Fehlerquellen auftreten können, die durch eine RemoteException angezeigt werden, muss bei Methoden angegeben werden, dass sie diese Ausnahme weiterreichen können.

```
import java.rmi.*;

public interface RemoteBall extends Remote {
  public void hit() throws java.rmi.RemoteException;
}
```

Das eigentliche Serverobjekt, der Ball, muss nicht nur diese Schnittstelle implementieren, sondern auch von RemoteObject bzw. einer Unterklasse wie UnicastRemoteObject, die eine Punkt-zu-Punkt-Kommunikation ermöglicht, abgeleitet sein. Im Konstruktor der Klasse Ball muss der Konstruktor der Superklasse UnicastRemoteObject durch super() aufgerufen werden, damit dieser sich richtig initialisiert. Auch dabei können Ausnahmen entstehen, so dass der Konstruktor insgesamt eine RemoteException erzeugen kann. Wenn dann ein Ball erzeugt wurde, kann dieser bei der Registry durch den Aufruf Naming.rebind() angemeldet werden.

```
import java.rmi.*;
import java.rmi.server.*;

public class Ball extends UnicastRemoteObject
  implements RemoteBall {

  public Ball() throws RemoteException {
    super();
  }

  public void hit(){
    System.out.println("Ball has been hit");
  }

  public static void main(String args[]){
```

```
    try {
      Ball ball = new Ball();
      Naming.rebind("Ball", ball);
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Jetzt müssen der Stub und das Skeleton der Klasse Ball erzeugt werden. Hierzu dient das Programm rmic, das als Parameter eine kompilierte Klasse annimmt, die wie die Klasse Ball für die Verteilung präpariert ist.

```
sun> rmic Ball
```

Dies erzeugt die Klassen Ball_Stub.class und Ball_Skel.class. Das Skeleton wird auf der Seite des Servers benötigt und steht dem Server üblicherweise zur Verfügung. Der Stub andererseits wird auf der Seite des Clients benötigt. Entweder wird der Stub dem Client als File zur Verfügung gestellt, oder der Client muss dieses File über das Netz nachladen. In diesem Fall muss für das System des Clients ein SecurityManager eingerichtet werden, der die Übertragung des Codes überwacht. Mit System.setSecurityManager (new RMISecurityManager()); kann der Client dies tun und den Code nachladen. Ehe der Server gestartet werden kann, muss die Instanz, die für den Namensdienst und für die Annahme von Anfragen verantwortlich ist, die rmiregistry, gestartet werden.

```
sun> rmiregistry
sun> java Ball
```

Der Client

Auch der Client muss für den entfernten Zugriff vorbereitet werden. Doch im Gegensatz zum Server muss er nicht zusätzliche Interfaces oder Oberklassen beerben. Um den Server finden zu können, wendet er sich an den Namensdienst auf dem Rechner des Servers und fragt mittels des Namens, mit dem der Server dort angemeldet ist, nach dem gewünschten Dienst. Dies tut er mit dem Aufruf Naming.lookup(). Als Parameter wird dieser Methode eine URL der Form rmi://some.server.com/ServiceName übergeben. Als Ergebnis wird eine Referenz auf ein Objekt vom Typ RemoteObject zurückgeben, das, um die spezifischen Methoden des Dienstes nutzen zu können, auf den richtigen Typ eingeschränkt werden muss (*casting*). Diese Referenz kann dann wie eine lokale Referenz verwendet werden. Lediglich die zusätzlichen Ausnahmen, die durch den entfernten Aufruf auftreten können, müssen abgefangen werden.

```
import java.rmi.*;

public class Bat {

  public Ball ball;
```

```
  public void play(RemoteBall ball) {
    try {
      ball.hit();
    } catch (RemoteException e) {
      System.out.println(e);
    }
  }

  public static void main (String args[]){
    Bat bat = new Bat();
    try {
      System.setSecurityManager (new RMISecurityManager());
      RemoteBall remoteBall= (RemoteBall)
          Naming.lookup("rmi://sun.informatik.uni-hamburg.de/Ball");
      bat.play(remoteBall);
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

Der Start dieses Programms auf einem beliebigen Rechner (hier auf dem Rechner win), der eine TCP/IP-Verbindung zu dem Serverrechner (hier sun) hat, führt, wohlgemerkt auf dem Serverrechner, zu der folgenden Ausgabe:

```
win> java Bat

sun| Ball has been hit
```

2.2.2 Kommunikation in heterogenen Sprachumgebungen

Das Konzept des RMI, wie er bisher vorgestellt wurde, eignet sich für die Überwindung der Heterogenität hinsichtlich der verwendeten Hardware- und Betriebssystemplattform. Der Java-RMI ermöglicht aber nicht eine Überwindung einer Sprachheterogenität, sondern setzt die Implementierung sowohl des Clients als auch des Servers in Java voraus. Das Konzept des RMI ist aber sehr wohl geeignet, um auch dieses Problem zu überwinden.

Eine der wichtigsten Bemühungen der Softwareindustrie auf dem Gebiet der verteilten Systeme ist die Erschaffung eines solchen sprachunabhängigen Programmiersystems. Die Organisation, die diese Standardisierung betreibt, ist die Object Management Group, kurz OMG. Die OMG hat sich zur Aufgabe gesetzt, eine netzwerktransparente Kommunikation heterogener Systeme auf der Basis objektorientierter Softwarekomponenten zu ermöglichen. Dabei versteht sich die OMG als Dachorganisation, die die Standardisierung steuert und vorantreibt, diesen Standard aber nicht selbst implementiert. Der daraus entstandene Standard trägt den Namen CORBA (Common Object Request Broker Architecture).

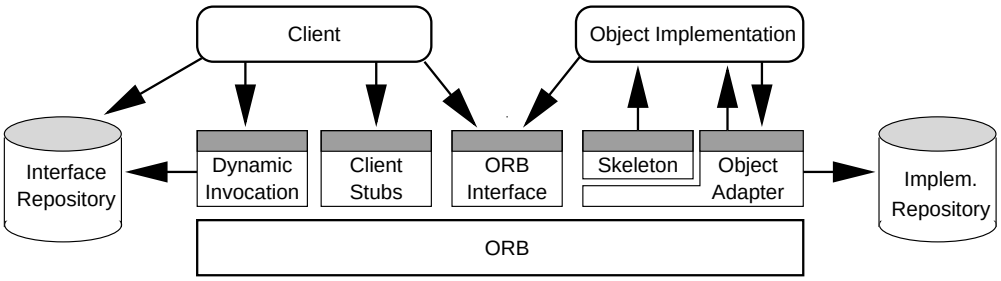
CORBA ist nicht eine Sprache, sondern eine Middlewareplattform. Sie stellt eine Infrastruktur zur Programmierung von verteilten Systemen dar. Zur Pro-

grammierung benötigt man eine Programmiersprache, die von CORBA unterstützt wird. Dazu gehören zum Beispiel C, C++, Smalltalk, Ada und natürlich Java. Der größte Vorteil von CORBA ist, dass man Teile seiner Applikation in verschiedenen Sprachen schreiben kann und doch alle Teile miteinander kooperieren können. Um dies zu erreichen, werden die verschiedenen Teile in einer weitgehend sprachunabhängigen Notation, der Interface Definition Language (IDL), definiert, die dann von einem Compiler in die entsprechende Zielsprache übersetzt werden kann, in der diese Schnittstellen schließlich implementiert werden müssen. Auf diese Schnittstellen kann schließlich von anderen Komponenten aus zugegriffen werden, und zwar nicht nur lokal, sondern auch entfernt über ein Kommunikationsnetz. Sehr gute Bücher zu diesem Thema sind [Siegel 1996] und [Redlich 1999].

Die CORBA-Architektur

CORBA ist eine Architektur, nicht ein Stück Software oder eine Programmiersprache. Diese Architektur besteht aus verschiedenen Komponenten, die in Abbildung 2.5 gezeigt sind und im Folgenden kurz erläutert werden. Das Kernstück ist der Object Request Broker (ORB), der dafür verantwortlich ist, eine Anfrage eines CORBA-Objekts, dem *Client*, an ein anderes CORBA-Objekt, den *Server*, entgegenzunehmen, den Server zu lokalisieren, den Aufruf weiterzuleiten und die Antwort wieder zum Client zu transportieren. Für den Client ist der Prozess der Lokalisierung des Servers verdeckt, er hat lediglich eine Referenz auf das Serverobjekt, ohne zu wissen, wo dieses Objekt sich befindet.

Abbildung 2.5: Die CORBA-Architektur



Der Client spricht den ORB nicht direkt an. Für das Serverobjekt gibt es einen Repräsentanten auf der Seite des Clients, den Client-Stub, der dem Client dieselbe Schnittstelle bietet, wie das Serverobjekt selbst. Der Client-Stub übernimmt die Aufgabe, mit dem ORB zu kommunizieren.

Es gibt noch einen weiteren Mechanismus, mit dem der Client auf ein Serverobjekt zugreifen kann, nämlich einen, der ohne einen Repräsentanten des Serverobjekts, dem Stub, auskommt. Dies ist insbesondere dann sinnvoll, wenn die Schnittstelle des Serverobjekts dem Client zur Übersetzungszeit nicht bekannt war. In solchen Fällen bietet CORBA einen dynamischen Mechanismus, das Dynamic Invocation Interface (DII).

Auf Standardoperationen von CORBA kann ein Client über ein ORB-Interface direkt zugreifen. Dies dient insbesondere zum Starten und Initialisieren des

ORBs.

Das Interface Repository enthält die Beschreibung aller registrierten Objekte, ihrer Methoden und Attribute. Durch eine Programmierschnittstelle kann der Client auf die Ablage zugreifen und das Interface darin enthaltener Objekte lesen.

Auf der Seite der Objektimplementierung, dem Server, stellt der Objektadapter die direkte Schnittstelle zum ORB dar. Er erhält Aufrufe für Objekte aus dem ORB, startet und instantiiert, wenn nötig, die Objektimplementierungen und übergibt den Aufruf an die Objektimplementierungen. Der Objektadapter registriert die Klassen, die er unterstützt, und deren Objektimplementierungen in der Implementierungsablage. Für neue Instanzen vergibt er eindeutige Objektreferenzen und verwaltet diese. CORBA schreibt für jede konforme ORB-Implementierung einen Standard-Objektadapter (den Basic Object Adapter, BOA) vor. Eine Implementierung kann aber auch weitere Objektadapter enthalten, ebenfalls standardisiert ist inzwischen der POA (Portable Object Adapter).

Das Implementation Repository stellt einen Namensdienst zur Verfügung, durch den erfragt werden kann, welche Klassen auf einem Server unterstützt und welche Objekte instantiiert sind, und verwaltet deren Objektreferenzen. Hier können auch weitere Informationen abgelegt werden, die z.B. zur Sicherheitsunterstützung oder zur Abrechnung dienen können.

Der Object Request Broker ist im CORBA-Standard durch sein Interface definiert. Er kann von verschiedenen Herstellern verschieden implementiert werden, solange das Interface dem Standard entspricht. Dies hat zur Folge, dass die Implementierungen von ORBs durchaus sehr unterschiedlich sein können. Es gibt ORBs, die auf Bibliotheken basieren, die zu den Stubs bzw. den Skeletons hinzugelinkt werden; es gibt ORBs, die auf einem Daemon-Mechanismus aufbauen; und es gibt solche, die zentral von einer Servermaschine verwaltet werden.

Um ein Objekt ansprechen zu können, ist jedem Objekt eine eindeutige Referenz zugewiesen. Diese Referenz ist die Information, die nötig ist, um ein Objekt über einen ORB zu identifizieren und anzusprechen. Diese Referenz wird einem Objekt bei der Erzeugung durch den ORB zugeteilt und behält so lange ihre Gültigkeit, wie das zugehörige Objekt existiert, selbst wenn das Objekt seinen Standort wechselt.

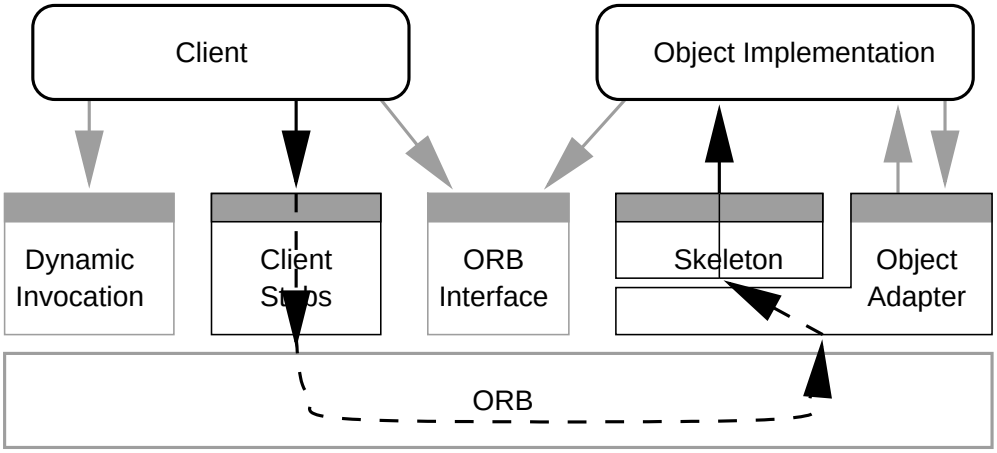
Diese Referenz kann anderen Objekten verfügbar gemacht werden, die darüber auf das Objekt zugreifen können. Dies kann auf verschiedene Weisen geschehen: Eine Referenz kann in einem File oder in einer Datenbank abgelegt sein und von einem Client herausgelesen werden. Oder der Client kann sich an einen Dienst wenden, der Objektreferenzen zur Verfügung stellt. Solche Dienste sind zum Beispiel der Naming Service oder der Trader Service. Ein Naming Server kann eine Abbildung von einem Namen auf eine Referenz erzeugen. Ein Trader kann verschiedene Objekte mit dem gleichen Service, aber unterschiedlichen Konditionen oder mit unterschiedlichen Qualitätsmerkmalen anbieten und den Client eines dieser Objekte auswählen lassen bzw. für den Client eine geeignete Auswahl treffen.

Hat ein Client eine Referenz auf ein Objekt erhalten, so kann er Operationen

auf diesem Objekt abrufen. Hierfür stehen dem Client zwei Mechanismen zur Verfügung: der dynamische Aufruf über das DII oder der statische Aufruf über den Client-Stub. Das DII wird zum Beispiel dann verwendet, wenn der Typ des aufzurufenden Objekts zur Übersetzungszeit nicht bekannt war. Der häufigere und einfachere Fall ist allerdings der Aufruf über einen Stub. Da der Stub einen Stellvertreter des Objekts auf der Clientseite darstellt, funktioniert der Aufruf des Objekts, aus Sicht des Client, genauso wie ein lokaler Aufruf.

Der Stub verpackt die Parameter in geeigneter Form (*marshalling*) und wendet sich dann mit diesem Aufruf an den ORB, der ihn an den Objektadapter weiterleitet. Der Objektadapter aktiviert das Objekt, erzeugt notfalls sogar eine neue Instanz und reicht den Aufruf an das Skeleton. Dieses entpackt (*demarshalling*) die Parameter wieder und ruft die Objektimplementierung auf (siehe Abbildung 2.6). Das Aufrufergebnis wird auf gleiche Weise in umgekehrter Richtung verpackt, verschickt und wieder entpackt und schließlich an den Client zurückgeliefert.

Abbildung 2.6: Der Weg eines Requests



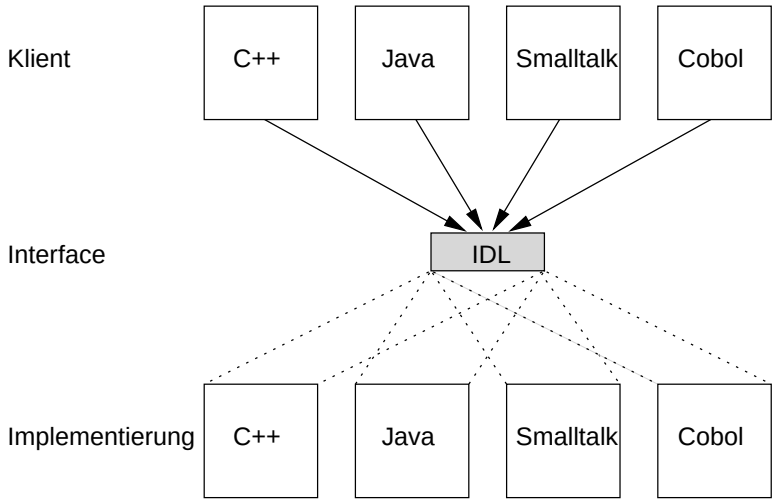
IDL

Durch die Interface Definition Language (IDL) werden Objekte mit ihren Methoden und Attributen beschrieben. Die IDL ist als eine Schnittstellenbeschreibungssprache angelegt, die unabhängig von den Sprachen ist, mit der diese Schnittstellen schließlich implementiert werden sollen. Dadurch kommt der IDL eine zentrale Rolle in CORBA zu. Durch sie ist es möglich, dass Client und Server in unterschiedlichen Sprachen implementiert sein können, ja nicht einmal wissen müssen, in welcher Sprache der jeweils andere programmiert ist. Syntaktisch ist die IDL an C++ angelehnt, ist aber um einige Elemente erweitert worden und hat keine algorithmischen Elemente.

Spezifikationen in IDL bestehen aus Modul-, Konstanten-, Typ-, Schnittstellen- und Ausnahmendefinitionen. Die Syntax der IDL soll im Folgenden vorgestellt werden, ohne jedoch auf alle Details einzugehen.

Die Definitionen in IDL sind in Modulen unterteilt. Jedes Modul bildet einen

Abbildung 2.7: Die Rolle der IDL



Namensraum (*scope*), in dem Identifikatoren von Typen, Konstanten, Attributen etc. definiert sind. Innerhalb des Namensraums müssen die Identifikatoren eindeutige Namen sein. Namen aus einem anderen Namensraum, also einem anderen Modul, können ebenfalls referenziert werden, wobei der Namensraum durch den Namen des Moduls angesprochen wird und Namensraum und Identifikator durch zwei Doppelpunkte getrennt werden.

Durch die Verbindung von Namensraum und Identifikator sind Namen auch global eindeutig, so dass CORBA-Dienste auch in sehr großen Netzen wie dem Internet eindeutig referenziert werden können. Alle weiteren Deklarationen müssen innerhalb eines Moduls definiert werden.

Das wichtigste Element in IDL ist sicherlich die Deklaration von Schnittstellen, den Interfaces. Interfaces können von anderen durch Vererbung, sogar durch mehrfache Vererbung, abgeleitet werden, wobei der Programmierer selbst die durch Vererbung bedingten Namenskonflikte beheben muss. Ein Interface bietet dann eine Reihe von Operationen. Jede Operation hat eine Signatur, welche aus dem Namen der Funktion, einem Resultattyp und einer Parameterliste besteht. Die Parameter in der Parameterliste bestehen aus einem Parameternamen, dessen Typ und einem der Attribute in, out oder inout. Ein in-Parameter wird nur vom Client zum Server übertragen, ein out-Parameter wird nur vom Server zum Client übertragen und ein inout-Parameter wird in beide Richtungen übertragen.

Neben den Operationen können auch Attribute definiert werden. Diese werden durch den Bezeichner attribute eingeleitet und werden dann durch einen Typ und einen Namen deklariert. Attribute, die nur gelesen werden dürfen, erhalten zusätzlich den Bezeichner readonly.

Operationen können Ausnahmen, wie sie aus Java oder aus C++ bekannt sind, verursachen, was in der Signatur einer Operation in IDL mit angegeben werden muss. Dazu kann eine Operationsdefinition eine raises-Klausel enthalten, in der Ausnahmen aufgeführt werden. Die Ausnahmen müssen ebenfalls deklariert werden, ehe sie verwendet werden können. Sie können entweder auf Ebene

des Moduls oder innerhalb eines Interfaces definiert werden. Anders als bei Java kann einer Ausnahme ein Kontext mitgegeben werden, der die verursachte Fehlermeldung genauer beschreiben kann.

Die einfachen Datentypen, die in IDL Verwendung finden, entsprechen in etwa denen von Java, wie zum Beispiel boolean, char, int oder float. Leider gibt es nicht für jeden Typ die genaue Entsprechung, so muss zum Beispiel statt eines int in Java ein long in IDL verwendet werden. IDL bietet die Möglichkeit, aus diesen einfachen Typen komplexere Datentypen durch ein Typsystem aufzubauen, so wie es in C++ üblich ist. Dafür lassen sich einfache Datentypen zu structs oder unions zusammenbauen und durch ein typedef mit einem spezialisierten Namen versehen. Auch sind Aufzählungen, die es in C++, aber nicht in Java gibt, Konstanten und mehrdimensionale Arrays in IDL vorhanden.

Als Beispiel soll hier wieder ein Chat dienen. Da CORBA eine typische Client/Server-Infrastruktur ist, soll dieses Beispiel hier auch als Client/Server-Architektur umgesetzt werden. Auf einer Servermaschine soll ein ChatServer gestartet werden können, bei dem sich einzelne ChatClients an- und abmelden können. Hierfür stellt der ChatServer die Operationen login() und logout() zur Verfügung, wobei der Client jeweils seinen Namen und beim Anmelden auch eine Referenz auf sich angeben muss. Nach einer Anmeldung kann der Client durch die Methode send() Nachrichten an den Chat schicken, wobei eine Nachricht jeweils aus seinem Namen und einem String besteht und in einem struct Message übergeben wird. Der Client soll jeweils von einer Anmeldung oder einer Abmeldung anderer Teilnehmer informiert werden und die gesendeten Nachrichten empfangen können. Das IDL-Interface eines solchen Dienstes könnte also folgendermaßen aussehen:

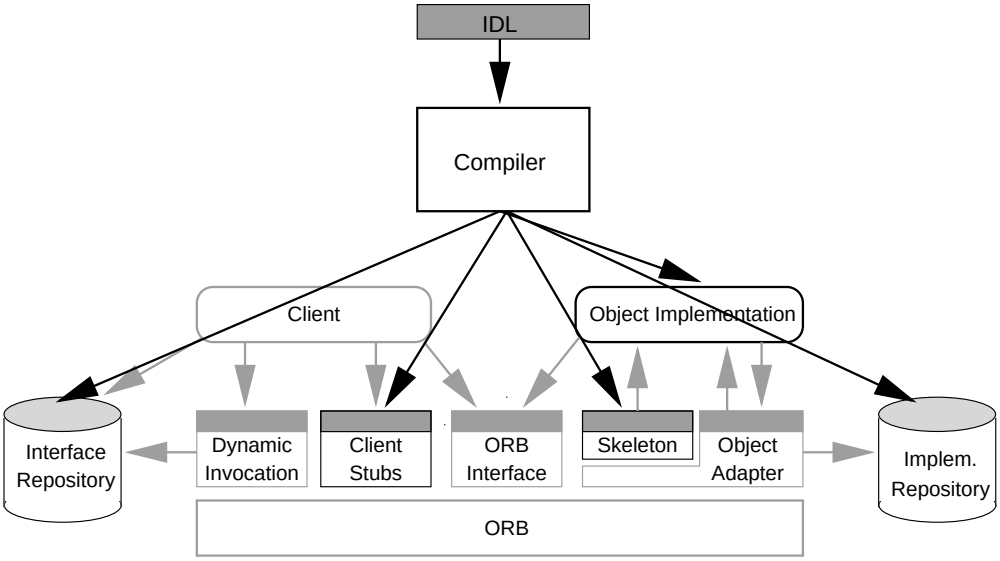
```
module Chat {

  typedef string Name;

  struct Message {
    Name name;
    string message;
  };

  exception UnknownName {};
  exception Reject {string reason;};

  interface ChatClient {
    void receiveEnter(in Name name, in ChatClient chatter);
    void receiveExit(in Name name);
    void receiveMessage(in Message message);
  };

  interface ChatServer  {
    const short MaxClients = 20;
    readonly attribute short numOfClients;
```

```
    void login (in Name name, in ChatClient chatter) raises (Reject);
    void logout (in Name name);
    void send (in Message message);
  };

};
```

Der IDL-Compiler

Hat man eine solche IDL-Schnittstelle definiert, kann man diese mit dem IDL-Compiler verarbeiten, der, je nach Komplexität des Beispiels und verwendetem CORBA-Produkt, mindestens die folgenden Ausgaben erzeugt.

Abbildung 2.8: Der IDL-Compiler



- Ein Client-Stub wird generiert, der eine Abbildung von den Konstrukten der Programmiersprache auf die Konstrukte in IDL vornimmt, diese IDL-Konstrukte dann in ein on-the-wire-Format übersetzt und den ORB aufrufen kann. Ebenso muss dieser das Ergebnis aus dem on-the-wire-Format auspacken und in Konstrukte der Zielsprache zurückübersetzen können.

- Ein Skeleton wird als das dazugehörige Gegenstück auf der Serverseite erzeugt, das die Informationen, die es vom Client-Stub erhält, wieder auspackt und mittels eines up-calls an das aufgerufene Objekt leitet.

- Ein Rahmen der Implementierung dieses Objekts wird ebenfalls vom IDL-Compiler erzeugt und muss vom Programmierer noch ausgefüllt werden.

- Als viertes erzeugt der Compiler alles, was im Interface- und im Implementation Repository eingetragen wird, und kümmert sich ebenfalls um die Eintragung.

Das IDL-Sprach-Mapping

Da die Schnittstellen in einem CORBA-System mit IDL spezifiziert werden, muss dieses auf eine Java-Schnittstelle (oder eine andere Sprache) abgebildet werden. Für jedes Element aus IDL, wie Modul, Interface und Attribut, muss genau festgelegt sein, wie dieses Element in eine Zielsprache umgesetzt wird. Nur dann ist eine Interoperabilität zwischen einer C++-Anwendung und einem Java-Programm oder zwischen verschiedenen ORBs möglich. Daher sind solche Abbildungen von IDL auf die Implementierungssprache, das *Mapping*, von der OMG standardisiert.

Auch für Java besteht ein Mapping-Standard, der nun kurz vorgestellt werden soll. Die IDL ist von ihrer Syntax recht stark an C++ orientiert, doch da auch Java eine gewisse Ähnlichkeit zu C++ aufweist, ist die Abbildung von IDL nach Java nicht sehr kompliziert. So stimmen die Basistypen wie boolean, char, float etc. in beiden Sprachen überein und können unmittelbar aufeinander abgebildet werden. Doch auch schon bei den einfachen Datentypen gibt es gewisse kleine Probleme, die der Entwickler genau kennen muss, um Schwierigkeiten zu vermeiden. Zum Beispiel gibt es in IDL sowohl Integertypen, die mit einem Vorzeichen behaftet sind (short, long und long long, die in jeweils 16, 32 und 64 Bit gespeichert werden), als auch solche ohne Vorzeichen (unsigned short, unsigned long und unsigned long long). In Java dagegen haben alle Integertypen stets eine Vorzeichenbehaftung. Ohne eine Vorzeichenbehaftung kann man mit 16 Bit den Wertebereich von 0 bis 65.535 abdecken, mit Vorzeichen wird dagegen der Wertebereich -32.768 bis 32.767 abgedeckt. Nun werden aber sowohl die IDL-Typen mit (z.B. short) als auch die Werte ohne Vorzeichen (unsigned short) auf die gleichen, vorzeichenbehafteten Typen in Java abgebildet (short). Dies kann zu Fehlern in der Interpretation des Wertes führen. Wenn zum Beispiel ein Parameter als IDL-unsigned short übergeben wird und einen Wert größer als 32.767 hat, dann wird er in Java als negativer Wert interpretiert. Man sollte also bereits in der IDL-Spezifikation auf vorzeichenlose Typen verzichten oder solche Fälle in der Implementierung abfangen. Bei der Übertragung von Zeichen (char) tritt ein ähnliches Problem auf. In Java werden Zeichen nach UNICODE mit 16 Bit kodiert, während sie in IDL für den Typ char mit nur 8 Bit dargestellt werden. Es wurde daher ein weiterer IDL-Typ mit 16 Bit eingeführt, der wchar heißt. Bei Fließkommazahlen tritt ein solches Problem allerdings nicht auf, da in beiden Fällen der gleiche Standard verwendet wird.

Für komplexe Datentypen bietet IDL in C-Tradition ausdrucksstarke Mechanismen wie typedef, struct und union, die in einer objektorientierten Sprache nicht nötig sind, da sie durch entsprechende Klassen definiert werden. Da diese Mechanismen aber nun einmal in IDL vorhanden sind, müssen sie auch geeignet abgebildet werden. Typedefs in IDL werden in Java schlicht ignoriert, so dass zum Beispiel für einen in IDL definierten typedef string Name; in Java weiterhin der Typ String eingesetzt wird. Der Aufzählungstyp enum ist in Java ebenfalls nicht vorhanden und muss durch die Erzeugung von Konstanten simuliert werden. Für structs und unions werden Klassen erzeugt, die die entsprechende Funktionalität zur Verfügung stellen.

Operationen und auch Attribute in IDL werden auf entsprechende Methoden

in Java abgebildet. Für Attribute werden Lese- und, wenn sie nicht als readonly deklariert sind, Schreibmethoden erzeugt, mit denen auf ein Attribut einer Klasse zugegriffen werden kann. Bei Operationen müssen auch die Parameter geeignet behandelt werden. In einem lokalen Programm werden Parameter als Referenz auf Objekte (oder als Wert bei Basistypen) übertragen. Dies ist bei entfernten Aufrufen nicht möglich, so dass Parameter üblicherweise als Kopie übergeben werden. Für Operationen, die nur in-Parameter haben, liegt der Fall einfach: Die Kopie wird bei der aufgerufenen Methode nur gelesen und nicht verändert, so dass keine weiteren Maßnahmen getroffen werden müssen. Doch bei out- und inout-Parametern müssen geänderte Werte wieder in den ursprünglichen Adressraum zurückkopiert werden.

IDL-Interfaces werden auf Java-Interfaces abgebildet, die anschließend durch entsprechende Klassen implementiert werden können. Durch diese Indirektion ist die Vererbung allerdings sehr viel flexibler: In IDL ist eine Mehrfachvererbung erlaubt, was in Java nur für Interfaces, aber nicht für Klassen gilt. Die Modulstruktur in IDL wird auf entsprechende Packages in Java projiziert, die in der in Java üblichen Verzeichnisstruktur abgelegt werden.

Aus der Definition des ChatClients wird das folgende Interface vom IDL-Compiler (in diesem Beispiel wurde der idl2java von Visigenic/Inprise [Inprise 1999] verwendet) generiert.

```
package Chat;

public interface ChatClient extends org.omg.CORBA.Object {
  public void receiveEnter(java.lang.String name, Chat.ChatClient chatter);

  public void receiveExit(java.lang.String name);

  public void receiveMessage(Chat.Message message);
}
```

Das generierte Interface vom ChatServer sieht folgendermaßen aus:

```
package Chat;
public interface ChatServer extends org.omg.CORBA.Object {
  final public static short MaxClients = (short) 20;
  public short numOfClients();

  public void login(java.lang.String name, Chat.ChatClient chatter ) throws Chat.Reject;

  public void logout(java.lang.String name);

  public void send(Chat.Message message);
}
```

Dies sind allerdings erst Interfaces und keine Klassen, die eine Implementierung enthalten. Die Implementierung erfolgt dann in von diesen abgeleiteten Klassen.

2.2.3 Migration von Objekten

Die bisher gezeigten Techniken für die verteilte Programmierung erlauben die Kommunikation von Objekten auf voneinander entfernten Rechnern. Doch ein ganz wichtiger Aspekt in der verteilten Programmierung ist die Migration von Objekten. Ein Objekt soll von einem Rechner auf einen anderen verschoben werden können und dabei seinen gesamten Informationsgehalt mitnehmen. Dies sollte möglichst zu jedem beliebigem Zeitpunkt erfolgen können, auch wenn auf diesem Objekt gerade eine Operation durchgeführt wird. Der gesamte Zustand und alle Referenzen auf andere Objekte müssen erhalten bleiben.

In den bisher gezeigten Techniken ist dies nicht möglich. Zwar erlauben Ansätze wie etwa iBus, RMI oder CORBA das Kopieren eines Objekts und – dank Java – ist auch der Code eines kopierten Objekts nachladbar, doch verliert ein Objekt dabei seine Identität. Objekte, die das Original referenzieren, halten auch nach dem Kopieren nur eine Referenz auf das Original und haben keinerlei Information über die Kopie. Das Original und die Kopie sind unterschiedliche Objekte. Sie werden unabhängig voneinander benutzt und ihr interner Zustand driftet auseinander. Eine konsistente Sicht auf das System ist nur unter erheblichem Aufwand zu erhalten. Der Versuch, eine Migration durch das Kopieren zu simulieren, gelingt deswegen nur unter sehr hohem Aufwand, da Referenzen auf das Original nach der Kopie nicht mehr gültig sind oder einzeln umgesetzt werden müssen.

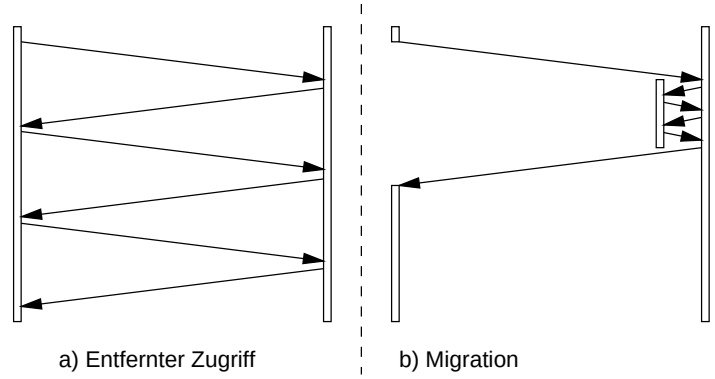
Die Migration von Objekten hat eine hohe Bedeutung. Sie ist technisch schwierig und war bisher nur in wenigen Systemen realisiert. Ein Beispiel eines solchen Systems ist Emerald [Black et al. 1987], [Jul 1989], allerdings ist dort der Code nicht migrierbar, so dass Objekte nur auf Rechner migrieren können, auf denen der übersetzte Klassen-Code schon vorhanden ist. In Java dagegen ist die Code-Migration sogar ein fundamentaler Bestandteil der Sprache. Nur dadurch können Applets über das Internet heruntergeladen und gestartet werden. Doch Migration von Objekten ist bisher in Java nicht möglich. Java bietet allerdings eine gute Grundlage für solch einen Mechanismus.

Die Migration ist für die folgenden Aufgabengebiete wichtig, wenn nicht gar erforderlich:

- Zeitaufwand minimieren. Der Zugriff auf ein entferntes Objekt über ein Netzwerk ist erheblich zeitaufwendiger als ein lokaler Zugriff. Dies ist ein Umstand, der sich auch in Zukunft nicht ändern wird. Die Verarbeitungsgeschwindigkeit von Rechnern steigt schneller als die Übertragungsgeschwindigkeit über ein Netzwerk. Dies bedeutet, dass bei mehrfachen (ausreichend vielen) Zugriffen eine Migration eines Objekts günstiger ist als eine entfernte Kommunikation. Natürlich ist dabei abzuwägen, welche Objekte migriert werden, wie groß die zu migrierenden Objekte sind und ob die Migration nicht schon mehr Zeit kostet als die entfernten Aufrufe.

- Lastbalancierung. Um die Belastung von Rechnern, etwa in einem Rechnerverbund oder Cluster, gleichmäßig zu verteilen, werden heutzutage einzelne Programme auf einem Rechner beendet und auf einem anderen wieder gestartet. Dabei muss häufig das gesamte System neu konfiguriert

Abbildung 2.9: Unterschied zwischen entferntem und lokalem Zugriff



werden. Mit Objektmigration ist es möglich, einfach Objekte von einem belasteten auf einen weniger belasteten Rechner zu verschieben.

- Besitz- oder Verantwortungswechsel. Oft sind an der Bearbeitung eines Objekts mehrere Firmen oder Abteilungen beteiligt. Dabei kann sich im Lebenszyklus eines Objekts der Besitzstand oder die Verantwortung über ein Objekt ändern. Dann sollten Objekte auch in den entsprechenden Verantwortungsbereich migriert werden können. Zum Beispiel kann eine Designabteilung eine CAD-Zeichnung erstellen, die anschließend in der Lagerverwaltung benötigt und an die Produktion verkauft wird. Solange am Design gearbeitet wird, sollte das CAD-Objekt auf dem Server der Designabteilung liegen. Nach Abschluss sollte es aber auf die Rechner der Produktion migrieren. Dennoch sollten sowohl die Designabteilung als auch die Lagerverwaltung weiterhin einen Zugriff auf dieses Objekt haben.

- Mobile Geräte. Eine wesentliche Einschränkung von mobilen Geräten im Gegensatz zu Computern, die fest mit einem Netzwerk verbunden sind, besteht darin, dass sie nicht fortlaufend, sondern eher sporadisch online sind. Die Kosten für die Anknüpfung ans Netz sind erheblich höher, und die Bandbreite ist geringer. Dies gilt in ähnlicher Weise auch für den PC zu Hause, der nur gelegentlich und kurzfristig über Modem an das Internet oder an ein Firmennetz angeschlossen wird. Bei solchen Geräten ist es von Vorteil, wenn zu bearbeitende Objekte nicht entfernt referenziert werden, sondern in dieses Gerät migriert werden können. Danach kann das Gerät wieder offline genommen und das Objekt bearbeitet werden. Zu einem passenden späteren Zeitpunkt kann dann das bearbeitete oder neu erzeugte Objekte wieder zurückmigriert werden.

Ein Produkt, das die Migration von Objekten ermöglicht, ist Voyager von ObjectSpace [ObjectSpace 1998], beschrieben auch in [Nelson 1998]. Voyager ermöglicht die Migration von Objekten für fast beliebige Objekte und dazu auf eine für den Programmierer recht einfache Art und Weise. Mit Voyager können die Fähigkeiten von Objekten dynamisch während der Laufzeit erweitert werden. Von einem Objekt lassen sich so genannte Facetten erzeugen, die zusätzliche Methoden für ein Objekt implementieren können.

Die Mobilität von Objekten ist ein Anwendungsfall für diese dynamische Erweiterung. Von einem Objekt a wird mit Mobility.of() eine Facette erzeugt, die ein Interface IMobility implementiert und damit die Methode moveTo() in zwei Varianten bereithält. In der ersten kann dieser Methode eine URL mitgegeben werden. Dann bewegt sich das betreffende Objekt auf den durch die URL spezifizierten Rechner. Dort muss lediglich eine Voyager-Laufzeitumgebung gestartet sein. In der zweiten wird eine Referenz auf ein entferntes Objekt, ein Proxy, übergeben, und das angesprochene Objekt bewegt sich auf den Rechner, auf dem auch das entfernte Objekt liegt.

```
try {
  IMobility mobileObj=Mobility.of(a);
  mobileObj.moveTo("win.informatik.uni-hamburg.de:8000");
} catch (MobilityException e) {
  System.out.println(e);
}
```

Referenzen auf ein sich bewegendes Objekt bleiben, aus Sicht des Programmierers, erhalten. Hinter den Kulissen sorgt Voyager dafür, dass ein Aufruf, der an ein solches Objekt geschickt wurde, zu dem aktuellen Aufenthaltsort des Objekts weitergeleitet wird. Dafür merkt sich die Voyager-Laufzeitumgebung, wohin migrierende Objekte verschoben wurden. Wenn anschließend versucht wird, auf diese zuzugreifen, erzeugt die Laufzeitumgebung eine interne Ausnahme, die den neuen Standort enthält. Der Proxy, der den Aufruf abgeschickt hat, fängt diese Nachricht ab, liest den neuen Standort und schickt den Aufruf erneut los. Dieser Mechanismus funktioniert sogar, wenn ein Objekt mehrfach verschoben wird.

Voyager unterbricht einen Methodenaufruf, der gerade ausgeführt wird, nicht, sondern wartet, bis dieser beendet ist, ehe ein Objekt bewegt wird. Allerdings werden neu eingehende Methodenaufrufe aufgehalten, die Migration durchgeführt und anschließend die aufgehaltenen Aufrufe zum neuen Standort weitergeleitet. Dies ist allerdings nur für Methoden möglich, die synchronisiert sind, also mit dem Schlüsselwort synchronized gekennzeichnet sind. Daher sollten möglichst alle Methoden eines zu verschiebenden Objekts synchronisiert sein. Darüber hinaus müssen alle zu migrierenden Objekte das Interface Serializable implementieren, da für die Migration intern der Serialisierungsmechanismus von Java eingesetzt wird.

Die Klasse Ball sollte also wie folgt angepasst werden, um migrierbar zu sein:

```
import java.io.*;

public class Ball implements IBall, Serializable {

  synchronized public void hit() {
        System.out.println("Ball has been hit");
  }
}
```

Eine Anwendung kann ein Objekt von der Klasse Ball nun verschieben. Im folgenden Beispiel wird auf dem Rechner sun ein Ball und ein Schläger erzeugt. Wenn der Schläger den Ball trifft, wird der Ball auf einen anderen Rechner verschoben. Hierfür stehen die Rechner lin und mac zur Verfügung, auf denen jeweils eine Voyager-Laufzeitumgebung gestartet sein muss.

```
import com.objectspace.voyager.*;
import com.objectspace.voyager.mobility.*;

public class Bat {

  public void play(IBall ball, String url) {
    try {
      ball.hit()
      Mobility.of(ball).moveTo(url);
    } catch (MobilityException e) {
      System.out.println(e);
    }
  }

  public static void main(String[] args) {
    try {
      Voyager.startup("7000");
      ClassManager.enableResourceServer();
      Bat bat = new Bat();
      IBall ball = (IBall) Proxy.of(new Ball());
      bat.play(ball,"//lin:8000");
      bat.play(ball,"//mac:9000");
      bat.play(ball,"//sun:7000");
    } catch( Exception exception ) {
        System.err.println( exception );
    }
    Voyager.shutdown();
  }
}
```

Bei der Ausführung dieses Programms ergibt sich folgendes Bild. Man beachte, dass sich die Referenz auf den Ball nicht ändert, sondern der Proxy den Ball stets auf dem richtigen Rechner findet.

```
lin> voyager 8000 -c http://sun:7000

mac> voyager 9000 -c http://sun:7000

sun> java Bat
sun| Ball has been hit
```

```
lin| Ball has been hit

mac| Ball has been hit
```

Üblicherweise ist die Migration für das migrierte Objekt vollständig transparent. Es erfährt nicht einmal, dass es bewegt wurde. In einigen Fällen kann dies aber sehr nützlich sein, wenn es doch davon erfährt, damit es hinter sich aufräumen oder an dem neuen Standort Vorkehrungen treffen kann. Ein Beispiel für einen solchen Fall wäre, wenn das Objekt in einer lokalen Datenbank gesichert ist und bei einer Migration auch von einer Datenbank in eine andere migrieren soll. Auch dieser Fall ist in Voyager abgedeckt. Wenn ein zu migrierendes Objekt zusätzlich das Interface IMobile implementiert, ruft Voyager in entsprechender Reihenfolge die Methoden preDeparture(), preArrival(), postArrival() und postDeparture() auf. Der Methode preDeparture() werden dabei sowohl der Ursprung als auch das Ziel als String übergeben. Die eigentliche Migration ist bei Aufruf von postArrival() vollendet, mit postDeparture() besteht aber noch die Möglichkeit, die inzwischen veraltete Kopie auf dem Ursprungsrechner aufzurufen, um hinter sich aufzuräumen. Eine entsprechende Erweiterung von Ball könnte also folgendermaßen aussehen:

```
import java.io.*;
import com.objectspace.voyager.mobility.*;

public class Ball implements IBall, IMobile, Serializable {

  synchronized public void hit() {
    System.out.println("Ball has been hit");
  }

  public void preDeparture(String source, String dest) {
    System.out.println("Ball about to move from "+source+" to "+dest);
  }

  public void preArrival() {}

  public void postArrival() {
    System.out.println("Move was successfull");
  }

  public void postDeparture() {}
}
```

Bei einer Migration eines Ball-Objekts von sun nach lin wird in diesem Fall die folgende Sequenz ausgegeben.

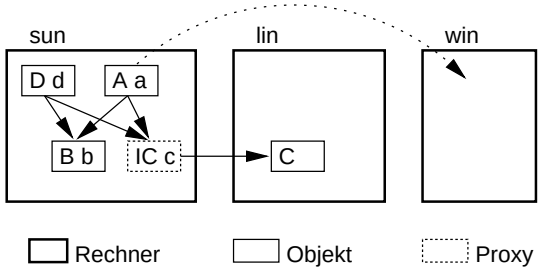
```
sun| Ball has been hit
sun| Ball about to move from tcp://sun:7000 to tcp://lin:8000
```

```
lin| Move was successfull
```

Das Verschieben von einfachen Objekten ist also in Voyager auf einfache Weise möglich. Was ist aber mit Objektstrukturen, die etwas komplizierter sind? Was geschieht mit Objekten, auf die das migrierende Objekt verweist? Werden sie ebenfalls migriert oder kopiert oder überhaupt nicht bewegt? Dies ist ein generelles Problem der Migration. Einerseits muss das migrierte Objekt an seinem neuen Standort auf alle Objekte, auf die es vorher zugreifen konnte, weiterhin zugreifen können, andererseits werden diese Objekte möglicherweise auch noch an dem alten Standort benötigt. Wenn man aber das Objekt kopiert, ist es zweimal vorhanden und Änderungen auf der einen Kopie bleiben bei der anderen Kopie unberücksichtigt. Eine andere Möglichkeit ist, die Objekte über Proxies zu referenzieren. Die Proxies lassen sich problemlos kopieren und leiten Aufrufe stets an das Original weiter. Dies hat aber den Nachteil, dass entfernte Aufrufe bezogen auf den Zeitaufwand erheblich teurer sind als lokale.

Zur Verdeutlichung soll hier ein Beispiel betrachtet werden, das in Abbildung 2.10 dargestellt ist: Ein Objekt a vom Typ A referenziert ein Objekt b vom Typ B ebenso wie ein Objekt vom Typ C über einen Proxy, der also das Interface IC implementiert und mit c bezeichnet werden soll. Das eigentliche Objekt liegt auf dem Rechner lin. Außerdem werden die gleichen Objekte von einem Objekt d referenziert. Nun soll das Objekt a vom Rechner sun auf den Rechner win verschoben werden.

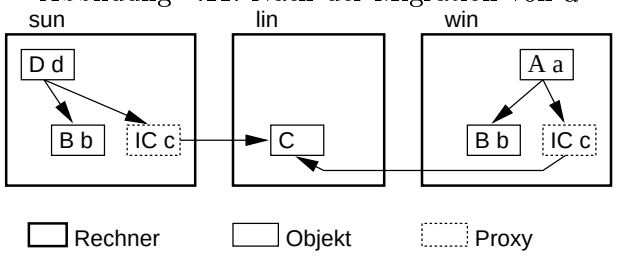
Abbildung 2.10: Eine Objektstruktur: Objekt a soll bewegt werden



Das Objekt a wird nach dem moveTo()-Befehl auf den Rechner win verschoben. Alle Referenzen müssen auch dort Gültigkeit haben, man spricht von referentieller Integrität, die gewahrt werden muss. Da auch das Objekt d Referenzen auf b und c hält, dürfen diese natürlich nicht einfach wegbewegt, sondern müssen kopiert werden. Für das Objekt b folgt daraus, dass es nun in zwei Versionen vorkommt und a und d unter Umständen verschiedene Sichten auf dieses Objekt erhalten. Die Konsistenz ist nicht gewährleistet. Für das Objekt c hingegen bedeutet dies lediglich, dass ein neuer Proxy auf das Original vorhanden ist. Der interne Mechanismus von Voyager sorgt dafür, dass auch diese Referenz bei der Garbage Collection berücksichtigt wird, indem ein Referenzzähler hochgezählt wird. Es ergibt sich das in Abbildung 2.11 gezeigte Szenario.

Ein weiteres Problem stellen natürlich auch lokale Referenzen auf zu migrierende Objekte dar, wie im obigen Beispiel das Objekt a. Diese verlieren durch das Verschieben des Objekts ihre Gültigkeit. Aus diesem Grund sollten Objekte, die verschoben werden sollen, ausschließlich über Proxies referenziert werden.

Abbildung 2.11: Nach der Migration von a



2.2.4 Asynchrone Aufrufe

In einem nicht-verteilten Softwaresystem sind Methodenaufrufe üblicherweise synchron, das heißt, der Aufruf blockiert so lange, bis die Methode ausgeführt worden ist und gegebenenfalls ein Ergebnis zurückgegeben wurde. Danach wird der nächste Befehl abgearbeitet. In verteilten Systemen sind aber häufig die Laufzeiten von Nachrichten so lang, dass es Sinn macht, den Aufruf nicht blockieren zu lassen, sondern gleich den nächsten Befehl abzuarbeiten und das Ergebnis erst zu einem späteren, geeigneten Zeitpunkt »abzuholen«. Dieses Verhalten wird als asynchron bezeichnet.

Diese Unterscheidung ist für nicht-verteilte Systeme, wie eben normale Java-Programme, nicht erheblich, für verteilte Systeme dagegen schon. In verteilten Systemen muss sehr viel flexibler auf die auftretenden Laufzeiten und Verzögerungen eingegangen werden. In Java allein kann ein asynchroner Aufruf durch die Erzeugung eines neuen Threads simuliert werden, dies bedeutet aber einen hohen Programmieraufwand und zusätzliche Komplexität. Voyager bietet daher eine einfachere Möglichkeit: Asynchrone Nachrichten werden hier durch das Konzept *Future* realisiert.

Um einen asynchronen Aufruf abzusetzen, wird die statische Methode invoke() der Klasse Future aufgerufen. Dieser Aufruf liefert sofort ein Ergebnis zurück, das allerdings zunächst nur ein Platzhalter und nicht das eigentliche Resultat darstellt. Das kann erst später anstelle des Platzhalters gelesen werden. Der Methode invoke() muss der geplante Methodenaufruf in seinen einzelnen Bestandteilen übergeben werden. Ein normaler Methodenaufruf besteht aus dem Aufrufziel (das Objekt auf dem die Methode ausgeführt werden soll), dem Methodennamen und den Aufrufparametern. Er hat zum Beispiel die folgende Form:

```
object.method(param1, param2);
```

Der Methode invoke() kann das Aufrufziel als Referenz übergeben werden, der Methodenname muss allerdings als String angegeben werden. Die Parameter können nicht einfach aufgelistet, sondern müssen in einem Objekt-Array gekapselt werden. Dabei müssen einfache Typen wie int oder bool in ihr Objekt-Pendant (Interger, Boolean) übertragen werden. Ein entsprechender asynchroner Aufruf sieht also folgendermaßen aus:

```
Future.invoke(object, "method", new Object[] {param1,param2});
```

Dieser Aufruf liefert ein Ergebnis vom Typ Result zurück. Die Methode isAvailable() dieser Klasse gibt an, ob das tatsächliche Ergebnis bereits eingetroffen ist. Mit den Methoden readInt(), readByte() etc. und readObject() kann das Resultat dann ausgelesen werden. Wenn diese Methoden aufgerufen werden, solange das Ergebnis noch nicht eingetroffen ist, blockieren diese, bis es eintrifft. Falls der Aufruf eine Ausnahme hervorgerufen hat, wird dies durch die Methode isException() angegeben.

Hierzu ein kleines Beispiel. Ein Timer-Objekt soll einen etwas größeren Zeitaufwand für eine Übertragung einer Nachricht simulieren. Der Timer blockiert eine gewisse Zeit, angegeben in Millisekunden, und liefert dann ein Ergebnis zurück, in diesem Fall schlicht eine Eins.

```
public class Timer implements ITimer {

  public int alarm(int milliseconds) {
    try {
      Thread.sleep(milliseconds);
      System.out.println("ALARM");
    } catch (Exception e) {}
    return 1;
  }
}
```

Bei einem synchronen Aufruf müsste ein aufrufendes Objekt warten, bis das Ergebnis zurückgeliefert wird. Durch die Möglichkeit des asynchronen Aufrufs kann statt dessen etwas Sinnvolles ausgeführt werden.

```
import com.objectspace.voyager.*;
import com.objectspace.voyager.message.*;
public class TimerClient {

  public static void main(String args[]) {
    try {
      Voyager.startup();
      ITimer timer = (ITimer) Factory.create("Timer","//sun:8000");
      Result result = Future.invoke(
        timer,"alarm", new Object[]{new Integer(2000)});
      if (!result.isAvailable()){
        System.out.println("Waiting for result, doing something else");
      }
      int res = result.readInt();
      System.out.println("Result returned");
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

Bei der Ausführung dieses Programms ist zu erkennen, dass die Applikation in der Lage ist, etwas zu tun, während auf das Ergebnis gewartet wird. In diesem Fall wird lediglich ein String ausgegeben (Waiting for result, doing something else), aber in ernsthafteren Anwendungen können so Wartezeiten sinnvoll genutzt werden.

```
sun> voyager 8000

lin> java TimerClient
lin| Waiting for result, doing something else

sun| ALARM

lin| Result returned
```

Wenn lediglich eine Methode auf einem entfernten Rechner angestoßen werden soll, aber das Ergebnis nicht relevant ist oder diese Methode ohnehin kein Ergebnis liefert, bietet Voyager auch die Möglichkeit, einen so genannten »one-way«-Aufruf abzusetzen. Dabei gibt es allerdings keine Rückmeldung darüber, ob der Aufruf angekommen ist, erfolgreich begonnen, beendet oder fehlerhaft abgebrochen wurde. Daher wird diese Art des Aufrufs auch als »fire-and-forget« bezeichnet. Der Aufruf wird, ebenso wie für den asynchronen Fall, über die Methode invoke(), hier allerdings der Klasse OneWay, erzeugt. Es besteht auch eine entsprechende Klasse Sync mit der Methode invoke(). Dies entspricht allerdings dem Standardverhalten, und sie wird daher in den seltensten Fällen benötigt werden.

Nützlicher ist da eine Variante der Methode invoke(), die drei zusätzliche Parameter erlaubt. Der erste von diesen, ein boolescher Schalter, bestimmt, ob ein Ergebnisobjekt als Kopie übertragen oder ein Proxy auf das Originalobjekt zurückgeliefert wird. Mit dem zweiten kann eine zeitliche Begrenzung für das Blockieren der readXxx()-Methode gesetzt werden, so dass zum Beispiel auf das Ausbleiben einer Nachricht reagiert werden kann. Als drittes kann eine Referenz auf einen EventListener übergeben werden, der bei Beendigung des Methodenaufrufs durch einen Callback benachrichtigt wird. Dadurch muss das Ergebnis weder erwartet (readXxx()) noch gepollt (isAvailable()) werden, sondern kann als Ereignis eintreffen.

2.3 Unmittelbare Objektkommunikation

Ein wichtiges Ziel bei der Entwicklung verteilter Programmiersprachen ist die Komplexität der Programmierung von Verteilung möglichst weitgehend zu verbergen. Für das Gebiet der Objektorientierung bedeutet dies idealer Weise eine direkte Kommunikation von Objekten auch über Rechnergrenzen hinweg. Damit wäre eine Transparenz der Verteilung erreicht, die die Entwicklung verteilter Systeme in dieser Hinsicht mit der Entwicklung nicht-verteilter Software gleichstellen würde oder diesem Ziel zumindest ein großes Stück näher rücken würde. Eine der ersten und der bekanntesten Sprachen, in der dieser Ansatz verfolgt

wurde ist Emerald, das im Folgenden stellvertretend für diesen Ansatz vorgestellt werden soll.

Emerald ist eine objektorientierte Sprache, die Ende der achtziger Jahre für die Programmierung verteilter Anwendungen an der Universität Washington entwickelt worden ist [Hutchinson 1987], [Black et al. 1987]. Das Hauptaugenmerk bei der Entwicklung lag auf dem Aspekt der Verteilung. Bezüglich der Objektorientierung, die damals insgesamt noch sehr jung war, weist Emerald einige Besonderheiten auf, die diese Programmiersprache zum Teil etwas merkwürdig erscheinen lassen. So gibt es in Emerald keine Klassen. Statt dessen wird ein Objekt durch die Ausführung eines *Konstruktors* erzeugt. Dieser Konstruktor ähnelt einer Klasse insoweit, als er alle notwendigen Informationen bezüglich eines Objektes besitzt (lokale Daten, Schnittstellendefinition und Quellcode).

Dennoch ist Emerald eine in gewisser Weise sehr reine objektorientierte Sprache. Es basiert auf einem einheitlichen Objektmodell für alle Sprachkonstrukte: Sämtliche Konstrukte und Datentypen werden durch Objekte repräsentiert. Polymorphismus wird in Emerald durch die Verwendung von Typkonformität erreicht. Emerald ist eine streng getypte Sprache, dessen Typsystem auf dem Konzept des *abstrakten Typs* basiert. Der abstrakte Typ definiert hierbei die Schnittstelle eines Objektes. Ein Objekt ist zu einem abstrakten Typ *konform*, wenn es mindestens die Schnittstelle des abstrakten Typs implementiert. Soll ein Objekt durch ein anderes ersetzt werden, so muss das neue Objekt konform zum gegebenen abstrakten Typ sein. In Emerald ist es sogar möglich, zur Laufzeit eine neue Objektimplementation für ein bereits existierendes Objekt hinzuzufügen, solange diese zu dessen abstrakten Typ konform sind. Allerdings wird diese Form des Polymorphismus durch Quellcode-Wiederholungen erkauft, da es in Emerald keine Vererbung gibt und deshalb auch keine Überladung von Eigenschaften.

Eine weitere Besonderheit von Emerald sind die so genannten *Prozesse*. Hierbei handelt es sich um einen Quellcode-Block, der während der Erzeugung eines Objektes asynchron gestartet wird. Die Ausführung findet nebenläufig zu anderen Prozessen statt. Objekte, die über einen Prozess verfügen, werden als *aktiv* bezeichnet. Alle anderen Objekte gelten als *passiv*.

Emerald hat vor allem als *verteilte* objektorientierte Sprache Aufmerksamkeit erregt und soll im Folgenden daher hauptsächlich unter diesem Aspekt vorgestellt werden. Besonders interessant sind dabei die folgenden drei Entwurfsziele:

1. **Netzwerktransparenz**. Es gibt keinen sprachlichen Unterschied zwischen einem lokalen und einem entfernten Methodenaufruf.

2. **Mobilität auf Objektebene**. Objekte können jederzeit über die lokalen Rechnergrenzen hinweg migriert werden. Dies schließt den Fall mit ein, dass sich ein Objekt selbst migriert.

3. **Effizienz**. Zugriffe auf lokale Objekte sollen möglichst direkt und effizient ausgeführt werden, das bedeutet zum Beispiel, dass die Entscheidung, ob ein Aufruf lokal oder entfernt ist (was das Laufzeitsystem entscheiden muss), ohne Netzwerkverkehr stattfinden soll.

In Emerald erfolgen alle Objektzugriffe über Referenzen. Eine Referenz auf ein lokales Objekt unterscheidet sich dabei nicht von einer entfernten Referenz. Somit wird die Verteilung von Objekten transparent. Es ist für einen Zugriff auf ein Objekt unerheblich, ob dieses lokal ist oder entfernt, und kann in beiden Fällen gleich benutzt werden. Dies gilt auch für die Übergabe von Parametern bei Methodenaufrufen. Man übergibt einfach eine Referenz, so wie man es in einem lokalen Programm tun würde.

Es gibt jedoch Situationen, in denen es von Vorteil ist, den Aufenthaltsort von Objekten explizit wählen zu können. Zum Beispiel um die Anzahl entfernter Methodenaufrufe zu minimieren, indem man das entsprechende Zielobjekt zum aufrufenden Objekt in die lokale Umgebung holt. Um dies zu ermöglichen, verfügt Emerald über die Möglichkeit Objekte zu migrieren. Hierfür stehen die folgenden Befehle zur Verfügung:

- **move X to Y** bewegt das Objekt X auf den Rechner, auf dem sich das Objekt Y befindet. Da auch ein Rechner durch ein Objekt repräsentiert wird, kann das Ziel auch ein konkreter Rechner sein.
- **fix X at Y** bewegt das Objekt X auf den Rechner, auf dem sich das Objekt Y befindet. Zusätzlich wird das Objekt X dort allerdings fixiert, wodurch verhindert wird, dass es wieder wegbewegt werden kann.
- **unfix X** gestattet das Bewegen des Objekts X wieder.
- **refix X at Z** hebt die Fixierung des Objekts X auf, bewegt es dahin, wo Z sich befindet, und fixiert es anschließend wieder. Hierbei handelt es sich um eine unteilbare (atomare) Aktion.

Bei dem move-Befehl gilt zu beachten, dass es sich hierbei eher um einen *Vorschlag* handelt. Das System ist dabei nicht verpflichtet, die Migration auszuführen (zum Beispiel, weil das Objekt fixiert ist), und genauso wenig ist das Objekt verpflichtet, am Zielort zu verweilen. Im Gegensatz hierzu sind die restlichen drei Befehle für System und Objekt bindend. Wenn also die Befehle fix bzw. refix erfolgreich ausgeführt wurden, bleibt das Objekt am Zielort, bis es explizit wieder freigegeben wird.

Eine weitere Ausprägung der Mobilität lässt sich im Zusammenhang mit der Parameterübergabe bei Methodenaufrufen finden. Normalerweise werden hierbei die Parameter in Form von Referenzen übergeben (*call-by-reference)*. Unter bestimmten Umständen kann dies aber zu einem erhöhten Kommunikationsaufwand führen. Befinden sich beispielsweise das aufrufende und das aufgerufene Objekt auf verschiedenen Rechnern, so muss das Zielobjekt für jeden Zugriff auf ein Parameterobjekt einen entfernten Methodenaufruf initiieren. Um dies zu vermeiden bzw. einzuschränken, bietet Emerald drei Varianten des klassischen *call-by-reference* an. Hierbei ist es aber dem Programmierer überlassen, ob er davon Gebrauch macht.

Die ersten beiden Varianten betreffen die einzelnen Parameterobjekte. Wird vor einem Parameter das Schlüsselwort move gestellt, so wird bei einem Methodenaufruf das entsprechende Parameterobjekt zum Rechner des aufgerufenen

Objektes bewegt (*call-by-move*). Das gleiche geschieht, wenn man das Schlüsselwort visit verwendet (*call-by-visit*). Allerdings wird in diesem Fall das Parameterobjekt nach Beendigung des Methodenaufrufes zum Aufrufer zurückbewegt. Die dritte Variante wiederum bezieht sich auf den Rückgabewert eines Methodenaufrufes. Stellt man diesem das Schlüsselwort move voran, so wird das Ergebnisobjekt automatisch zum Aufrufenden bewegt (*call-by-move-return*).

Um eine möglichst große Effizienz zu erzielen, gibt es einige zusätzliche Besonderheiten. So können Objekte als unveränderlich gekennzeichnet werden. Dies bietet sich zum Beispiel für Objekte an, die lediglich Funktionen anbieten oder nur konstante Daten enthalten. Dadurch kann das System zur Laufzeit eine Optimierung vornehmen und diese Objekte in die lokale Umgebung kopieren, statt entfernt auf sie zuzugreifen.

Ein wichtiger Aspekt im Zusammenhang mit Objektmigration ist die Frage nach dem Umfang der Migration oder der *Granularität*. Gemeint ist hiermit, ob neben dem eigentlichen Objekt noch weitere Objekte mitmigriert werden sollen. Wird zum Beispiel nur das einzelne Objekt migriert und dabei eventuell aus einer Gruppe von zusammengehörigen Objekten gerissen, so kann dies in einem stark erhöhten Kommunikationsaufkommen resultieren. Indem man Gruppen von Objekten, die zueinander in enger Beziehung stehen, zusammenhält, kann dies umgangen werden.

Um dies zu ermöglichen, muss der Programmierer explizit festlegen, welche Objekte in Bezug auf Mobilität eine Gruppe bilden sollen. Hierfür stellt Emerald das Schlüsselwort attached zur Verfügung, welches einer Objektreferenz vorangestellt wird. Wird ein Objekt migriert, so folgen ihm alle auf diese Art referenzierten Objekte automatisch. Diese Beziehung ist transitiv, aber nicht symmetrisch. Dies bedeutet, dass wenn ein Objekt A mit einer Referenz auf ein Objekt B, die als attached gekennzeichnet ist, bewegt wird, so folgt Objekt B automatisch. Wenn aber Objekt B migriert wird, bewegt sich A nicht mit.

Insgesamt wurden die Netzwerktransparenz und die Objektmigration in Emerald sehr eindrucksvoll demonstriert und in mehreren Sprachen wie Trellis/DOWL [Achauer 1993], [Schaffert 1986] und Beta [Brandt und Madsen 1993] nachgeahmt. Auch für Java existieren Umsetzungen dieses Konzeptes.

Kapitel 3

Nebenläufigkeit

In nicht-verteilten Systemen, wie sie etwa ein PC mit nur einer CPU darstellen, spielte Nebenläufigkeit bis vor einigen Jahren noch eine untergeordnete Rolle. Es gab nur einen Benutzer, der jeweils nur ein Programm zur Zeit startete und die Programme liefen streng sequenziell ab bis sie terminierten. Klassische Programmiersprachen wie Pascal oder C enthielten keinen eigenen Nebenläufigkeitsmechanismus. Erst durch die Einführung von modernen Betriebssystemen die Nebenläufigkeit unterstützen, wurde zum Beispiel die gleichzeitige Bearbeiten zweier Programme möglich. Doch auch heute gibt es eine Vielzahl von Systemen, die keine Nebenläufigkeit kennen. Beispiele hierfür sind etwa Embedded Systems, Handys oder Palmtops.

In verteilten Systemen ist Nebenläufigkeit oft eine natürliche Gegebenheit. Selbst wenn die einzelnen Teilsysteme keine Nebenläufigkeit kennen, wird durch die Kopplung eine mögliche Nebenläufigkeit eingeführt. Andererseits ist Nebenläufigkeit in verteilten Systemen auch auf den einzelnen Systemen eine zusätzliche Notwendigkeit. Sie wird unter anderem benötigt, um die Kommunikation zwischen zwei Teilsystemen zu entkoppeln. Ähnlich wie in interaktiven Systemen die Kommunikation mit dem Benutzer häufig von eigenen Kontrollflüssen behandelt wird, kann damit eine asynchrone Kommunikationsform für entfernte Aufrufe erreicht werden.

Dies bietet den Vorteil einer schnelleren Verarbeitung, bringt aber auch zusätzliche Komplexität mit sich. Die Komplexität liegt nicht (oder nicht mehr) in der Programmierschnittstelle, sondern in der Sache selbst.

- Wenn zwei (oder mehrere) Prozesse wechselseitig auf Ressourcen zugreifen wollen, die der jeweils andere Prozess reserviert hat, kommt es zu einer Verklemmung, auch als *Deadlock* bezeichnet.

- Ein verwandtes Problem sind *Livelocks*. Bei einem Livelock wartet ein Prozess auf ein Ereignis, das nicht eintreffen wird, oder auf eine Ressource, die nie zur Verfügung gestellt wird.

- Wenn eine Ressource zwar vorhanden ist, aber nicht gleichmäßig verteilt wird, spricht man von *Unfairness*. Dies kann durchaus ein gewünschter Effekt sein damit das System insgesamt schneller Fortschritte macht, auch wenn einzelne Prozesse dabei unfair behandelt werden können. Wenn aber

ein Prozess keinen Fortschritt mehr macht, wird dies zum Problem, und man spricht von *Starvation*.

- Das Ergebnis einer Berechnung kann von der Ausführungsreihenfolge verschiedener Prozesse abhängen. Dann spricht man von einer *race condition*.

Nebenläufigkeit wird in der Literatur als Themengebiet oft den Betriebssystemen oder der Rechnerarchitektur zugeordnet und meist auch im Rahmen entsprechender Vorlesungen und Bücher gelehrt. Und bisher war es tatsächlich so, dass die Behandlung von Nebenläufigkeit vor allem auf Ebene des Betriebssystems notwendig war, etwa bei der Verwaltung von Ressourcen, die von mehreren Prozessen genutzt wurden, wie beispielsweise Festspeichermedien oder Drucker. Oder die Nebenläufigkeit trat eng gekoppelt mit den Rechnerarchitekturen auf, die Parallelität unterstützen, zum Beispiel in Parallelrechnern mit mehreren CPUs oder Vektorrechnern mit mehreren Recheneinheiten in einer CPU.

Dies hat sich durch in modernen Programmiersprachen wie Java geändert. Java ist die erste weit verbreitete Sprache, in der Nebenläufigkeit in Form von leichtgewichtigen Prozessen direkt unterstützt wird, und zwar unabhängig von der Hardware und vom Betriebssystem, auf dem die Java Virtual Machine läuft. Damit ist Nebenläufigkeit ein Thema für das Software Engineering geworden. Die Komplexität der Programmierung von Nebenläufigkeit ist damit erheblich reduziert worden. Programme, in denen Nebenläufigkeit benötigt wird, können somit auf der einen Plattform entwickelt und auf einer anderen eingesetzt werden. Der Nebenläufigkeitsmechanismus von Java ist in [Oaks und Wong 1997], [Magee und Kramer 1999] und in [Kredel und Yoshida 1999] beschrieben.

Nebenläufigkeit wird in verschiedenen Kontexten und zur Lösung unterschiedlicher Aufgaben eingesetzt:

- In modernen Betriebssystemen ist die Ausführung mehrerer Programme gleichzeitig auf einem Rechner möglich. Dies geschieht meistens mit jeweils einem schwergewichtigen Prozess pro Programm. Bei der Ausführung mehrerer Java-Programme auf einer Java Virtual Machine werden allerdings leichtgewichtige Prozesse abgespalten.

- Voneinander unabhängige Probleme innerhalb eines Programms können relativ problemlos nebeneinander bearbeitet werden. Zwar erzielt man dabei (bei Verwendung einer CPU) insgesamt keine Beschleunigung, dafür blockieren aber größere Aufgaben nicht die CPU, bis die Aufgabe vollständig gelöst ist. So können etwa gute Textverarbeitungsprogramme gleichzeitig den Druck vorbereiten, die Rechtschreibprüfung durchführen und das Hilfesystem anzeigen.

- Ein Problem, das in mehrere Teilprobleme zerlegt werden kann, wird durch Parallelisierung und Aufteilung auf mehrere CPUs schneller gelöst. Hierfür ist natürlich eine entsprechende Hardware nötig, doch moderne Chiparchitekturen und Betriebssysteme sorgen dafür, dass Mehrprozessorsysteme im Server- und auch im Desktopbereich mehr und mehr eingesetzt werden.

Die Kunst liegt allerdings in der Aufteilung des Problems in Teilprobleme, die meist nicht ganz voneinander unabhängig sind und daher einen erhöhten Koordinationsaufwand erfordern.

- Blockierend wartende Prozesse, die zum Beispiel auf ein von außen kommendes Ereignis warten, können vom Hauptkontrollfluss abgespalten werden, damit dieser weiter arbeiten kann. Dies kommt in verteilten Systemen sehr häufig vor und wird in mehreren Programmbeispielen im Verlaufe des Textes behandelt.
- Ein besonderer Fall, in dem wartende Prozesse sehr häufig auftreten, sind interaktive Programme. Dabei wartet das Programm auf Eingaben vom Benutzer, muss aber andere Tätigkeiten weiterhin durchführen. Interaktive Programme sind ohne die Möglichkeit, Nebenläufigkeit in der Sprache auszudrücken, erheblich komplizierter zu entwickeln.

In diesem Kapitel werden die Konzepte vorgestellt, die heute zur Behandlung von Nebenläufigkeit in Programmiersprachen zur Verfügung stehen oder vorgeschlagen werden. Zunächst wird in Abschnitt 3 auf den Fall eingegangen, in dem die Sprache an sich keine Nebenläufigkeit bietet, sondern vielmehr die Hardware dies zur Verfügung stellt oder das Betriebssystem dies durch (schwergewichtige) Prozesse simuliert. Dann wird der heute übliche Mechanismus der leichtgewichtigen Prozesse oder Threads in Abschnitt 3.2 am Beispiel der Java-Threads diskutiert. Schließlich werden in Abschnitt 3.3 Konzepte diskutiert, die eine Nebenläufigkeit auf Ebene der Objekte oder Komponenten anbieten und eine engere Integration der Nebenläufigkeit in die übrigen Sprachmechanismen zum Ziel haben.

3.1 Nebenläufigkeit durch Hardware oder Prozesse

In verteilten Systemen entsteht Nebenläufigkeit oft schon durch das Vorhandensein von mehreren Recheneinheiten, die unabhängig von einander und parallel zueinander arbeiten oder durch die von modernen Betriebssystemen zur Verfügung gestellten (schwergewichtigen) Prozesse, die auf voneinander getrennten Datenbereichen und nebenläufig zueinander operieren. Auf die Erzeugung dieser Nebenläufigkeit wird hier nicht weiter eingegangen, da sie systemimmanent ist. Vielmehr soll hier auf die Problematik der Synchronisation und Koordination eingegangen werden.

Der Austausch von Daten oder die Synchronisation von Abläufen kann in eng gekoppelten Systemen über einen gemeinsamen Speicherbereich erfolgen. Betriebssysteme bieten Mechanismen, um Datenbereiche vorübergehend zu reservieren oder um Lese/Schreibrechte sinnvoll zu vergeben, so dass eine störungsfreie Koordination möglich ist. In lose gekoppelten Systemen hingegen wird hierfür häufig die entfernte Kommunikation herangezogen. Doch diese erfordert (für die meisten Techniken) eine bilaterale Punkt-zu-Punkt-Kommunikation, in der sich die Kommunkationspartner bekannt sind und auf die schon eingegangen wurde. Im Folgenden soll daher eine anonyme Form des Datenaustausches für lose gekoppelte nebenläufige Systeme vorgestellt werden.

3.1.1 Anonyme Kopplung von Prozessen

Ein Konzept zur anonymen Kopplung von Prozessen sind Tupelräume. In Tupelräumen können Paare von Werten (tuple, triple, quadruple, ...) in einen Speicherraum abgelegt werden und wieder herausgenommen werden. Auf diesen Tupelraum kann aus einem verteilten System heraus zugegriffen werden, so dass ein beliebiger Rechner ein Tupel hineinlegen und ein anderer es wieder herausnehmen kann. Dadurch entsteht ein Kommunikationsmechanismus, der für die Kopplung von Prozessen in verteilten oder parallelen Systemen sehr geeignet ist: Die Kommunikation ist asynchron, anonym, verteilt und persistent.

Das Konzept der Tupelräumen wurde in den 80er Jahren von Nicholas Carriero [Carriero und Gelernter 1989], [Carriero und Gelernter 1990] und David Gelernter [Gelernter 1985] an der Universität Yale entwickelt. Das Projekt, ebenso wie die Sprache, die daraus hervorging, ist als *Linda* bekannt. In Linda können beliebige Tupel von Werten mit einem out-Befehl in einen global sichtbaren Tupelraum abgelegt werden, durch einen read-Befehl gelesen und einen in-Befehl auch wieder herausgenommen werden. Dabei werden Tupel assoziativ adressiert, das heißt, sie werden nicht an einen bestimmten Adressplatz geschrieben, sondern können über ihren Inhalt adressiert werden: Um ein Tupel zu finden, gibt man ein Muster an, das einen Teil der Information des gesuchten Tupels enthält. Das System gibt dann ein zu diesem Muster passendes Tupel zurück. Zum Beispiel kann ein Rechner einen Auftrag erzeugen, etwa von der Form out(berechne, 42, 933), und ein anderer Rechner nach einem Auftrag fragen, in(berechne, *, *); dieser erhält einen dazu passenden Auftrag zurück, den er dann bearbeiten kann. Die Kommunikation, die dadurch entsteht, ist asynchron, weil die Kommunikationspartner nicht zu einer bestimmten Zeit miteinander kommunizieren müssen. Sie ist anonym, das heißt, keiner von den Kommunikationspartnern muss wissen, wer der jeweils andere ist. Sie ist verteilt, da räumlich voneinander getrennte Rechner miteinander kooperieren können, solange beide den Tupelraum erreichen können. Und sie ist persistent, denn die Daten im Tupelraum können länger leben als die Programme, die sie erzeugt haben.

Mit solch einem Kommunikationsmechanismus lassen sich sehr unterschiedliche Kommunikationsmuster realisieren. Wie eben schon angedeutet, lassen sich sehr einfach »Producer-Consumer«-Beziehungen erstellen, wobei der Tupelraum als Auftragspuffer dient. Dabei ist die Anzahl der Producer, ebenso wie die Zahl der Consumer beliebig erweiterbar, ohne dass einer der Beteiligten davon informiert werden müsste. In dieser Form werden Tupelräume sehr gerne in der Parallelverarbeitung eingesetzt. Der Tupelraum kann aber auch als leichtgewichtige Datenbank fungieren, in die ein Gerät oder Programm seine Daten schreibt, ehe es ausgeschaltet wird, und sie wieder holt, wenn es wieder gestartet wird. Es können aber auch Ereignisse auf diese Weise bekannt gemacht werden, so dass der Tupelraum auch als Event-Service dienen kann.

Anbindungen für Linda an Programmiersprachen wurden für eine Reihe von Sprachen wie PASCAL, C und Fortran entwickelt. Doch Linda ist per se nicht objektorientiert. Linda selbst erlaubt nur einfache und daraus zusammengesetzte Datentypen, aber keine Objekte. Eine Übertragung dieses Konzeptes auf die Objektorientierung wurde von Sun vorgeschlagen. Das entstandene System wird

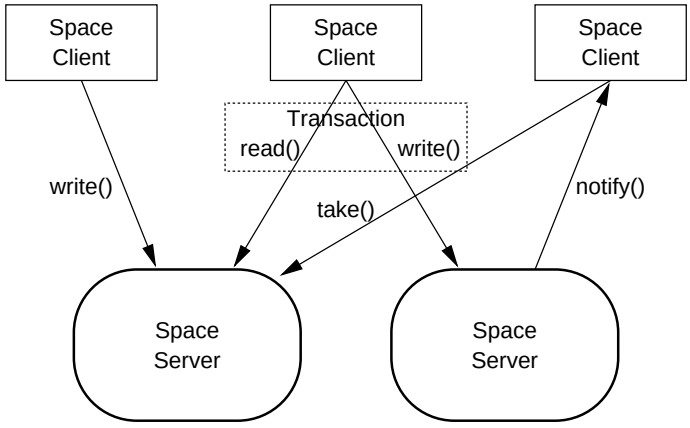
JavaSpaces genannt. Eine Beta-Version der Spezifikation wurde im Sommer 98 veröffentlicht. Inzwischen ist JavaSpaces ein integraler Bestandteil von Jini. Daher sind die Schnittstellen von JavaSpaces im Package net.jini.space zu finden.

Tupel heißen in JavaSpaces Eintrag oder *Entry*. Ein Objekt muss das Interface net.jini.space.Entry implementieren, um in einen Space geschrieben werden zu können, kann aber ansonsten ein ganz gewöhnliches Java-Objekt sein. Die Attribute eines solchen Objekts werden als *Felder* bezeichnet. Als Beispiel soll hier eine Nachricht mit drei Feldern für ein BulletinBoard dienen:

```
public class Message implements net.jini.space.Entry {
  public String subject;
  public String body;
  public int category;

  public Message(String _subject, String _body, int _category) {
    subject = _subject;
    body = _body;
    category = _category;
  }
}
```

Solch ein Objekt kann ganz gewöhnlich erzeugt und anschließend in einen Tupelraum abgelegt werden. Ein Tupelraum wird in JavaSpaces schlicht *Space* genannt und zentral von einem Server verwaltet. Ein Client greift über ein Netzwerk auf diesen Server zu und kann seine Schnittstelle entfernt nutzen. Damit aber der Programmierer möglichst wenig mit der Verteilung zu tun hat, wird ein lokales Objekt erzeugt, das den Server vertritt und dem Client die Schnittstelle lokal anbietet. Die Methoden, die diese Schnittstelle anbietet, entsprechen von ihrer Funktionalität her etwa denen von Linda, wurden aber, um die Lesbarkeit und deren intuitives Verständnis zu erleichtern, umbenannt. Das Schreiben in einen Space heißt write(), das Lesen read() und das Herausnehmen take().

Abbildung 3.1: Operationen auf einen JavaSpace



Beim Schreiben wird die Gültigkeitsdauer eines Eintrags mit angegeben, so dass veraltete Objekte aus dem Space entfernt werden und der Space nicht

durch Altlasten verunreinigt wird. Die Gültigkeit eines Eintrags kann allerdings verlängert werden durch einen Mechanismus, der am Leasing angelehnt ist und genauer im Kapitel über Jini erläutert wird. Als Ergebnis einer Schreiboperation erhält man ein Lease-Objekt, mit dessen Hilfe sich die Gültigkeit verlängern lässt.

```
JavaSpace space = getSpace();
Message message = new Message("VW Golf","gebrauchter Golf, Baujahr ...", 2);

long oneday=24*60*60*1000; // 24 Stunden in Millisec
Lease lease = space.write(message, null, oneday);
```

read() und take() liefern jeweils einen Eintrag aus dem Space zurück, wobei read() einen Eintrag nur kopiert und im Space belässt, während take() den Eintrag auch aus dem Space löscht. Beide Befehle sind blockierend, das heißt, wenn kein passender Eintrag vorhanden ist, warten diese, bis einer vorhanden ist. Daher muss bei beiden durch Angabe eines Timeout in Millisekunden die Wartezeit begrenzt werden. Um nach einem Eintrag zu suchen, wird diesen Operationen ein Eintrag-Muster mitgegeben. Dieses Muster, auch als *Template* bezeichnet, ist vom gleichen Typ wie der gesuchte Eintrag, kann aber Platzhalter enthalten, die nicht spezifiziert sind, also auf null gesetzt sind. Diese Platzhalter werden *Wildcards* genannt.

```
Message msgTemplate = new Message("Biete", null, null);
Message result = space.read(msgTemplate, null, 100);
```

Der Space-Server vergleicht die vorhandenen Einträge feldweise mit dem Template und liefert einen Eintrag zurück, bei dem alle Felder identisch sind, die nicht mit einer Wildcard besetzt sind. In diesem Fall würde also eine Nachricht der Kategorie »Biete« zurückgeliefert werden. Es wird dabei genau ein Eintrag zurückgeliefert. Um an alle Einträge der Kategorie »Biete« heranzukommen, muss ein relativ hoher Aufwand getrieben werden. Ein wiederholtes read() liefert auch nicht den nächsten passenden, sondern wieder irgendeinen passenden Eintrag, den eben erhaltenen eingeschlossen.

Der Server speichert Einträge in einer serialisierten Form ab, allerdings wird nicht das ganze Objekt, sondern die einzelnen Felder des Objekts serialisiert. Daher ist es ihm möglich, auf diese Felder zuzugreifen, ohne vorher das gesamte Objekt deserialisieren zu müssen. Der Vergleich von Einträgen mit Templates erfolgt auf genau diese Weise: Die serialisierten Felder werden mit den serialisierten Feldern des Templates verglichen. Dieser Vergleich erfolgt dabei bitweise. Ist kein passender Eintrag vorhanden, blockiert der Aufruf, bis ein neuer zur Anfrage passender Eintrag eintrifft oder der Timeout abläuft. Es gibt zu beiden Aufrufen eine Variante, die nicht blockiert und readIfExists() respektive takeIfExists() heißt.

Eine weitere interessante Methode ist das notify(), mit dem ein Client sein Interesse an einem neu eingehenden Eintrag, der einem Template entspricht, kundtun kann. Trifft ein solcher Eintrag ein, wird ein mitgegebener Eventhandler

aufgerufen, der dieses Ereignis aufnehmen und geeignete Maßnahmen ergreifen kann.

Alle Methoden von JavaSpaces können transaktional behandelt werden. Dafür gibt man beim Aufruf ein Transaktionsobjekt als Parameter mit an. Soll keine Transaktion verwendet werden, kann man statt dessen null angeben.

3.2 Nebenläufige Kontrollflüsse

Java bietet in der Sprache selbst eine Unterstützung von Nebenläufigkeit mit leichtgewichtigen Prozessen. In den meisten bisher gebräuchlichen Sprachen war dies nur durch direkten Zugriff auf Betriebssystembibliotheken möglich, was den Umgang damit schwierig und Portabilität fast unmöglich machte. Java ermöglicht Nebenläufigkeit nicht nur plattformunabhängig und mit unmittelbarer Sprachunterstützung, sondern auch noch auf eine relativ einfache Art und Weise.

Kern dieses Konzeptes ist die Klasse java.lang.Thread. Diese Klasse erlaubt es, einen leichtgewichtigen Prozess zu starten, zu unterbrechen, wieder aufzuwecken, zu verlangsamen, zu beschleunigen und zu beenden. Eine Handvoll weiterer Klassen, wie das Interface Runnable, helfen bei der Verwendung dieser Klasse. Das Konzept wird durch Methoden unterstützt, die in der Mutter aller Objekte, der Klasse java.lang.Object, definiert sind, sowie durch die zwei Schlüsselwörter synchronized und volatile, die fester Bestandteil des Sprachumfangs sind.

Leichtgewichtige Prozesse werden häufig als Thread (englisch für Faden) bezeichnet, was auf den »Kontrollfaden« hinweisen soll. Diese Bezeichnung ist auch in Java üblich und wird im Folgenden verwandt. Es gibt zwei Möglichkeiten, einen neuen Thread abzuspalten. Der einfachere Weg besteht darin, eine Klasse von der Klasse Thread abzuleiten und eine Instanz dieses Erben zu erzeugen. Dafür muss die erbende Klasse zwingend die Methode run() implementieren. In dieser Methode, die vom System aufgerufen wird, muss festgelegt werden, was in diesem neuen Thread ausgeführt werden soll. Sie stellt die Wurzelmethode dieses Threads dar. Nach der Erzeugung einer Instanz muss die Methode start() aufgerufen werden, die dafür sorgt, dass systemintern ein neuer Thread abgespalten und in diesem die Methode run() gestartet wird. Parameter, die diesem neuen Thread mitgegeben werden sollen, können nicht mittels run(), sondern nur über den Konstruktor des Erben von Thread übergeben werden.

```
class ExampleThread extends Thread {
  int parameter;
  ExampleThread (int param) {
    parameter=param;
  }

  public void run() {
    //was soll der neue Thread ausführen?
    ...
  }
```

```
  public static void main(String[] args) {
    // Instantiieren und Starten des Beispiel-Threads
    ExampleThread t = new ExampleThread(42);
    t.start();
  }
}
```

Da bei diesem Verfahren die Vererbung eingesetzt wird und Java nur Einfachvererbung erlaubt, ist eine zweite Variante vonnöten, die das direkte Erben umgeht. Ein Thread kann auch ohne die Vererbung nutzen zu müssen, erzeugt werden. Doch damit dieser weiß, was er ausführen soll, benötigt er eine Klasse, die eine run()-Methode enthält. Dafür wird eine Klasse geschrieben die das zur Verfügung stehende Interface Runnable implementiert. Dieses Interface kennzeichnet diese Klasse als eine, die die Methode run() enthält, so dass der Thread sie aufrufen kann. Eine Instanz dieser Klasse kann bei der Erzeugung eines Threads an dessen Konstruktor übergeben werden und dieser ruft die Methode run() auf. Auf diese Weise ist es möglich einer Klasse, die von einer beliebigen Klasse erbt, einen Thread zuzuordnen.

```
class ExampleRunnable extends SomeClass implements Runnable {
  int parameter;

  ExampleRunnable (int param) {
    parameter=param;
  }

  public void run() {
    //was soll der neue Thread ausführen?
    ...
  }

  public void main(String[] args) {
    // Instantiieren von Runnable und Starten des Threads
    ExampleRunnable r = new ExampleRunnable(42);
    new Thread(r).start();
  }
}
```

Die Klasse Thread ermöglicht es, auf den Zustand eines solchen Threads zuzugreifen und auf ihn Einfluss zu nehmen. Mit sleep() kann ein Thread für eine festgelegte Zeit verzögert werden. Mit suspend() wird er auf unbestimmte Zeit unterbrochen und kann durch einen Aufruf der Methode resume() wieder zur Aktivität erweckt werden.

3.2.1 Nebenläufigkeit und Verteilung

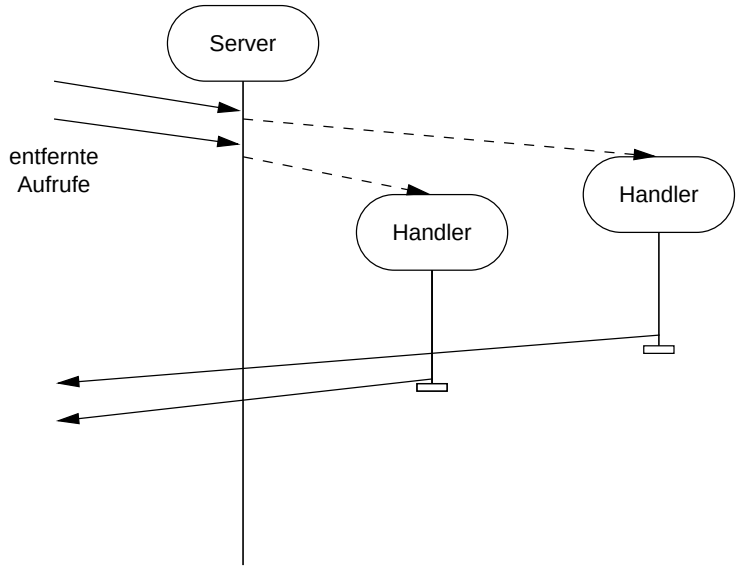
Die Threads, wie sie bisher behandelt wurden, dienen zunächst einmal der Nebenläufigkeit innerhalb eines Programms und auf einem Rechner. Es ist kein

Mechanismus zur Verteilung. Es ist zum Beispiel nicht (ohne weiteres) möglich, einen Thread auf einem anderen Rechner zu erzeugen. Dennoch stellt die Nebenläufigkeit ein sehr wichtiges Element der verteilten Programmierung dar. Es sollen in den folgenden zwei Abschnitten zwei Muster vorgestellt werden, die in der Kommunikation in verteilten Systemen eine große Bedeutung haben. Weitere Muster in der Nebenläufigkeit werden in [Lea 1997] beschrieben.

3.2.2 Server und Handler

Ein Muster, das auf der Serverseite eines verteilten Systems sehr häufig anzutreffen ist, ist das vom *Server und Handler*. Ein Server ist ein Programm, das Dienstleistungen wie zum Beispiel aufwendige Berechnungen oder Datenbankzugriffe für andere Programme anbietet. Ein Server erhält von einem Client einen Auftrag, den dieser dann zu erledigen hat. Da solche Server häufig auf zentralisierten, gut ausgestatteten und teuren Rechnern laufen (die selbst auch als Server bezeichnet werden, doch dies ist eine Bezeichnung auf einer anderen sprachlichen Ebene), sollen sie diese Dienste nicht nur für einen, sondern für eine ganze Reihe von Clients erledigen. Dafür ist eine Architektur notwendig, in der der Server nicht erst eine Aufgabe vollständig löst und anschließend auf neue wartet, sondern wo er erreichbar bleibt und mehrere Aufgaben gleichzeitig entgegennehmen kann.

Abbildung 3.2: Der Server leitet Aufrufe an den Handler weiter

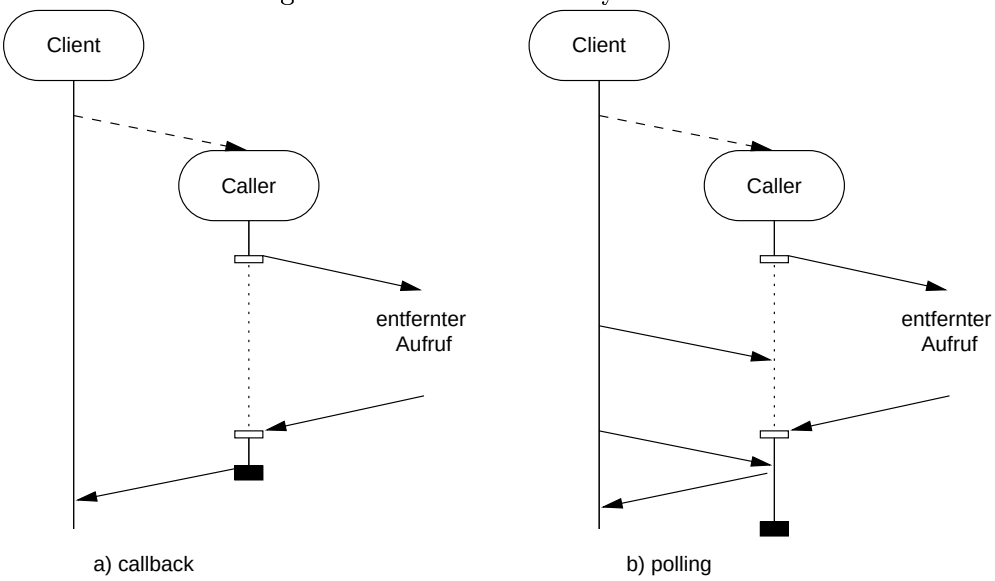


Um dies zu erreichen, werden Threads eingesetzt. Der Server erledigt die Aufgabe gar nicht selbst, sondern delegiert diese an eine andere Klasse weiter, die in einem eigenen Thread läuft. Somit benötigt der Server nur eine sehr kurze Zeit, um eine Aufgabe entgegenzunehmen und weiterzuleiten und ist direkt anschließend wieder erreichbar. Die Klasse, die die eigentliche Arbeit verrichtet, wird als *Handler* bezeichnet.

3.2.3 Asynchrone Aufrufe

Ein Muster, das man dagegen häufig auf der Clientseite vorfindet, ist der asynchrone Aufruf. Bei einem normalen (synchronen) Aufruf muss der Aufrufende warten, bis der Aufgerufene die ihm zugeteilte Aufgabe erledigt hat, und ist so lange blockiert. Dies ist die normale Semantik eines Methodenaufrufs. Doch manchmal soll der Aufrufende seine Aufgaben fortsetzen, während der Aufruf abgearbeitet wird. Dies tritt sehr häufig in parallelen Systemen auf, in denen auf solche Weise die Teilaufgaben eines Problems an verschiedene Prozessoren verteilt werden können. Auch in verteilten Systemen spielt dieses Muster eine sehr große Rolle, da die Kommunikationszeiten an sich schon relativ groß sind, was zu dem Aspekt der Parallelität noch hinzukommt.

Abbildung 3.3: Simulation eines asynchronen Aufrufs



Wie später noch gezeigt wird, gibt es Systeme, die einen asynchronen Aufruf direkt unterstützen. In Java jedoch sind Methodenaufrufe stets synchron und die Asynchronität muss daher durch Nebenläufigkeit hergestellt werden. Dafür wird ein zusätzlicher Thread erzeugt, der einen Methodenaufruf zwar synchron aber nebenläufig zum Hauptaktivitätsstrang ausführen kann, wie Abbildung 3.3 zeigt.

Da der Hauptaktivitätsstrang, der hier von der Klasse Client repräsentiert wird, nicht auf das Ergebnis wartet, bedarf es eines anderen Mechanismus, um das Ergebnis zu erhalten. Hierfür gibt es im Prinzip zwei Möglichkeiten. Einerseits kann der Client das Objekt, das für den asynchronen Aufruf sorgt (der Caller), laufend fragen, ob das Ergebnis schon eingetroffen ist. Wenn dies nicht der Fall ist, kann er sich um andere Dinge kümmern. Ist das Ergebnis aber eingetroffen, kann er es auslesen und verwerten. Dieses Verfahren nennt man *Polling*. Andererseits kann der Caller eine Methode des Client aufrufen, sobald er das Ergebnis erhalten hat. Dies wird als *Callback* bezeichnet.

3.3 Objekt- und komponentenbasierte Nebenläufigkeit

Java ist eine der ersten weit verbreiteten Sprachen, in die Threads direkt integriert sind. Dies hat weithin großen Zuspruch gefunden und vereinfacht die nebenläufige Programmierung erheblich. Auch Mechanismen zur Synchronisation sind mittels eines Monitors für jedes Objekt direkt integriert.

Doch Threads sind ein Basismechanismus, ähnlich wie Sockets ein Basismechanismus für die entfernte Kommunikation sind. Bei Kommunikationsmechanismen ist man auf dieser Ebene nicht stehen geblieben, sondern hat nach Möglichkeiten gesucht, diesen Mechanismus geeigneter in das objektorientierte Paradigma zu integrieren, und hat Mechanismen wie RPC, RMI und sogar Unified Objects entwickelt. Auch für die Nebenläufigkeit wäre ein Konstrukt auf einer höheren Abstraktionsebene wünschenswert.

In seinem Buch *Object-Oriented Software Construction* ([Meyer 1997], S. 951-1036) diskutiert Bertrand Meyer eindrucksvoll die Möglichkeiten, aber auch die Einschränkungen, Nebenläufigkeit und Objektorientierung zu vereinen. Er beginnt seine Ausführungen mit einer Analyse bisheriger Ansätze.

Ein häufig verwendeter Ansatz basiert auf der Vorstellung des *Prozesses*. Diese werden häufig für die Programmierung von nicht-objektorientierter Nebenläufigkeit verwendet. Ein Prozess ist eine Programmeinheit, die einen bestimmten Algorithmus ausführt und diesen normalerweise wiederholt, bis ihn ein externes Ereignis beendet. Prozesse bauen auf autonomen, gekapselten Modulen auf, speichern Werte bis zum nächsten Aufruf, bieten eine klar definierte Schnittstelle an und verwenden für die Kommunikation einen Mechanismus, welchen man allgemein als *message passing* betiteln könnte. Damit sind sie Objekten an sich sehr ähnlich. Was sie von Objekten unterscheidet ist, dass sie *aktiv* sind: Sie haben einen eigenen Kontrollfluss und durchlaufen eine Schleife, in der sie eine gewisse Funktionalität bereitstellen. Dadurch bieten sie die Möglichkeit der Nebenläufigkeit.

Objekte dagegen sind *passiv*: Sie warten bis jemand an ihnen eine Methode aufruft und ihnen einen Kontrollfluss übergibt. Nach entsprechender Ausführung geben sie den Kontrollfluss zurück und wechseln wieder in den wartenden Zustand. Eine viel verfolgte Idee ist nun, die Objekte den Prozessen gleich zu machen, indem man ihnen einen eigenen Kontrollfluss gibt, man spricht dann von *aktiven Objekten*.

Dabei stößt man allerdings auf Probleme im Zusammenhang mit der Vererbung, die ein fundamentales Charakteristikum der Objektorientierung ist. Im allgemeinen werden bei erbenden Klassen neue Methoden hinzugefügt. Diese neuen Methoden müssten selbstverständlich in der Definition des Prozessteils und bei der Synchronisation berücksichtigt werden, was eine Reorganisation des gesamten Moduls nach sich ziehen kann. Man spricht hier von der Vererbungsanomalie. In vielen Projekten, die diesen Ansatz verfolgen, wurde daher das Konzept der Vererbung (und damit die Objektorientierung an sich) in Frage gestellt. Doch Meyer argumentiert, dass nicht die Objektorientierung das Problem darstellt, sondern die aktiven Objekte.

Er schlägt ein anderes Konzept vor, das die Probleme der Vererbungsanomalie löst, gut mit objektorientierten Konzepten harmoniert und zudem in seiner Einfachheit besticht. Seinen Vorschlag basiert auf der Programmiersprache *Eiffel*, die in vielen wichtigen Aspekten Java sehr ähnlich ist.

Dieser Vorschlag, der als *SCOOP* (simple concurrent object-oriented programming) bekannt ist [Meyer 1990], [Meyer 1997], scheint einerseits nicht nur für Eiffel, sondern auch für andere Sprachen geeignet zu sein, andererseits – und hier viel wichtiger – scheint er auch ein für die Verteilung sehr passendes Konzept zu sein. Doch zunächst soll dieses Konzept vorgestellt werden.

3.3.1 Nebenläufigkeit durch Prozessoren

Das Ziel von Nebenläufigkeit ist es, verschiedene Berechnungen gleichzeitig ausführen zu können. Für die Ausführung einer (sequentiellen) Berechnung benötigt man eine einzelne CPU (die häufig auch als Prozessor bezeichnet wird, doch dieser Begriff soll in dieser Arbeit für ein höheres Prinzip aufgespart werden). Möchte man aber mehrere Berechnungen gleichzeitig (echt parallel) ausführen, so benötigt man dementsprechend mehrere CPUs. Meyer setzt sein Konzept der Nebenläufigkeit auf diesem Bild auf und überträgt es auf die Objektorientierung: Statt sich an die Idee eines *Prozesses* zu halten, von denen die aktiven Objekte mit ihrem jeweiligen Kontrollfluss abgeleitet wurden, orientiert er sich an der Idee, dass eine eigenständige Ausführungseinheit (ähnlich einer CPU) passive Objekte verwaltet und deren Ausführung übernehmen kann. Er nennt diese Ausführungseinheit *Prozessor* und definiert diesen Begriff folgendermaßen:

> Ein Prozessor ist ein autonomer Kontrollfluss, welcher in der Lage ist, die sequentielle Abarbeitung von Befehlen an einem oder mehreren Objekten auszuführen.

Ein nebenläufiges objektorientiertes System kann aus einer beliebigen Anzahl solcher Prozessoren bestehen, in einem sequentiellen Systemen wird nur ein Prozessor benötigt. Dieser Prozessor darf *nicht* mit einem physikalischen Prozessor (CPU) verwechselt werden. Vielmehr kann ein Prozessor, wie er eben vorgestellt wurde, auf unterschiedliche Art und Weise umgesetzt werden:

- durch einen *Computer* (mit seiner CPU) im Netzwerk,
- einen *Prozess*, wie er von einigen Betriebssystemen unterstützt wird (z.B. Unix, Windows etc.),
- eine *Koroutine* (simuliert Nebenläufigkeit durch abwechselnde/rotierende Nutzung der CPU),
- einen *Thread*, wie er von so genannten multi-threaded Betriebsystemen unterstützt wird (z.B. Solaris, Windows NT etc.).

Zu den Aufgaben eines Prozessors gehört die Ausführung von Berechnungen. Eine Berechnung findet statt, wenn ein Prozessor eine bestimmte Aktion an einem bestimmten Objekt ausführt. Hieraus wird deutlich, dass jeder Methodenaufruf

an einem Objekt von einem Prozessor ausgeführt werden muss. Um dies tun zu können, wird der Prozessor zu einer Art Verwalter von Objekten.

Bei der Erzeugung eines Objekts wird diesem automatisch ein Prozessor zugewiesen, bei dem es für den Rest seiner Lebenszeit bleibt. Dieser Prozessor ist dann für die Ausführung von Methodenaufrufen an diesem Objekt verantwortlich.

Um Nebenläufigkeit hervorzurufen, werden mehrere Prozessoren eingesetzt. Damit ist die Grundlage für einen Nebenläufigkeitsmechanismus gelegt. Doch erst durch die Verwendung von Asynchronität ist es wirklich möglich, einen Vorteil daraus zu ziehen. Durch synchrone Aufrufe bleibt es bei einem einzigen Kontrollfluss, der sich nur über mehrere Prozessoren erstreckt, aber keine Nebenläufigkeit erzielt. Dem steht der asynchrone Aufruf gegenüber. Hier setzt das aufrufende Objekt seine Berechnung fort, sobald es den Methodenaufruf an das Zielobjekt abgesetzt hat. Dafür müssen allerdings die beiden Objekte sich in unterschiedlichen Prozessoren befinden. Meyer integriert diese beiden Aufrufformen folgendermaßen in sein Konzept:

- Werden zwei Objekte O1 und O2 vom selben Prozessor verwaltet, so findet ein Methodenaufruf von O1 an O2 *synchron* statt. Dies bedeutet, dass das Objekt O1 mit der Ausführung weiterer Berechnungen warten muss, bis der Methodenaufruf am Objekt O2 beendet ist.

- Werden zwei Objekte O1 und O2 von zwei verschiedenen Prozessoren verwaltet, so wird ein Methodenaufruf von O1 an O2 *asynchron* ausgeführt. Das Objekt O1 kann also mit der Berechnung fortfahren, sobald es den Methodenaufruf am Objekt O2 initiiert hat.

Damit sich dieser semantische Unterschied der Aufrufe im Quellcode widerspiegelt, muss die Programmiersprache geändert werden. Für die Programmiersprache *Eiffel* wird dies durch die Einführung des Schlüsselwortes separate gelöst. Eine Deklaration der Form

```
x : SomeType
```

entspricht der üblichen synchronen Semantik. Wird statt dessen die folgende Deklaration verwendet,

```
x : separate SomeType
```

so soll ausgedrückt werden, dass sich das Objekt in einem anderen Prozessor befindet und Aufrufe von Methoden am Objekt x asynchron stattfinden.

Aufgrund der unterschiedlichen Aufrufsemantiken für separate und non-separate Objekte muss gewährleistet sein, dass einer Referenz, die auf ein non-separate Objekt verweist, niemals ein separate-Objekt zugewiesen werden kann. Eine Referenz, die fälschlicherweise als non-separate deklariert ist, aber auf ein Objekt in einem anderen Prozessor verweist, bezeichnet Meyer als *Verräter* (traitor). Um das Entstehen solcher *Verräter* zu verhindern, wurden von ihm vier Konsistenzregeln aufgestellt, die vom Programmierer eingehalten werden müssen und sich folgendermaßen zusammenfassen lassen: Bei Zuweisungen muss das

Ziel als separate deklariert sein, wenn die Quelle es ist; wird eine separate Referenz als Parameter übergeben, muss auch die Parametervariable der Schnittstelle so deklariert sein; separate Rückgabewerte dürfen nur separaten Variablen zugewiesen werden.

Durch einen asynchronen Aufruf ist es möglich, nach der Initiierung eines Methodenaufrufs mit der Berechnung fortzufahren. Um das Ergebnis des Aufrufs zu erhalten, schlägt Meyer das Konzept *wait-by-necessity* (zuerst beschrieben in [Caromel 1989]) vor. Ziel dieses Konzepts ist es, den Programmierer von der Last zu befreien, die Ergebnisse eines asynchronen Aufrufs abholen zu müssen. Dabei wird angenommen, dass das Ergebnis eines Methodenaufrufs im allgemeinen nicht sofort benötigt wird. An dem Punkt, an dem das Ergebnis eines Methodenaufrufs konkret gebraucht wird, etwa bei einer Zuweisung oder einem Aufruf auf diesem Objekt, wird automatisch gewartet.

3.3.2 Nebenläufigkeit und Synchronisation

Jede nebenläufige Sprache, objektorientiert oder nicht, muss Mechanismen anbieten, mit denen sich nebenläufige Ausführungen synchronisieren lassen, also zeitliche Abhängigkeiten zwischen diesen Ausführungen definiert werden.

Meyer analysiert verschiedene (zum Teil nicht-objektorientierte) Synchronisationslösungen. Diese unterteilt er in synchronitäts- und kommunikationsbasierte Verfahren. Zu den ersteren gehören z.B. Semaphore, kritische Bereiche, Monitore und Pfadausdrücke (*path expressions*). Sind die Synchronitätsprobleme gelöst, bleibt trotzdem die Frage, wie die nebenläufigen Einheiten miteinander kommunizieren sollen. Hat man jedoch einen guten Kommunikationsmechanismus entwickelt, kann dadurch gleichzeitig das Synchronitätsproblem mitgelöst werden, denn Kommunikation impliziert Synchronisation. Falls also der Kommunikationsmechanismus flexibel genug ist, bietet er automatisch die gesamte benötigte Synchronisation.

Kommunikationsbasierte Verfahren wurden erstmals von Hoare in CSP (*Communicating Sequential Processes*) vorgestellt. Die Kommunikation findet bei diesem Verfahren mit Hilfe so genannter Kanäle statt, über die Informationen ausgetauscht werden. Betrachtet man vor diesem Hintergrund die grundlegende Vorgehensweise der Objektorientierung, nämlich eine Methode, gegebenenfalls mit Parametern, an einem Objekt aufzurufen, so stellt sich heraus, dass dies ein kommunikationsbasiertes Verfahren ist. Und daraus schließt Meyer, dass die Synchronisation sich direkt in die Objektorientierung integrieren ließe.

Danach findet die Synchronisation durch den Methodenaufruf und die Parameterübergabe statt. Um sich den exklusiven Zugriff auf ein als separate deklariertes Objekt zu sichern, muss man dieses als Parameter bei einem Methodenaufruf verwenden. Die Methode wird erst dann ausgeführt, wenn das separate Objekt verfügbar ist, also durch niemand anderen reserviert wurde. Dies hat zur Folge, dass separate Objekte somit zur ausschließlichen Nutzung zur Verfügung stehen. Die Synchronisation mit der Ergebnisübergabe erfolgt genau dann, wenn das Ergebnis weiter verwendet wird, ein Mechanismus, der, wie schon erwähnt, als *wait-by-necessity* bezeichnet wird.

Unter gewissen Umständen sollte es, laut Meyer, auch möglich sein, die Me-

thodenausführung zu unterbrechen (*interrupt*). In dem Fall wird eine Ausnahme (*exception*) erzeugt, so dass das Objekt, welches den Methodenaufruf ursprünglich initiiert hat, dies auch mitbekommt und entsprechende Korrekturmaßnahmen, wie zum Beispiel eine Wiederholung zu einem späteren Zeitpunkt, einleiten kann.

3.3.3 Verteilung

Dieser relativ einfache Mechanismus ist gut für die Nebenläufigkeit geeignet. Aber auch für verteilte Systeme lässt sich dieser Mechanismus einsetzen und Meyer deutet Gedanken dazu auch an. Doch er zielt dabei eher auf eng gekoppelte verteilte Systeme, wie etwa Cluster. Meyer erwähnt zwar die Möglichkeit, Prozessoren zu migrieren, äußert sich aber nicht genauer dazu, wie ein entsprechender Mechanismus aussehen könnte. Doch durch die Einschränkung, dass ein Objekt während seiner gesamten Lebenszeit bei seinem Prozessor bleibt, besteht die Möglichkeit der Migration einzelner Objekte nicht. Objekte sollten stets mit ihrem Prozessor zusammen migriert werden.

Insgesamt bleibt das Konzept, so wie es von Meyer vorgestellt wird, eher ein Nebenläufigkeitskonzept, das nur bedingt für die Verteilung in eng gekoppelten Systemen mit einer relativ statischen Verteilung geeignet ist. Meyer hat am Beispiel der Programmiersprache *Eiffel* gezeigt, wie diese beiden Konzepte im Rahmen der Objektorientierung umgesetzt werden könnten. Eine konkrete Implementierung kann er aber leider noch nicht vorweisen. Doch dieser Mechanismus könnte ein Schlüssel zu einem verteilten Java sein.

Kapitel 4

Persistenz

Persistenz ist die nachhaltige Speicherung von Daten über die Laufzeit eines Programmes hinaus. Sie ist natürlich nicht nur in verteilten Systemen wichtig, doch sie bekommt hier eine besondere Bedeutung. Während in nicht-verteilten Systemen häufig einfache Mechanismen ausreichen – zum Beispiel können die Daten eines Textverarbeitungsprogramms einfach in eine Datei geschrieben werden – sind in verteilten Systemen oft höherwertige Mechanismen von Nöten. Häufig wird in verteilten Systemen von mehreren Benutzern oder Teilsystemen auf gemeinsame Daten, zum Teil sogar gleichzeitig und konkurrierend, zugegriffen. Dadurch werden Mechanismen zur Sicherung eines transaktionalen Verhaltens, zur Gewährleistung eines hohen Durchsatzes oder zur Absicherung bei Ausfällen nötig.

Persistenz eignet sich ebenfalls für die Koppelung verschiedener Anwendungen und Hardwaresysteme. Da Datenbanken schon sehr lange im Einsatz sind und viele Altsysteme ihre Daten auf diese Weise verwalten, stellen sie auch eine Koppelungsmöglichkeit zwischen neuen und alten Systemen dar. Datenbanken bieten ebenfalls die Möglichkeit des entfernten Zugriffes und somit auch eine entfernte Datenkommunikation. Der entfernte Zugriff auf Datenbanken ist vielleicht sogar der am meisten eingesetzte Mechanismus zur verteilten Kommunikation.

In diesem Kapitel werden die heute zur Verfügung stehenden Konzepte für die Persistenz aufgeführt. Zunächst wird in Abschnitt 4.1 auf einfache dateibasierte Mechanismen eingegangen, die die Daten – wiederum aus Sicht des Mechanismus – unstrukturiert als sequentiellen Datenstrom ablegen. Neue Entwicklungen wie XML verschaffen diesen Konzepten eine neue Aktualität. Im Abschnitt 4.2 werden dann Konzepte diskutiert, die für die Speicherung die Struktur der Daten ausnutzen. Sie werden durch relationale Datenbanken implementiert, denen in der Praxis eine sehr große Bedeutung zu kommt. Doch auch hier existiert das Bestreben, die Konzepte für die Persistenz näher an die Konzepte der Programmiersprachen heranzubringen, was seinen Ausdruck in objektorientierten Persistenzmechanismen findet, die im Abschnitt 4.3 vorgestellt werden.

4.1 Unstrukturierte Persistenz

Der grundlegendste Mechanismus, Daten persistent zu machen, ist, sie in eine serialisierten Form zu einem Datenstrom zu wandeln, der als solches in eine Datei abgelegt und ebenso wieder eingelesen werden kann. Klassische Anwendungen aus dem Einzelarbeitsplatzbereich wie Textverarbeitung, Tabellenkalkulation oder Bildbearbeitung etc. basieren auf diesem Mechanismus. Dieses Konzept ist so grundlegend, dass das Filesystem in Betriebssystemen eine zentrale Stellung einnimmt. Im Betriebssystem Unix beispielsweise stellt das Konzept File eine fundamentale Abstraktion dar, die nicht nur zur Speicherung von Anwendungsdaten, sondern auch für die Interaktion mit Peripheriegeräten, etwa der Maus, CD- oder Ziplaufwerken, benutzt wird. Dieses Konzept findet aber auch in verteilten Systemen breite Anwendung. Verteilte Systeme können über moderne Filesysteme wie dem Network File System (NFS) von der Firma Sun in einer netzwerktransparenten Art auf gemeinsam genutzte verteilte Daten zugreifen.

Für die Speicherung von Daten in einer Datei werden die relevanten Daten von der Anwendung in einen linearen Datenstrom umgewandelt und sequenziell auf die Festplatte geschrieben. Beim Laden muss diese Sequenz wieder in die anwendungspezifischen Daten umgewandelt werden.

Diese Datentransformation ist anwendungsabhängig und das Format der erzeugten Datensequenz ist daher häufig anwendungspezifisch und proprietär. Die Transformation selbst muss dabei aufwendig und für jedes neue Programm, ja bei jeder Erweiterung des anwendungsspezifischen Datenschemas, neu programmiert werden.

Eine Alternative hierzu ist die unmittelbare Serialisierung der Objektstruktur. Dabei wird von einem Wurzelobjekt ausgehend der erreichbare Objektbaum automatisch in eine sequentielle Form gebracht und kann so abgespeichert werden. Beim Laden wird, ebenfalls automatisch, der Objektbaum wieder identisch aufgebaut. Java bietet seit der Version 1.1 solch eine Serialisierung. Vorteil ist, dass sich dieser Mechanismus sehr einfach, schnell und unproblematisch implementieren lässt. Erkauft wird dies mit einer verhältnismäßig langen Laufzeit und einer eingeschränkten Einsetzbarkeit der Dateien. Der Algorithmus ist, im Gegensatz zu einer auf eine Anwendung spezialisierten Umsetzung, langsam. Der erzeugte Datenstrom ist für Menschen und andere Anwendungen nicht oder nur kaum lesbar und dient daher nur für das unmittelbare Laden und Speichern oder den Austausch eng integrierter Anwendungen.

Ein Ansatz, der die Nachteile proprietärer Speicherformate als auch der Serialisierung vermeiden kann und ebenfalls zu den dateibasierten Persistenzmechanismen zu rechnen ist, wird derzeit von W3C-Consortium entwickelt und standardisiert und ist als XML bekannt. XML steht für eXtensible Markup Language, eine Tag-basierte, erweiterbare Markierungssprache.

In einer XML-Datei werden einzelne Elemente durch Tags markiert und können so sowohl von menschlichen als auch von maschinellen Lesern erkannt und interpretiert werden. Die Tags können für einen Anwendungsfall spezifisch entwickelt werden; die Bedeutung der Tags und ihre Beziehung untereinander wird in einem zusätzlichen Dokument, einer Document Type Definition oder DTD, festgelegt, die eine Grammatik in einer ähnlichen Weise wie die bekannte

Bacchus-Naur-Form beschreibt. Dadurch entsteht eine jeweils anwendungsspezifische Sprache, mit der die Daten einer Anwendung in einer Textdatei sequentiell beschrieben werden.

Der Vorteil gegenüber herkömmlichen Speicherformaten liegt darin, dass die Dokumente mit einer festgelegten Sprache zusammenhängen und dadurch selbst-beschreibend werden. Daher können sie nicht nur von den erzeugenden Anwendungen, sondern von beliebigen Programmen gelesen und weiterverarbeitet werden. Und auch die Entwicklung von entsprechenden lesenden und schreibenden Komponenten wird erheblich vereinfacht. Wegen der schematischen und wohldefinierten Struktur, die all diesen Datenbeschreibungssprachen gemein ist, lassen sich große Teile der nötigen Verarbeitungssoftware wiederverwenden oder automatisch erzeugen. Es stehen eine Reihe von Werkzeugen für verschiedene Programmiersprachen bereit, die zur Verarbeitung oder Erzeugung von XML-Dokumenten eingesetzt werden können.

Als Beispiel sei hier auf Werkzeuge hingewiesen, die zum Einlesen von Dateien nötig sind und als Parser bezeichnet werden. Da in XML-Dateien die Strukturinformation über die Daten in der Datei mit enthalten sind, lassen sich für XML-Dateien Parser entwickeln, die allgemein und von der Anwendung unabhängig einsetzen lassen. Dabei bestehen zwei prinzipiell unterschiedliche Vorgehensweisen. Die Tag-Struktur eines XML-Dokumentes stellt im wesentlichen eine Baum-Struktur dar. Diese lässt sich einerseits als Ganzes einlesen und als Objektbaum in einer objektorientierten Sprache im Hauptspeicher einer Verarbeitungseinheit halten. Die Erzeugung eines solchen Baumes und eine dazugehörende API ist standardisiert und wird als DOM (Document Object Model) bezeichnet. Vorteil dieses Verfahrens ist, dass der im Hauptspeicher vorliegende Baum mit weitgehenden Operationen umgeformt, erweitert, analysiert oder anders verarbeitet werden kann. Nachteil ist, dass dieser Ansatz viel Speicher verbraucht und verhältnismäßig langsam ist. Der zweite Ansatz wird als SAX (Simple API for XML) bezeichnet und beruht darauf, den Baum nicht einzulesen, sondern lediglich für jedes gefundene Blatt ein Ereignis zu senden, das interessierte Anwendungen interpretieren können. Dieser Ansatz ist erheblich schneller und verbraucht nur wenig Speicherplatz, da aber der Baum nicht vollständig vorliegt, lässt er sich nicht so umfassend bearbeiten wie mittels DOM. Für beide Ansätze stehen Implementierung in verschiedenen Sprachen zur Verfügung.

Eine weitere wichtige Kategorie von Werkzeugen stellen Transformatoren dar, die XML-Dateien in andere Formate, typischerweise ebenfalls XML-basiert, umwandeln können. Dies kann zum Beispiel für die Umwandlung von XML in HTML genutzt werden, so dass ein Datensatz, der unabhängig von seiner Darstellungsform gespeichert oder übermittelt wurde, in ein Format umgewandelt werden kann, das sich zur Darstellung in Webbrowsern eignet.

XML zeigt deutlich, dass zwischen der Persistenz und der entfernten Kommunikation sehr enge Bezüge bestehen. Während XML einerseits als dateibasierter Persistenzmechanismus klassifiziert werden kann, dient es andererseits auch zur Übermittlung von Daten. Dadurch, dass die Struktur der Daten in den Datenstrom selbst einkodiert ist, kann sowohl zur Speicherung als auch zur Übertragung ein einfacher Mechanismus eingesetzt werden, der den Daten-

strom als unstrukturiert behandeln kann, also die unstrukturierte Persistenz bzw. die unstrukturierte Datenkommunikation. Dies erhöht zwar den Umfang der zu übertragenden oder zu speichernden Daten, erleichtert aber die Entwicklung entsprechender Mechanismen soweit, dass diese allgemeingültig werden und anwendungsunabhängig standardisiert werden können.

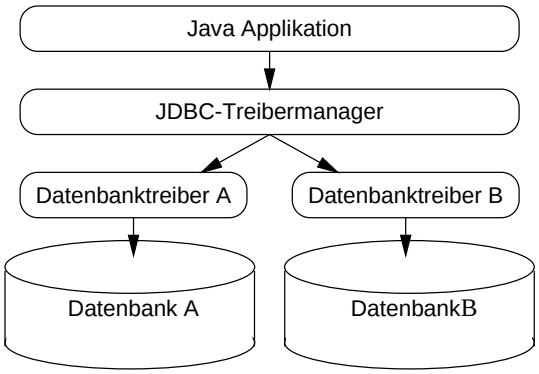
4.2 Datenstrukturbasierte Persistenz

Daten weisen häufig eine inhärente interne Struktur auf, die insbesondere für die Persistenz gut genutzt werden kann. Aufbauend auf der von Codd [Codd 1970] zuerst beschriebenen Relationentheorie, lassen sich solche Daten gut durch Tabellen und Relationen zwischen diesen beschreiben. Datenbanken, die diese Strukturinformation ausnutzen, werden als relationale Datenbanken bezeichnet und bieten insbesondere sehr effiziente Zugriffsmöglichkeiten.

In diesem Abschnitt soll die Anbindung von Anwendungen an relationale Datenbanken vorgestellt werden. Für die Sprache Java besteht hierfür ein Standard, die Java Database Connectivity (JDBC) [Reese 1997], [Hamilton et al. 1997], mit der sich auf fast jede relationale Datenbank zugreifen lässt. Zunächst wird die generelle Struktur von JDBC vorgestellt und auf die wichtigsten Schnittstellen genauer eingegangen.

Eine Anwendung kann gleichzeitig zu mehreren Datenbanken eine Verbindung aufbauen. Um dies zu ermöglichen, wurden zwei Schichten in das JDBC eingebaut, die bei der Verwaltung dieser Verbindungen behilflich sind und den Zugriff möglichst einfach machen. Für jeden Typ von Datenbank muss ein Datenbanktreiber vorhanden sein, der die direkte Kommunikation mit der Datenbank übernimmt. Für die Verwaltung dieser Treiber sorgt der Treibermanager, bei dem die Treiber einmal angemeldet werden und schließlich automatisch verwaltet werden. Diese generelle Struktur von JDBC wird in Abbildung 4.1 dargestellt.

Abbildung 4.1: Die Struktur von JDBC



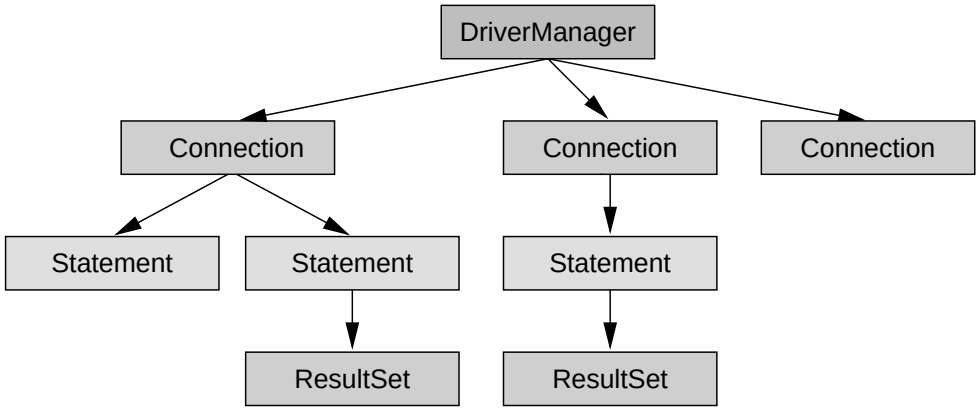
Der Treibermanager stellt die Verbindungen zu den Datenbanken her. Er ist das Bindeglied zwischen der Anwendung und den Datenbanken. Mittels DriverManager.registerDriver() registrieren die Treiber sich selbständig bei dieser Klasse, sobald sie aktiviert wurden, und melden sich per unregisterDriver() wieder

ab.

Datenbanktreiber gibt es in verschiedenen Ausprägungen, die sich vor allem durch die Art der Kommunikation auf Protokollebene unterscheiden. Ein Treiber kann auf die bestehenden Zugriffsprotokolle von ODBC zurückgreifen, proprietäre Datenbankprotokolle verwenden oder für JDBC standardisierte Protokolle fahren.

Der Ablauf einer Datenbankanwendung mit Hilfe von JDBC sieht für eine normale Folge von Abfragen folgendermaßen aus: Die zentrale Klasse von JDBC ist der DriverManager. Dieser stellt eine oder mehrere Verbindungen zu je einer Datenbank her (die Connection). Über die Connection werden einzelne Anfragen (Statement) erzeugt und an die Datenbank übergeben, die einen Satz von Ergebniszeilen, das ResultSet, zurückliefert. Die Ergebniszeilen werden von der Anwendung nacheinander ausgelesen und weiterverarbeitet. Innerhalb einer Connection können mehrere Statements im Rahmen einer Transaktion durchgeführt werden und mit einem commit() bestätigt oder mit einem rollback() verworfen werden. Schließlich wird die Connection wieder geschlossen. Die Beziehung dieser Klassen ist in Abbildung 4.2 gezeigt.

Abbildung 4.2: Der Treibermanager verwaltet mehrere Verbindungen



Eine der wichtigsten Funktionen des Treibermanagers ist die Erstellung und Verwaltung von Verbindungen zu der oder den Datenbanken. Diese Verbindungen werden jeweils von einem Objekt der Klasse Connection verwaltet. Zum Aufbau solch einer Verbindung wird dem Treibermanager eine URL einer Datenbank, ein Benutzername und ein Passwort übergeben, der daraufhin – wenn möglich – eine Verbindung zurückliefert.

```
// GET CONNECTION
Connection con;
try{
   con = DriverManager.getConnection(
      "jdbc:odbc://sun.informatik.uni-hamburg.de/BulletinBoardDb",
       userName,password);
}catch(Exception e){
   System.out.println(e);
}
```

Nachdem eine Verbindung aufgebaut wurde, können über diese Anfragen an die Datenbank gestellt werden. Bei der Entwicklung von JDBC wurde viel Wert darauf gelegt, dass einfache Anfragen einfach gelöst werden können, während für komplexere Anfragen auch eine komplexere Schnittstelle in Kauf genommen werden kann. Daher werden drei Sorten von Anfragen unterschieden. Für gewöhnliche SQL-SELECT-Anfragen steht dem Programmierer eine sehr einfach zu benutzende Form zur Verfügung, das Statement. Für kompliziertere Anfragen stehen die Klassen PreparedStatement und CallableStatement zur Verfügung. Der Zugriff auf Metadaten einer Datenbank, etwa die Struktur der Tabellen, wird durch eine Klasse DatabaseMetaData ermöglicht.

Um eine normale SQL-Anfrage zu stellen, wird ein Objekt der Klasse Statement erzeugt und mit dem Aufruf executeQuery(), der als Parameter einen SQL-Anfragestring erhält, an die Datenbank übermittelt. Die Datenbank antwortet mit einem Objekt der Klasse ResultSet, das eine Menge von Ergebniszeilen enthält, die einzeln durchlaufen und ausgelesen werden können.

```
Statement stmt = con.create();
String query="SELECT * FROM BlackBoard";
ResultSet results=stmt.executeQuery(query);
```

Für Datenbankoperationen wie UPDATE, INSERT und DELETE stellt Statement die Methode executeUpdate() zur Verfügung, die einen booleschen Wert zurückliefert.

```
String insert="INSERT INTO BlackBoard VALUES ('Eine neue wichtige
      Nachricht', 'Marko')";
Boolean bool=stmt.executeUpdate(insert);
```

Die Klasse PreparedStatement ist eine Erweiterung von Statement, die für Anfragen, die sich häufig in ähnlicher Form wiederholen, vorgesehen ist und eine Leistungsoptimierung ermöglicht. Mittels PreparedStatements lässt sich eine Anfrage mit Platzhaltern an die Datenbank übergeben, die diese Anfrage vorkompiliert. Die Platzhalter werden mit einer set-Anweisung gesetzt und mit einem executeQuery()-Aufruf nachgereicht.

```
PreparedStatement prepstmt = con.prepareStatement(
  SELECT * FROM BlackBoard WHERE user = ?);
prepstmt.setString(1,Marko);
ResultSet result=prepstmt.executeQuery();
prepstmt.setString(1,Harald);
ResultSet result=prepstmt.executeQuery();
```

CallableStatements werden im Zusammenhang mit *stored procedures* angewandt und bieten neben den in-Parametern noch einen out-Parameter.

Schließlich muss das Ergebnis, das nach einer Abfrage als Objekt der Klasse ResultSet vorliegt, verarbeitet werden. Die Methode next() erlaubt es, durch die Zeilen des Ergebnisses zu blättern. Die einzelnen Spalten können mit einer get-Anweisung, die den Typ des Datenfeldes kennen muss, ausgelesen werden.

```
while (result.next()) {
        String msg=results.getString(Message);
        String user=results.getString(User);
        System.out.println(Message from +user+ : +msg);
}
```

4.3 Objektorientierte Persistenz

Die im vorangegangenen Kapitel behandelten relationalen Datenbanken sind in der Praxis weit verbreitet, die Implementierungen sehr zuverlässig, hoch optimiert und performant. Doch dadurch, dass in relationalen Datenbanken Datenstrukturen eingesetzt werden, die nicht recht zum objektorientierten Paradigma passen, ergibt sich eine Kluft zwischen Programmiersprache und Datenbankmodell, die mit relativ viel Aufwand vom Programmierer geschlossen werden muss. Man spricht hier auch von einem *Impedance mismatch*.

Relationale Datenbanken sind nach dem theoretisch fundierten Relationenmodell von Codd entwickelt worden, und die Brücke zu Programmiersprachen wurde erst nachträglich geschlagen. Den umgekehrten Weg gehen objektorientierte Datenbanken: Nachdem sich das objektorientierte Paradigma für die Programmierung als sehr erfolgreich herausgestellt hat, werden nun Datenbanken entwickelt, die zu den hier vorhandenen Datenstrukturen, eben den Objekten, kompatibel sind. Diese Datenbanktechnologie ist noch sehr viel jünger als die relationale und in Skalierbarkeit, Stabilität und Performanz noch nicht auf einem vergleichbaren Niveau. Doch die Programmierung unter Verwendung von Objektdatenbanken ist erheblich einfacher und die Produktivität lässt sich mit ihnen beträchtlich steigern.

4.3.1 Objektorientierte Datenbanken

Anders, als bei relationalen Datenbanken, bei denen es eine standardisierte Anfragesprache SQL und standardisierte Schnittstellen wie ODBC und JDBC gibt, ist die Art, wie auf Objektdatenbanken gearbeitet und zugegriffen wird, noch sehr uneinheitlich. Zwar gibt es Standardisierungsbemühungen, vor allem von der ODMG (Object Database Management Group), einer Teilorganisation der OMG, die die Standardisierung für objektorientierte Datenbanken betreibt. Doch sind diese Bemühungen noch nicht abgeschlossen und haben auch nur beschränkte Relevanz. Daher muss man sich heutzutage noch einzelne Produkte ansehen. Die wichtigsten Datenbanken auf diesem Gebiet sind von Firmen wie Poet, Versant, Objectory und ObjectDesign.

Im folgenden wird die Datenbank ObjectStore von ObjectDesign [ObjectDesign 1998] verwendet. Dies hat mehrere Gründe: Zum einen ist ObjectDesign auf diesem Gebiet Marktführer, so dass die Wahrscheinlichkeit, dieses Produkt in der Praxis wiederzufinden, am größten ist. Zum zweiten ist die Unterstützung von ObjectStore für Java sehr gut und die Benutzung relativ einfach. Drittens existiert ObjektStore in drei verschiedenen Versionen, von denen die erste, ObjectStore PSE, völlig kostenfrei und die zweite zu moderaten Preisen im dreistelligen Bereich erhältlich ist. Diese Version, ObjectStore PSE Pro,

ist zu dem Vollprodukt kompatibel, so dass die Programmierung dieselbe ist. Diese Version ist für Single-User-Systeme ausgelegt, bietet ansonsten aber den vollen Leistungsumfang. Das Vollprodukt, ObjectStore, bietet darüber hinaus Möglichkeiten für den Mehrbenutzerbetrieb und große Datenmengen.

Zugriff auf persistente Objekte

Um auf Objekte in einer PSE Pro-Datenbank zuzugreifen, sind vorbereitend drei Schritte notwendig.

1. Sitzung eröffnen,
2. Datenbank öffnen oder erzeugen,
3. Transaktion starten.

Alle Aktivitäten auf einer Datenbank werden innerhalb von so genannten *Sitzungen* (Session) durchgeführt. Eine Sitzung muss zunächst erzeugt werden, danach können mehrere Threads einer Applikation an dieser Sitzung teilnehmen (join()). In PSE wird genau eine Sitzung unterstützt, während PSE Pro auch mehrere parallel nebeneinander geöffnete Sitzungen erlaubt.

```
Session session = Session.create(null,null);
session.join();
```

Ehe auf eine Datenbank zugegriffen werden kann, muss sie geöffnet werden. Um dies zu tun, wird bei PSE/PSE Pro die Methode Database.open() aufgerufen. Datenbanken haben einen Namen und werden über diesen identifiziert. Geschlossene Datenbanken werden auf Festspeichermedien auf dem Dateisystem gehalten. Die Datenbankdatei liegt in einfachen Fällen im gleichen Verzeichnis wie die Java-Anwendung. So kann sie einfach über den Dateinamen identifiziert werden. Beim Öffnen kann angegeben werden, ob auf die Datenbank nur lesend oder auch schreibend zugegriffen wird. Dies wird im zweiten Parameter über die Konstanten READ_ONLY oder UPDATE, die in der Klasse ObjectStore definiert sind, angegeben.

```
Database db = Database.open("CounterDb.odb", ObjectStore.UPDATE);
```

Natürlich muss eine solche Datenbank existieren, um auf sie zugreifen zu können. Wenn noch keine existiert, wird sie mit der Methode Database.create() erzeugt. Für Datenbanken können Zugriffsrechte definiert werden, ähnlich wie unter Unix Dateien Lese- und Schreibrechte für unterschiedliche Benutzergruppen haben. Hierfür stehen Konstanten wie ALL_READ, ALL_WRITE und OWNER_WRITE zur Verfügung. Java erlaubt allerdings nicht die Übersteuerung der Dateisystemmodi.

```
Database db = Database.create (
"CounterDb.odb", ObjectStore.ALL_READ|ObjectStore.ALL_WRITE);
```

Der Rahmen für alle Aktionen auf den Daten einer Datenbank ist die Transaktion. Innerhalb einer Transaktion kann auf Daten lesend oder schreibend zugegriffen werden. Wenn die Transaktion beendet wird, werden alle (geänderten) Daten persistent in die Datenbank geschrieben. Allerdings auch erst dann: Vor dem Ende einer Transaktion sind die Änderungen nicht für andere sichtbar. Eine Transaktion wird mit begin() begonnen und mit commit() beendet. Beide Methoden haben einen Parameter, der den Zugriff auf die Daten regelt. Beim Öffnen muss eine Zugriffsart angegeben werden. Transaktionen können als nur lesend definiert werden. Dann können mehrere solche Transaktionen gleichzeitig existieren. Wenn diese allerdings schreibend zugreifen sollen, darf nur eine einzige Transaktion geöffnet sein. Durch die Konstanten UPDATE und READ_ONLY wird dies unterschieden.

```
Transaction trx = Transaction.begin(ObjectStore.UPDATE);
```

Jetzt kann auf persistente Objekte zugegriffen werden. Objekte sind genau dann persistent, wenn sie entweder ein *Wurzelobjekt* der Datenbank sind oder von einem Wurzelobjekt erreicht werden können. Für eine Datenbank können mehrere Wurzelobjekte definiert werden. Üblicherweise sind dies Objekte, über die viele andere Objekte referenziert werden, wie zum Beispiel eine Hashtabelle oder ein Vektor. Unglücklicherweise sind im JDK bis Version 1.1 gerade diese Strukturen nicht persistenzfähig. Für diese Fälle existieren Austauschstrukturen, die denselben Zweck erfüllen und persistent sein können. ObjectStore bietet hierfür eigene Klassen, zum Beispiel OSHashtable und OSVector. Doch ab JDK 1.2, in der Collections eingeführt wurden, ist dieses Manko behoben.

Ein Wurzelobjekt wird durch createRoot() erzeugt, das einen Namen an ein Objekt bindet. Wurzelobjekte sind die Einstiegspunkte in den persistenten Objektgraph und Startpunkte der Navigation. Nur über diese lassen sich persistente Objekte erreichen.

```
OSHashtable counterHash = new OSHashtable();
db.createRoot("MyCounterHash", counterHash);
Counter accesses = new Counter();
accesses.set(42);
counterHash.put("AccessCounter", accesses);
trx.commit();
```

Über den Namen kann auf ein Wurzelobjekt durch die Methode getRoot() später wieder zugegriffen werden und über diese schließlich wieder auf alle persistenten Objekte.

```
Transaction trx2 = Transaction.begin(ObjectStore.UPDATE);
counterHash = (OSHashtable) db.getRoot("CounterHash");
accesses = (Counter) counterHash.get("AccessCounter");
```

Wie schon erwähnt, können Transaktionen durch commit() beendet und damit die Änderungen persistent gemacht werden. Alle Änderungen werden gleichzeitig und ungeteilt gespeichert. Alle Objekte, die von den Wurzelobjekten erreichbar

sind, werden persistiert. Damit geschieht die Speicherung persistenter Objekte in einer für den Programmierer einfachen und transparenten Weise. Es ist auch möglich, die Änderungen einer Transaktion zu verwerfen, womit der Zustand vor Beginn der Transaktion wiederhergestellt wird. Alle Änderungen auf den Daten werden entweder ganz oder gar nicht in die Datenbank übernommen. Für das Verwerfen wird die Methode abort() aufgerufen.

```
accesses.increase();
if (accepted) {
  // der Zähler wird erhöht
  trx2.commit()
} else {
  // der Zähler wird NICHT erhöht
  trx2.abort()
}
```

Der Zugriff auf persistente Objekte ist (im Standardfall) nur innerhalb von Transaktionen möglich. Nachdem ein commit() durchgeführt wurde, werden alle persistenten Objekte vom Java-Programm aus unerreichbar gemacht. Sie werden dann als *schal* (stale) bezeichnet, und ihre Werte werden gelöscht – natürlich nur im Programm und nicht im persistenten Speicher. Wenn auf ein schales Objekt zugegriffen wird, generiert das Datenbanksystem eine Ausnahme. Wenn auf diese Objekte erneut zugegriffen werden soll, muss eine neue Transaktion gestartet werden und das gewünschte Objekt vom Wurzelobjekt aus erreicht werden. Dies ist einerseits sinnvoll, da so Änderungen außerhalb von Transaktionen nicht in die Datenbank geschrieben werden können und das System nicht in einen inkonsistenten Zustand wechseln kann. Andererseits ist es aber sehr praktisch, gewisse Zustände zu erhalten und zum Beispiel in neuen Transaktionen weiterverwenden zu können. Wenn der Programmierer dies wünscht, kann er das durch einen zusätzlichen Parameter in der commit()-Methode (wie auch in der abort()-Methode) angeben. Dieser Parameter ist wieder ein int und für diesen stehen ebenfalls Konstanten zur Verfügung, die den gewünschten Zustand ausdrücken. Die Konstante RETAIN_STALE ist äquivalent zum Standardfall ohne Parameter. Mit RETAIN_HOLLOW bleiben Referenzen auf persistente Objekte auch außerhalb der Transaktion gültig, der Zugriff bleibt aber weiter untersagt und erzeugt eine Ausnahme. In einer neuen Transaktion kann aber wieder beliebig auf die Objekte zugegriffen werden. Um zumindest lesend auf solche Objekte zugreifen zu können, gibt man RETAIN_READONLY an. Mit RETAIN_UPDATE ist sogar eine Veränderung erlaubt, doch wenn die nächste Transaktion beginnt, werden diese Änderungen wieder verworfen.

Ein Nachteil dieser Vorgehensweisen ist, dass entsprechende Objekte nicht vom Garbage Collector eingesammelt werden können und somit den Hauptspeicher belasten, auch wenn sie nicht mehr benötigt werden. Man sollte also gelegentlich eine Transaktion mit RETAIN_STALE beenden.

Zum Schluss muss die Datenbank geschlossen und die Sitzung beendet werden. Wird dies nicht getan, bleibt die Datenbank gesperrt und kann bis zum Löschen des Locks nicht wieder geöffnet werden. Dieser Lock wird bei PSE durch

das Anlegen und Löschen eines (ansonsten leeren) Verzeichnisses realisiert, das den Namen der Datenbank mit der Endung .odx trägt.

```
db.close();
session.terminate();
```

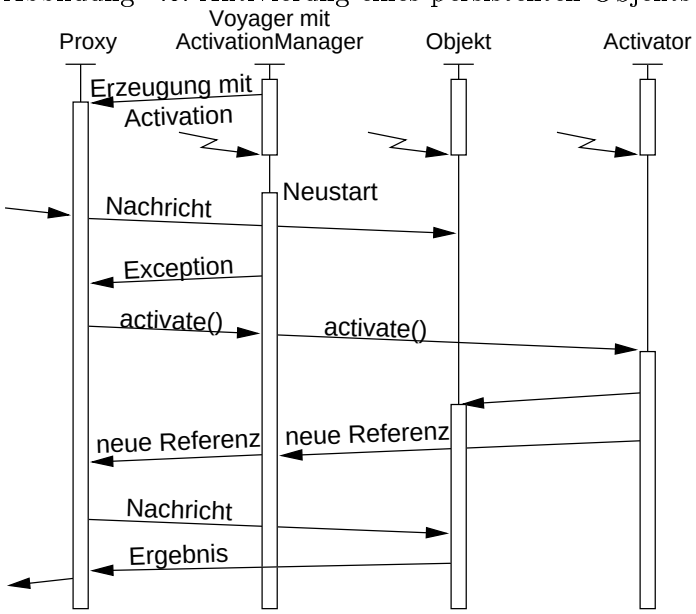
4.3.2 Reaktivierung von Objekten

In verteilten Systemen können einzelne Komponenten ausfallen und dabei Informationen über ihren Zustand verlieren. Persistenz kann dafür sorgen, dass die Daten bzw. die Objekte an sich nicht verloren gehen, doch die Konsistenz von entfernten Referenzen auf diese Objekte lässt sich so nicht gewährleisten. Hierfür benötigt man weitergehende Konzepte, die Objekte aus einer Datenbank wieder aktivieren können, sobald über entfernte Referenzen auf sie zugegriffen wird. Dies wird als Aktivierung bezeichnet. ObjectSpace bietet in Voyager einen so genannten *Activation Framework*, der genau dies leistet und besonders für den Einsatz in verteilten Systemen geeignet ist. Dabei müssen die Klassen der zu persistierenden Objekte in keiner Weise verändert oder vorbereitet werden. Es gibt auch keine zusätzlichen Prä- oder Postcompiler. Der Mechanismus setzt allerdings den Einsatz von Voyager und insbesondere die Verwendung von Proxies voraus. Diese Fähigkeit von Voyager lässt sich gut mit einer objektorientierten Datenbank verbinden, was im Folgenden mit ObjectStore demonstriert werden soll.

Wenn ein Objekt nicht mehr im Hauptspeicher existiert, kann es eigentlich nicht mehr referenziert und aufgerufen werden, auch nicht über entfernte Referenzen. Dass genau dies doch funktioniert, dafür sorgt das Voyager Activation Framework. Ein persistentes Objekt kann wieder aktiviert werden, sobald oder so lange auf dem ursprünglichen Rechner unter der entsprechenden Portnummer eine Voyager-Laufzeitumgebung gestartet ist. Proxies, die auf so ein persistentes Objekt verweisen, können in anderen Programmen natürlich noch existieren. Wenn ein Objekt über solch ein Proxy (das eine gewisse Aktivierungsinformation enthalten muss) aufgerufen wird und versucht das eigentliche Objekt zu kontaktieren, stellt es fest, dass das Objekt nicht mehr existiert. Dann schickt es die Aktivierungsinformation an die Voyager-Laufzeitumgebung und diese reaktiviert das Objekt. Danach kann der Aufruf ganz normal ausgeführt werden. Für den Entwickler ist dieser Mechanismus, der in Abbildung 4.3 skizziert ist, transparent.

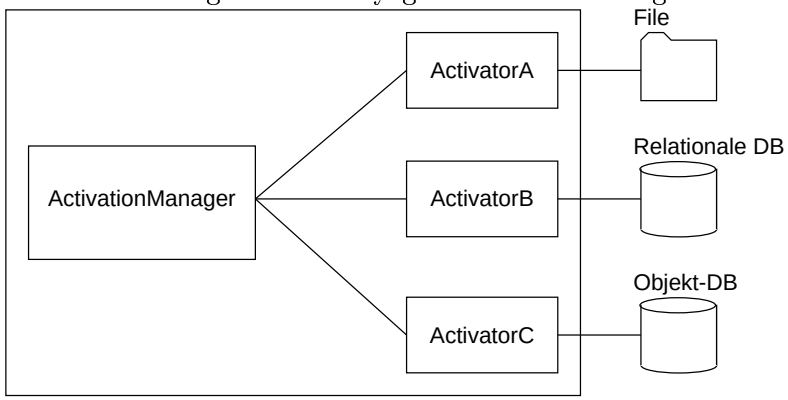
In jeder Voyager-Laufzeitumgebung ist ein ActivationManager enthalten, der eine Reihe von Activator-Objekten verwaltet. Jedes von diesen wiederum kann mit einer Datenbank verbunden sein, die entweder schlicht das serialisierte Objekt in eine Datei schreibt oder in Tabellen in eine relationale Datenbank abspeichert oder in einer objektorientierten Datenbank persistiert. Zu jedem Typ von Datenbank muss eine Klasse existieren, die für die Abbildung auf die Datenbankstruktur sorgt. Diese kann entweder für beliebige Objekte geschrieben oder für spezielle Anwendungen entwickelt werden. So kann ein Objekt zum Beispiel auf eine existierende Tabellenstruktur einer relationalen Datenbank abgebildet

Abbildung 4.3: Aktivierung eines persistenten Objekts



werden. Diese Abbildungsklasse muss das Voyager-Interface IActivator implementieren und wird dann Activator genannt.

Abbildung 4.4: Der Voyager ActivationManager



Ein neuer Activator wird beim Manager durch den Aufruf Activation.register() angemeldet. Dieses Interface schreibt die Methoden getMemento() und activate() vor. Der ersten Methode wird ein Proxy auf ein Objekt mitgegeben. Sie liefert einen String zurück, der für das Wiederauffinden des Objekts in der Datenbank sorgt. Dieser String wird *Memento* genannt, was soviel wie Erinnerungszeichen heißt. Der zweiten Methode wird dieses Memento übergeben, mit dem dann das Objekt wieder aus der Persistenzstarre erweckt und aktiviert wird. Danach kann es über den Proxy wieder angesprochen werden.

Diese Methoden sind allerdings nicht vom Entwickler aufzurufen, sondern vom ActivationManager. Der Entwickler muss in seinem Programm entscheiden, welche Objekte persistiert werden sollen und welcher Activator verwendet werden

soll. Dem Activator muss, je nach Implementierung, bei der Erzeugung der Name oder die URL der Datenbank übergeben werden, und er wird dann mit Activation.register() angemeldet. Dem Activator wird danach mitgeteilt, welche Objekte persistiert werden sollen. Dafür wird ihm eine Referenz auf ein Objekt oder auf dessen Proxy mit der Methode Activation.enable() übergeben. Anschließend erhalten alle neu erzeugten Proxies dieses Objekts die Information, die zur Aktivierung des Objekts nötig ist. Dies sind die URL des Programms, das Memento des Objekts und der Klassenname des Aktivators, der das Memento erzeugt hat.

Der Activator schreibt das Objekt beim Aufruf dieser Methode, ehe das Objekt vom Garbage Collector vernichtet wird und beim Beenden des Programms bzw. beim Beenden der Voyager-Laufzeitumgebung mit Voyager.shutdown() in die Datenbank. Das Objekt ist damit persistent und kann reaktiviert werden.

4.3.3 Orthogonale Persistenz

Nachdem nun Mechanismen zur persistenten Speicherung von Daten aus Java heraus beschrieben worden sind, stellt sich die Frage, ob man die Eigenschaft der Persistenz nicht vielleicht als elementaren Bestandteil der Sprachumgebung selbst in eine Sprache wie Java einbauen kann. Die Vision dabei ist, die Komplexität, die bisherige Ansätze zur Persistenz mit sich bringen, möglichst vollständig vor dem Programmierer zu verbergen und die Daten einer Anwendung oder gar eine Anwendung als Ganzes möglichst einfach persistent zu machen. In diesem Kapitel wird ein Ansatz vorgestellt, in dem genau dies versucht wird.

PJama ist eine experimentelle persistente Programmierumgebung für die Programmiersprache Java. Entwickelt wird PJama von den Mitgliedern der Forschungsgruppe »Persistence and Distribution« des »Department of Computing Science« an der Universität Glasgow in Zusammenarbeit mit den Sun Microsystems Laboratories im Rahmen des Projekts »Forest« [Atkinson et al. 1996], [Atkinson und Jordan 1998].

PJama realisiert die Prinzipien der orthogonalen Persistenz. Das bedeutet, dass jeder in einer Sprache vorkommende Datentyp ohne eine Transformation in ein anderes Datenformat persistent gemacht und wieder in den Hauptspeicher geladen werden kann. Im Falle von Java sind dies die Basistypen sowie zusammengesetzte Typen wie Arrays und Objekte. Dies bezieht aber auch die Threads und die grafische Oberfläche einer Anwendung und deren Zustand mit ein. Ebenfalls wird der zu einem Objekt gehörende Code mit abgespeichert.

PJama besteht aus der Implementation einer eigenen virtuellen Maschine, einigen veränderten Java-Basisklassen, eigenen spezifischen APIs sowie modifizierten Swing-Klassen. Es handelt sich im Grunde um ein alternatives JDK, das anstelle des normalen JDK eingesetzt werden kann und die Persistenz automatisch mit in eine Anwendung einbezieht. Vom Anwendungsprogrammierer selbst wird dafür nur ein Minimum an Aufwand erfordert.

Um mit persistenten Datenstrukturen arbeiten zu können, müssen diese auf einem dauerhaften Speichermedium vorgehalten werden. Die Datei, in der PJama Objekte speichert, ist der so genannte *persistent Store*. Einzelne Objekte werden dort als *persistente Wurzeln* namentlich registriert, wobei – gemäß dem Prinzip der »Persistenz durch Erreichbarkeit« – alle mit diesen Objekten in Ver-

bindung stehenden Klassen (Superklassen, Interfaces, benutzte andere Klassen etc.) ebenfalls persistent gemacht werden (transitive Persistenz). Die Vorgehensweise beim Einsatz von PJama lässt sich also wie folgt zusammenfassen:

1. Erzeugung eines persistenten Stores
2. Registrierung persistenter Wurzeln (»Füllen« des Stores)
3. Programmläufe mit Zugriff auf die als persistente Wurzeln gespeicherten Objekte und Klassen

Der persistent Store wird von PJama durch das PJStore-API zur Verfügung gestellt. Aus Gründen der Konsistenz kann ein Store nur von einer laufenden Anwendung benutzt werden, umgekehrt kann in der derzeitigen Version eine Anwendung auch nur auf einen Store zugreifen (diese Einschränkung soll jedoch in zukünftigen Versionen entfallen). PJama unterstützt außerdem die Evolution einzelner Klassen, Klassenhierarchien und persistenter Daten, d.h., Klassen eines persistenten Stores können durch neuere Versionen ersetzt werden, neue Klassen können eingefügt, alte Klassen gelöscht werden.

Für die Arbeit mit dem Store stehen die folgenden Tools zur Verfügung:

- opj – der PJama-Interpreter
- opjcs – erzeugt einen neuen Store (ohne persistente Wurzeln)
- opjgc – ein off-line Garbage Collector für persistente Stores
- opjsubst – ersetzt Klassen eines existierenden Stores (off-line)
- opjc – ein spezieller Compiler, der opjsubst unterstützt

Diese Tools arbeiten (mit Ausnahme von opjcs) alle auf einem bestehenden Store, der ihnen z.B. durch die Option -store als Argument übergeben wird. Compiliert werden PJama-Anwendungen – wie jedes andere Java-Programm auch – mit dem Compiler javac. Der Compiler opjc ist Teil des PJama-Evolutionssystems opjsubst und kommt nur bei der benutzerdefinierten Konvertierung bereits persistenter Daten zum Einsatz.

Die nachfolgenden Abschnitte behandeln Schritt für Schritt den Umgang mit PJama.

Der persistente Store

PJama speichert sowohl den Code als auch den Zustand von Objekten zusammen in einem persistenten Store. Der Store wird nicht im klassischen Sinne von Programmen geöffnet, sondern zusammen mit dem Namen einer ausführbaren Klasse dem PJama-Interpreter als Argument übergeben. Dieser durchsucht den Store nach der genannten Klasse und ruft deren main-Methode auf. Existiert diese Klasse nicht im Store, so wird der übliche Klassenpfad durchsucht. Die Regeln der »Persistenz durch Erreichbarkeit« garantieren, dass alle Klassen, die

von einer einmal persistent gemachten Klasse benötigt werden, ebenfalls persistent sind. In diesem Sinne kann ein persistenter Store auch als eine virtuelle Maschine angesehen werden, die zu einem bestimmten Zeitpunkt »eingefroren« (*stabilisiert*) wurde.

Die einfachste Art, einen persistenten Store zu erzeugen, ist die Verwendung des Tools opjcs. Als Argument wird der Pfadname des Stores angegeben (nach Konvention gekennzeichnet durch die Endung .pjs):

```
sun> opjcs /home/user/stores/TestStore.pjs
```

Ist der angegebene Pfad relativ, so geht opjcs vom aktuellen Arbeitsverzeichnis aus bzw. dem Verzeichnis, das durch die Umgebungsvariable STOREPATH definiert ist:

```
sun> setenv STOREPATH /home/user/stores
sun> opjcs TestStore.pjs
```

Ein neu erzeugter Store enthält eine Instanz der Implementation des PJStore-Interfaces, eine leere Tabelle der Objekte, die als persistente Wurzeln registriert sind, alle für die Funktion der virtuellen Maschine notwendigen Klassen sowie die Klassen, die von diesen aus erreichbar sind. Per Default wird er außerdem mit gewissen Eigenschaften initialisiert (»AutoWindows«), die es ermöglichen, bestehende AWT-Programme ohne Veränderungen am Code pseudo-persistent zu machen oder auch komplett »persistenzunabhängige« Programme zu schreiben. Für jeden Store wird schließlich noch eine Datei mit der zusätzlichen Endung .log angelegt, die bei der *Stabilisierung* des Stores (siehe Abschnitt 4.3.3) benutzt wird, um im Falle eines Systemabsturzes die Wiederherstellung zu ermöglichen.

Bei opjcs werden Optionen *nach* dem Pfadnamen des Stores angegeben:

- **overwrite.** Ein existierender Store gleichen Namens wird gelöscht, bevor der neue Store erzeugt wird.
- **verbose.** Bewirkt die Bildschirmausgabe verschiedener Informationen über den erzeugten Store sowie die von opjcs durchgeführten Aktionen.
- **autoWindowsOff.** Ein mit dieser Option erzeugter Store bietet nicht die o.g. Möglichkeit, AWT-Programme nachträglich persistent zu machen.

Wird opjcs ohne Argumente aufgerufen, startet das Programm in einem interaktiven Modus. Der Benutzer kann so den Pfadnamen des Stores und die gewünschten Optionen über eine grafische Oberfläche (AWT) eingeben.

Als zweite Möglichkeit kann ein persistenter Store aus einem Programm heraus erzeugt werden. Der typische Grund für diese Vorgehensweise besteht darin, gleich beim Anlegen des Stores eine Menge von persistenten anwendungsspezifischen Objekten zu speichern. Objekte und Klassen, die bei der ersten Stabilisierung (siehe Abschnitt 4.3.3) eines neuen Stores persistent gemacht werden, werden in einer speziellen Bootstrap-Region abgelegt, die gleich beim Start einer persistenten Anwendung geladen wird, was zu einer verbesserten Performanz führen kann.

Ein Programm, das einen neuen Store erzeugt, muss zunächst das PJStore-Interface und dessen Implementation importieren:

```
import org.opj.store.PJStore;
import org.opj.store.PJStoreImpl;
```

Um einen Store anzulegen, erzeugt man eine neue Instanz von PJStoreImpl, indem man dem Konstruktor einen String mit dem Namen des Stores übergibt:

```
PJStore ps = new PJStoreImpl("/home/user/stores/TestStore.pjs");
```

Beschreibt der übergebene String einen relativen Pfadnamen, so wird vom aktuellen Arbeitsverzeichnis ausgegangen bzw. dem Verzeichnis, das durch die Umgebungsvariable STOREPATH definiert ist. Der Kontruktor PJStoreImpl() ohne Argumente kann benutzt werden, wenn der Name des Stores durch das vorherige Setzen der Variable PJSTORE bestimmt wurde. Eine Kombination beider Variablen ist möglich.

Nachdem auf diese Weise ein persistenter Store erzeugt wurde, wird ein beliebiges Objekt persistent gemacht, indem es entweder selbst als persistente Wurzel namentlich registriert wird, oder aber durch ein Wurzelobjekt referenziert wird. Zur Registrierung persistenter Wurzeln steht die Methode newPRoot zur Verfügung, der als Argumente ein String mit dem Namen, unter dem das Objekt registriert wird, und eine Referenz auf das Objekt selbst übergeben wird:

```
Object anObject = new Object();
ps.newPRoot("My Object", anObject);
```

Das Programm (hier mit dem Namen CreateTest.java) wird mit dem Compiler javac übersetzt und anschließend durch den PJama-Interpreter opj ausgeführt:

```
sun> javac CreateTest.java
sun> opj CreateTest
```

Nach erfolgreicher Beendigung des Programms sind die Dateien TestStore.pjs und TestStore.pjs.log im entsprechenden Verzeichnis erzeugt worden.

Der PJama-Interpreter bietet neben allen Optionen, die auch der Java-Interpreter akzeptiert, zusätzlich u.a. die Option -store, die den Pfadnamen des Stores angibt. Diese Option entspricht einem temporären Setzen der Umgebungsvariablen PJSTORE. Wurde im Programm also der Konstruktor PJStoreImpl() ohne Argumente verwendet, so kann der Aufruf auch lauten:

```
sun> opj -store /home/user/stores/TestStore.pjs CreateTest
```

Der aktuelle Wert von PJSTORE spielt dabei keine Rolle und bleibt unverändert.

Um auf einen existierenden Store zuzugreifen, wird die statische Methode getStore() der Klasse PJStoreImpl aufgerufen:

```
PJStore ps = PJStoreImpl.getStore();
```

Die Objekte des Stores können dann durch bzw. über die persistenten Wurzeln verändert werden. Die Methode getPRoot() bekommt einen String mit dem Namen des Wurzelobjekts und liefert eine Referenz vom Typ Object:

```
Object anObject = ps.getPRoot("My Object");
```

Natürlich können auch (wie oben beschrieben) weitere Objekte als persistente Wurzeln registriert werden, wofür die Methode newPRoot() dient:

```
Object anotherObject = new Object();
ps.newPRoot("2nd Object", anotherObject);
```

Ein Wurzelobjekt kann natürlich auch wieder entfernt werden.Dazu ruft man die Methode ps.discardPRoot("2nd Object") auf. Hierbei wird nur der entsprechende Namenseintrag aus der Tabelle der persistenten Wurzeln gelöscht, das Objekt selbst verbleibt im Store. Es ist Aufgabe der Garbage Collection, das Objekt zu entfernen, falls es nicht mehr erreichbar ist.

Soll ein anderes Objekt unter einem bereits existierenden Namen als Wurzelobjekt registriert werden, so ist es nicht nötig, den Namen des ursprünglichen Objekts zunächst zu löschen, um dann eine neue persistente Wurzel zu setzen – die Methode setPRoot() vereinigt diese beiden Schritte:

```
Object oneMoreObject = new Object();
ps.setPRoot("My Object", oneMoreObject);
```

PJama hat in der vorliegenden Version die Einschränkung, dass eine Anwendung zur Laufzeit nur auf *einem* Store arbeiten kann: PJStoreImpl.getStore() bietet nicht die Möglichkeit, den zu benutzenden Store explizit anzugeben. Dieser wird beim Start des Programms (UseTest.java) dem Interpreter als Option übergeben:

```
sun> opj -store /home/user/stores/TestStore.pjs UseTest
```

Stabilisierung

Wurden an persistenten Objekten Änderungen vorgenommen, so muss der Store entsprechend aktualisiert werden. Diesen Vorgang bezeichnet man als Stabilisierung des Stores: Alle Objekte, die entweder direkt oder indirekt persistent gemacht wurden, werden überprüft und sämtliche geänderten Zustände in den Store geschrieben. Die Stabilisierung eines Stores ist eine atomare Handlung, so dass selbst bei einem Systemabsturz während der Stabilisierung der Store in einem konsistenten Zustand hinterlassen wird (wenngleich auch beim erneuten Aufruf des Interpreters möglicherweise einige Fehlerbehebungsroutinen im Hintergrund durchgeführt werden).

Der PJama-Interpreter stabilisiert den benutzen Store *automatisch* bei normaler Beendigung des Programms. Der Programmierer muss sich nicht explizit um die Stabilisierung kümmern. Im Falle eines Programmabsturzes hat der Store unverändert den Zustand wie zu Beginn der Programmausführung.

Soll ein Store außerdem zur Laufzeit einer Anwendung stabilisiert werden, so kann zu einem beliebigen Zeitpunkt dessen Methode stabilizeAll() aufgerufen

werden. Genau wie die implizite Stabilisierung am Programmende ist auch diese explizite Stabilisierung atomar: Bei einem Absturz während der Stabilisierung wird der Store in dem Zustand hinterlassen, den er zum Zeitpunkt der letzten Stabilisierung angenommen hat.

```
PJStore ps = PJStoreImpl.getStore();
...
ps.stabilizeAll();
...
```

Bei Anwendungen mit mehreren Threads kann die Stabilisierung problematisch sein, wenn die Threads nicht miteinander kooperieren. Während einer Stabilisierung werden alle Threads angehalten – die Stabilisierung ist atomar, isoliert und dauerhaft. Solange sie sich jedoch auf *alle* persistenten Objekte bezieht, gibt es keine Garantie für die semantische Konsistenz der angehaltenen Threads (mit Ausnahme des Threads, der die Stabilisierung hervorgerufen hat). Für zukünftige Versionen von PJama ist die Unterstützung persistenter Threads geplant, die dann bei einer Stabilisierung mit ihrer Ausführung fortfahren können, bis sie möglicherweise einen konsistenten Zustand erreicht haben.

Teil II

Vereinigung von Verteilung, Nebenläufigkeit und Persistenz

Kapitel 5

Grenzen und Möglichkeiten von Verteilungskonzepten

In den Kapiteln des ersten Teils dieser Arbeit wurden Konzepte vorgestellt, die in verteilten Systemen für die Bewältigung der Probleme von Verteilung, Nebenläufigkeit und Persistenz Einsatz finden. Dabei wurde deutlich, dass sehr unterschiedliche Konzepte für die Behandlung jedes einzelnen dieser Aspekte eingesetzt werden. In diesem Teil soll nun untersucht werden, ob eine Vereinfachung durch die Integration dieser Konzepte zu einem einzigen Konzept erreichbar ist. In diesem Kapitel soll zunächst das Wünschenswerte sorgfältig mit den Grenzen des Sinnvollen abgewogen werden, ehe in den anschließenden Kapiteln ein entsprechendes Konzept entwickelt wird.

In den Kapiteln 2 bis 4 wurden bereits für die einzelnen dort behandelten Aspekte jeweils Konzepte vorgestellt, die eine stärkere Integration in die Konzepte der Programmiersprache zum Ziel hatten.

Für den Aspekt der Nebenläufigkeit wurde das Konzept der leichtgewichtigen Prozesse vorgestellt, die als Java-Threads erstmalig ein fester Bestandteil einer weit verbreiteten objektorientierten Sprache wurden, vorgestellt. Doch die Diskussion des Konzeptes der Prozessoren, wie sie von Bertrand Meyer vorgeschlagen werden, legt nahe, dass hier eine bessere Integration möglich ist. Dieses Konzept ist aber bisher noch nicht in die Praxis umgesetzt worden.

Auf dem Gebiet der Persistenz wurden in den letzten Jahren große Fortschritte bei der Annäherung geeigneter Konzepte an die der Programmiersprachen erzielt. Im praktischen Einsatz befinden sich bereits objektorientierte Datenbanken, die den *impedance mismatch,* der für datenstrukturbasierten Konzepte für objektorientierte Sprachen gegeben ist, überwinden. Aber auch weitergehende Konzepte, die Persistenz orthogonal in die Sprache integrieren, werden erfolgreich entwickelt und wurden anhand des Beispiels Pjama vorgestellt.

Den Fokus der Diskussion in dieser Arbeit bildet jedoch der Aspekt der Verteilung. Für diesen wurde in Kapitel 2 das Konzept der unmittelbaren Kommunikation vorgestellt, in dem für die Verteilung auf der Sprachebene eine weitgehende Transparenz erreicht wurde. Dieser Ansatz wird in der Literatur als »Unified Objects« bezeichnet. Die Realisierbarkeit wurde durch Emerald und andere Sprachen gezeigt. Doch ob dieser Ansatz in einem verteilten Szenario

auch sinnvoll und praxistauglich ist, wurde mehrfach in Zweifel gezogen. Dies soll in Abschnitt 5.1 diskutiert werden. Aus dieser Diskussion werden als Folgerung Grenzen des Sinnvollen gezogen.

Auf Basis diesen Ergebnisses wird das Konzept der Kommunikation über Stellvertreter noch einmal betrachtet. Es wird herausgearbeitet, dass das in Java verwendete RMI hinter dem technisch Möglichen zurückbleibt. Aus diesen Betrachtungen wird in Abschnitt 5.2 als Synthese ein Anforderungskatalog erstellt, der das technisch Mögliche mit dem Sinnvollen verbindet.

5.1 Die Grenzen der unmittelbaren Kommunikation

Das in Emerald und anderen ähnlichen Sprachen verfolgte Ziel, den Unterschied zwischen lokalen und entfernten Objekten möglichst gänzlich verschwinden zu lassen, so dass die Verteilung letztendlich transparent wird, bezeichnet man als die Vision der *Unified Objects* (vereinheitlichte Objekte). In einem viel zitierten und respektierten wissenschaftlichen Bericht »A Note on Distributed Computing« [Waldo et al. 1994], setzen sich Jim Waldo und drei Kollegen mit dieser Vision auseinander. Sie analysieren dafür Projekte wie Emerald, Arjuna und Clouds, aber auch CORBA und ähnliche RPC-Systeme. Die Diskussion in diesem Abschnitt folgt im wesentlichen den Betrachtungen aus dem genannten Bericht.

Es gibt einige Punkte, die den Ansatz der Unified Objects sehr attraktiv machen:

- Ob ein Aufruf lokal oder entfernt ist, hat keine Auswirkungen auf die *formale Korrektheit* eines Programms. Denn wenn ein Objekt über eine bestimmte Schnittstelle verfügt und diese Schnittstelle *semantisch* korrekt unterstützt wird, hat die Lokalität eines Objekts keinen Effekt auf die Korrektheit des Programms.

- Die Schnittstelle eines Objekts ist unabhängig vom Kontext, in dem es einmal genutzt werden wird. Die Schnittstelle muss nicht nach Kriterien der Lokalität entworfen werden.

- Dadurch lassen sich verteilte Anwendungen mit den gleichen objektorientierten Entwurfsmethoden entwickeln, wie lokale Anwendungen.

- Ausfall- und Leistungsaspekte können in der Implementation einer Anwendungskomponente berücksichtigt werden. Die Betrachtung dieser Aspekte kann beim initialen Entwurf unberücksichtigt bleiben.

Die Tatsache, dass es Sprachsysteme gibt, die diese Vision auch realisieren – wie zum Beispiel Emerald –, zeigt, dass es durchaus möglich ist, diese Vision zu verwirklichen. Doch die Anwendungen, die damit gebaut werden konnten, waren stets von begrenzter Größe. Kritiker dieser Vision wie Jim Waldo argumentieren, dass dadurch die Unzulänglichkeiten der Vision verborgen bleiben und erst zu Tage treten, wenn die Anwendung eine gewisse Größe überschreitet. Für realistische Szenarien aber geht diese Vision an der Realität vorbei und ist

damit nicht die erstrebenswerte Lösung. Dies soll in den folgenden Abschnitten anhand verschiedener Aspekte dargestellt werden.

5.1.1 Unterschiede zwischen lokaler und verteilter Programmierung

Es gibt seit den siebziger Jahren Ansätze, zusätzliche Kommunikationsmechanismen in Programmiersprachen zu integrieren, so dass sich auch verteilte Anwendungen direkt damit entwickeln lassen. Sie folgten den jeweiligen Programmierparadigmen ihrer Zeit und waren zum Teil Erweiterungen existierender oder auch neuentwickelte Sprachen. In den siebziger Jahren waren dies *nachrichtenbasierte* Sprachen (etwa Occam oder CSP). In den Achtzigern basierten diese meist auf dem *entfernten Prozeduraufruf* (RPC) und in den Neunzigern schließlich auf dem objektorientierten entfernten Methodenaufruf. Die frühen Ansätze hatten nur einen sehr begrenzten Erfolg und blieben in der Handhabung schwierig und fehleranfällig. Heute liegt die Hoffnung vieler, diese Vision zu erreichen, auf dem komponentenbasierten Ansatz, der die Objektorientierung um zusätzliche Modularitäts- und Integrationskonzepte erweitert.

Allerdings gibt es auch weniger optimistische Betrachtungsweisen, die dieser Vision keine Chance einräumen. Dabei wird argumentiert, dass jeder Versuch der Vereinigung lokaler und entfernter Zugriffe versagen muss, da die Programmierung verteilter Anwendungen einfach nicht das gleiche ist, wie die Programmierung nicht-verteilter Anwendungen. Nur durch das bloße Integrieren von Kommunikationsmechanismen in die Sprache wird die Programmierung verteilter Anwendungen nicht leichter, da die Kommunikation zwischen den Teilen einer verteilten Anwendung nicht das eigentliche Hauptproblem darstellt. Die schwerwiegenden Probleme der verteilten Programmierung betreffen vielmehr ganz andere Bereiche. Dies sind zum Beispiel die Handhabung von Teilausfällen und das Fehlen eines zentralen Ressourcenverwalters. Des weiteren stellen auch das Zusichern angemessener Leistung und der Umgang mit Nebenläufigkeit schwerwiegende Probleme dar. Und schließlich müssen die Unterschiede bei Speicherzugriffen von lokalen und verteilten Einheiten gelöst werden. Erfahrungen haben gezeigt, dass bei der Entwicklung verteilter Anwendungen obige Probleme im Vordergrund standen und nicht die Arbeit mit dem bloßen entfernten Kommunikationsmechanismus. Diese Unterschiede zwischen lokaler und verteilter Ausführung sollen nun ausführlicher diskutiert werden, wobei insbesondere auf die Aspekte Latenzzeit, Speicherzugriff, Teilausfall und Nebenläufigkeit eingegangen wird.

5.1.2 Latenzzeit

Als erstes soll die Thematik der Latenzzeit behandelt werden, die den offensichtlichsten Unterschied zwischen lokalen und entfernten Aufrufen darstellt. Die Latenzzeit ist die Zeit, die nur für den Aufruf einer Methode ohne ihre Ausführung verbraucht wird, die also ein Maß für den Kommunikations- und Übertragungsaufwand bietet. Entfernte Aufrufe haben eine um 4 bis 5 Größenordnungen höhere Latenzzeit als lokale Aufrufe. Berücksichtigt man das Verhält-

nis, mit der die Entwicklung der Prozessor- und der Netzwerkgeschwindigkeit voranschreitet, so scheint eine Verringerung dieses Verhältnisses auch auf lange Sicht unwahrscheinlich. Statt dessen ist eher noch eine Vergrößerung zu erwarten.

Werden diese Leistungsunterschiede zwischen lokalen und entfernten Aufrufen bei der Entwicklung ignoriert, führt dies zu Anwendungen, die mit relativ hoher Wahrscheinlichkeit mit Leistungsproblemen zu kämpfen haben und nur in kleinen überschaubaren Anwendungen ausreichend robust sind. Die Verfechter der Unified Objects begegnen dieser Argumentation folgendermaßen:

- Auch wenn die Geschwindigkeit der Hardware schneller zunimmt als die der Übertragungstechnik, werden Effizienzargumente an Bedeutung verlieren, da auch hier enorme Geschwindigkeitszuwächse zu verzeichnen sind. Dies lässt sich auch bei anderen technologischen Neuentwicklungen beobachten, die sich am Anfang ebenfalls mit Effizienzproblemen auseinandersetzen mussten, die aber schließlich durch Optimierung und entsprechende Geschwindigkeitszuwächse an Akzeptanz gewonnen haben.

- Durch die Entwicklung und den Einsatz geeigneter Werkzeuge lassen sich Kommunikationsmuster zwischen den Objekten aufzeigen. Mit den neu gewonnenen Informationen können dann Objekte so gruppiert werden, dass die eng miteinander gekoppelten Objekte sich in demselben Adressraum befinden und solche mit wenig Kommunikation zueinander entfernt platziert werden können. Wichtig bleibt jedoch, dass zuerst die Anwendung korrekt funktioniert, bevor man sich Sorgen bezüglich der Effizienz macht.

Ob es möglich sein wird, die Effizienzunterschiede zwischen lokalen und entfernten Aufrufen ausreichend zu verdecken, hängt sehr stark von der jeweiligen Anwendung und der zukünftigen technologischen Entwicklung ab. Aber selbst wenn dies möglich ist, bleiben weitere Probleme, die schwerwiegender sind.

5.1.3 Speicherzugriff

Ein ebenfalls offensichtlicher Unterschied zwischen lokaler und entfernter Ausführung betrifft Speicherzugriffe, im besonderen durch den Einsatz von *Pointern*. Dies sind Zeiger, mit denen auf bestimmte Speicherbereiche direkt zugegriffen werden kann und deren Gültigkeit auf den lokalen Adressraum beschränkt ist. Wird versucht, solch einen Zeiger in einen anderen (entfernten) Adressraum zu übertragen, kommt es zu unvorhersehbaren Auswirkungen. Eine Programmiersprache, die eine Vermischung von Pointern und Objekten zulässt, ist C++. Solange der Programmierer nicht ausschließlich mit den objektorientierten Konzepten arbeitet und alternativ weiterhin bzw. zusätzlich Pointer einsetzen kann, d.h., über direkten Speicherzugriff verfügt, besteht ein großes Problempotential.

Ein möglicher Ausweg ist die Verwendung von *distributed shared memory*, was allerdings nur im Rahmen von recht eng gekoppelten Systemen sinnvoll ist. Die andere Möglichkeit ist die saubere Anwendung des objektorientierten Paradigmas, indem auf alle Objekte über logische Referenzen statt über physikalische Adressen zugegriffen wird. Java folgt von sich aus der zweiten Möglichkeit, doch

auch innerhalb der Objektorientierung treten einige speicherbezogene Probleme auf.

- Viele objektorientierte Sprachen bieten eine automatische Speicherbereinigung (Garbage Collection), die aber nur innerhalb eines lokalen Adressraumes oder, wie bei Java, innerhalb einer virtuellen Maschine funktioniert. Verteilt man eine Anwendung über mehrere Adressräume oder virtuelle Maschinen, ist dieser Mechanismus ungleich schwerer zu realisieren.

- Bei der Übergabe von Parametern bei einem Methodenaufruf werden im lokalen Fall meist Referenzen auf Objekte übergeben, so dass der Aufrufer ebenso wie der Aufgerufene auf dasselbe Objekt zugreifen und Änderungen am Objekt für beide sichtbar sind. Im entfernten Fall müssen aber aus Effizienzgründen (wegen der viel größeren Latenzzeit) meist Kopien übergeben werden. Dadurch entstehen Kopien eines Objekts, die nur schwer konsistent zu halten sind. Änderungen an einer Kopie sind am Original nicht sichtbar und umgekehrt gilt das gleiche.

Wie auch bei der Latenzzeit bleibt es beim Speicherzugriff vorstellbar, den Unterschied zwischen lokalen und entfernten Speicherzugriffen zu überdecken. Die in den nächsten Abschnitten vorgestellten Probleme, welche im Zusammenhang mit verteilter Ausführung durch Teilausfall und Nebenläufigkeit entstehen können, werfen jedoch die Frage auf, ob eine Vereinigung konzeptionell überhaupt noch möglich ist.

5.1.4 Teilausfälle

Das Versagen von technischen Bausteinen oder Elementen ist leider eine Tatsache in computerbasierten Systemen, sowohl bei der lokalen als auch bei der verteilten Ausführung. Der Unterschied zwischen den beiden Ausführungsarten besteht darin, dass es bei der verteilten Ausführung meistens nur zu Teilausfällen kommt, während bei der lokalen Ausführung Ausfälle entweder total sind, d.h., alle Komponenten, die zusammen für eine Anwendung arbeiten, sind betroffen, oder die Ausfälle können durch eine zentrale Instanz, welche sämtliche lokalen Ressourcen verwaltet, also zum Beispiel das lokale Betriebssystem, entdeckt werden.

So etwas ist im Zusammenhang mit verteilter Ausführung nicht möglich, da hier einzelne Komponenten, zum Beispiel Rechner oder Netzwerkverbindungen, ausfallen können, während die verbleibenden Komponenten weiterarbeiten. Diese Ausfälle verteilter Komponenten finden im allgemeinen unabhängig voneinander statt. Des weiteren gibt es keine globale Instanz, die in der Lage ist festzustellen, welche Komponente ausgefallen ist und anschließend die anderen Komponenten davon in Kenntnis setzen kann. Kein globaler Zustand existiert, der untersucht werden kann, um genau herauszufinden, welcher Fehler aufgetreten ist. In einem verteilten System kann der Ausfall einer Netzwerkverbindung nicht vom Ausfall eines Prozessors in einem anderen Rechner unterschieden werden.

Kommt es zu solchen Ausfällen, kann der Fehler nicht einfach durch eine *exception* abgefangen werden, wie es bei der lokalen Ausführung möglich ist. Vielmehr kommt es zu Komplikationen, wenn das Zielobjekt eines Aufrufs einfach verschwindet und dabei der Kontrollfluss nicht zurückkehrt. Daraus folgt, dass es eines der Hauptanliegen der verteilten Ausführung sein muss zu gewährleisten, dass der Systemzustand nach solch einem Ausfall konsistent ist. Diese Art Problematik tritt im Rahmen der lokalen Ausführung in dem Maße nicht auf.

Die Auswirkungen von Teilausfällen sind weitreichend. Sie betreffen den Entwurf von Schnittstellen, wie auch die Semantik von Operationen dieser Schnittstellen. Durch Teilausfälle wird die Programmierung um einen weiteren Aspekt ergänzt, dessen Handhabung keinesfalls trivial ist. Durch solche Fälle wird ein System indeterministisch. Im Gegensatz zur lokalen Ausführung, bei der Kenntnis über verschiedene Systemzustände erlangt werden kann, zum Beispiel vor und nach einem Ausfall, ist dies bei verteilter Ausführung nicht möglich. Dies erfordert die Anpassung der Schnittstellen, welche für Kommunikationsaufgaben verwendet werden, so dass die von Teilausfällen betroffenen Objekte konsistent reagieren können.

Möchte man trotz Teilausfällen robuste Anwendungen erhalten, so setzt dies bestimmte Ergänzungen im Schnittstellenbereich voraus. Das bloße Nachbessern der Implementation ist hierbei unzureichend. Statt dessen gilt es, die Schnittstellen so zu modifizieren, dass sie in der Lage sind, den Grund eines Ausfalls zu nennen, oder wenn dies nicht möglich ist, die Wiederherstellung eines sinnvollen Systemzustands zu unterstützen.

Das Vorhandensein von Teilausfällen bei verteilter Ausführung sollte aber nicht dahingehend interpretiert werden, dass es kein gemeinsames Objektmodell sowohl für lokale als auch für verteilte Ausführung geben kann. Aber anstatt zu fragen, *ob* man entfernte Methodenaufrufe wie lokale Methodenaufrufe aussehen lassen kann, gilt es vielmehr zu fragen, was der *Preis* für solch ein Unterfangen ist. Für die Umsetzung eines solchen einheitlichen Modells kommen nach [Waldo et al. 1994] nur zwei verschiedene Modelle in Frage.

Im Rahmen des ersten Ansatzes werden alle Objekte so behandelt, als wären sie ausschließlich *lokal*. Daraus folgt, dass die Schnittstellen der Objekte Verteilungsaspekte komplett außer acht lassen müssen. Verteilte Systeme, die unter Anwendung dieses Ansatzes entwickelt werden, weisen fundamentale Defizite auf. Ihr Verhalten bei Teilausfällen ist indeterministisch, was zur Folge hat, dass solche Systeme sehr fehleranfällig und somit nicht sehr robust sind. Emerald folgt diesem Ansatz. Für eng gekoppelte verteilte Systeme kann dieser Ansatz, wie Emerald zeigt, durchaus ausreichend sein.

Im Gegensatz zum ersten Ansatz werden beim zweiten Ansatz alle Objekte so behandelt, als wären sie ausschließlich *entfernt*. Dies macht sich insbesondere beim Schnittstellenentwurf bemerkbar. Denn nun verfügen sämtliche Objekte, also auch solche, die niemals entfernt verwendet werden sollten, über zusätzliche Sicherheitsaspekte in ihren Schnittstellenbeschreibungen. Des weiteren gilt zu beachten, dass dieser Ansatz, ähnlich wie beim Problem des Speicherzugriffs, nur dann funktioniert, wenn entweder der Programmierer sich diszipliniert an die Vorgaben hält, d.h., nur verteilte Objekte benutzt, oder die Programmiersprache

entsprechend angepasst wird.

Die Verwendung dieses Ansatzes hätte aber zur Folge, dass man das Gegenteil von dem erreicht, was das ursprüngliche Ziel eines einheitlichen Objektmodells für lokale und entfernte Ausführung gewesen ist. Denn der wirkliche Grund für den Versuch solcher Vereinigung ist es, die Entwicklung verteilter Anwendungen näher an die Entwicklung lokaler Anwendungen zu rücken und somit die verteilte Anwendungsentwicklung als Ganzes leichter zu machen. Dies wird aber beim zweiten Ansatz, der zwar sicher und robust ist, nicht erreicht. Statt dessen wird selbst die bisher einfache Entwicklung lokaler Anwendungen unnötig komplex.

5.1.5 Nebenläufigkeit

In vielen verteilten Systemen hat man nicht nur mit den Problemen der Verteilung, sondern auch mit denen der Nebenläufigkeit zu kämpfen. In Sprachen wie Java sind diese Aspekte mit voneinander unabhängigen Konzepten zu behandeln, etwa mit Threads einerseits und RMI andererseits. Der RMI an sich ist nicht nebenläufig, sondern führt strickt sequentielle und synchrone entfernte Aufrufe aus. Die Threads hingegen haben an sich nichts mit Verteilung zu tun. Doch um die Vorteile verteilter Systeme nutzen zu können, müssen beide gemeinsam verwendet werden, um zum Beispiel einen asynchronen und parallel ablaufenden entfernten Aufruf abzusetzen.

Die Nebenläufigkeit birgt ganz eigene Probleme bei der Integration in eine objektorientierte Sprache. In einigen Projekten wird der Aspekt der Nebenläufigkeit unabhängig von der Verteilung in eine Sprache integriert, um zum Beispiel in Clustern oder Parallelrechnern Einsatz zu finden, in denen man ganz andere Annahmen über Latenzzeit, Speicherzugriff und Teilausfall machen kann, als in verteilten Systemen. Dabei stößt man auf die folgenden Probleme:

- Mit der Einführung von Nebenläufigkeit hält auch ein Indeterminismus in eine Sprache oder ein System Einzug. Die Ausführungsreihenfolge von nebenläufigen Programmteilen kann nicht mehr garantiert werden oder muss durch zusätzliche Maßnahmen wie Synchronisation erzwungen werden.

- Einen geeigneten Mechanismus zur Synchronisation zu finden ist schwierig. Natürlich besteht die Möglichkeit, Konstrukte wie Semaphore, Monitore, Locks und Ähnliches zu verwenden, doch dies läuft auf eine schwierige und für jedes Problem neu zu lösende Programmierung hinaus. Wünschenswert ist ein besser in die Sprache integrierter Mechanismus.

- Durch das exklusive Reservieren oder Sperren von Ressourcen, das mit der Synchronisation einhergeht, besteht die Möglichkeit von Deadlocks. Ein geeigneter Synchronisationsmechanismus muss diese entweder vermeiden oder erkennen und beheben können.

- Ein schwieriges Problem bei Integration eines Synchronisationsmechanismus in objektorientierte Sprachen stellt die Vererbung dar. In vielen Ansätzen wird die Synchronisation auf Ebene der Klasse definiert, indem

> Regeln für die nebenläufige oder sequentielle Ausführung von Methoden eingeführt werden. Wenn nun eine weitere Klasse von dieser Klasse abgeleitet wird, kann dies die bisherige Synchronisation völlig durcheinander werfen. Man spricht hierbei von der Vererbungsanomalie.

Bei der Integration von Nebenläufigkeit in eine verteilte Sprache bedeutet dies vor allem, dass lokale Objekte zueinander sequentiell und entfernte Objekte zueinander potentiell nebenläufig sind. Die Behandlung von Nebenläufigkeit muss also für entfernte Objekte erfolgen. Um das Ziel der Vereinigung von entfernten und lokalen Objekten im Sinne der Unified Objects zu erreichen, bedeutet dies, wie auch bei den Teilausfällen, dass die für Teilausfälle angeführte Argumentation sich ebenfalls auch auf Nebenläufigkeit übertragen lässt. Denn entweder berücksichtigen alle Objekte die nebenläufige Semantik, oder sie lassen dieses Problem bewusst außer acht und sind bereit, die damit verbundenen Konsequenzen zu tragen.

5.1.6 Unterscheidbarkeit von lokalem und entferntem Zugriff

Die bisherige Diskussion legt deutlich nahe, dass eine Vereinigung von lokalem und verteiltem Programmiermodell im Sinne der Unified Objects nicht sinnvoll ist. Das ursprüngliche Ziel lässt sich nur auf zwei Wegen erreichen. Dabei ist beiden gemein, dass es im Rahmen der Umsetzung nur Objekte von einer einzigen Art, d.h., lokal oder entfernt, geben kann. Denn sonst gäbe es kein einheitliches Modell mehr.

Durch die Entscheidung, sämtliche Objekte so zu behandeln, als wären sie alle lokal, werden durch das Nichtbeachten von wichtigen Verteilungsaspekten, wie zum Beispiel Teilausfall und Indeterminismus, unkalkulierbare Risiken eingegangen. Der entgegengesetzte Ansatz, alle Objekte so zu behandeln, als wären sie ausschließlich entfernt, bringt eine unnötige Komplexität mit sich. Denn hierdurch sind auch solche Objekte betroffen, die überhaupt nicht entfernt verwendet werden sollen. Diese Objekte müssten Verteilungsaspekte implementieren, die sie nie benötigen werden und wodurch ihre Entwicklung unnötig kompliziert wird. Generell lässt sich sagen, dass die Entwicklung von Anwendungen, ob lokal oder verteilt, im Rahmen dieses Ansatzes weitaus komplexer wäre, als dies normalerweise notwendig ist.

Viel sinnvoller wäre es statt dessen, die unvereinbaren Unterschiede zwischen lokaler und verteilter Programmierung zu akzeptieren und sich der Unterschiede während aller Phasen der Entwicklung und Implementierung von verteilten Anwendungen bewusst zu sein. Dies erreicht man durch eine unterschiedliche Behandlung von lokalen und entfernten Objekten. Lokale und entfernte Objekte müssen dafür voneinander unterschieden werden können.

Hierdurch bleibt die Entwicklung der lokalen Aspekte einer verteilten Anwendungen unverändert einfach. Für die verteilten Aspekte kann sich der Programmierer aufgrund der Unterscheidung auf die verschiedenen Verteilungsprobleme, wie zum Beispiele Teilausfälle oder Indeterminismus, bewusst einstellen. Doch auf welche Weise kann solch eine Unterscheidung von lokalen und entfernten Objekten oder Aufrufen erfolgen?

5.2 Die Grenzen der Kommunikation über Stellvertreter

Jim Waldo hat unter Berücksichtigung dieser Diskussion entscheidend an der Entwicklung des Java-RMI mitgewirkt. RMI ist eng genug in Java integriert, um ihn als einen Bestandteil der Sprache bezeichnen zu können. Ist Java mit RMI schon die Lösung auf die gestellte Frage? Stellt Java selbst schon eine verteilte Programmiersprache dar? In RMI werden entfernte Objekte durch das Typsystem eindeutig gekennzeichnet, sie erben von der Klasse RemoteObject, und es muss die entfernt zugreifbare Schnittstelle in einem Interface festgelegt werden, das von dem Interface Remote abgeleitet ist. Damit sind entfernte Referenzen eindeutig und ausreichend gekennzeichnet und der RMI erfüllt die Forderungen aus der obigen Diskussion.

Doch der RMI weist Nachteile und Begrenzungen auf, die die Bezeichnung »verteilte Sprache« für Java fragwürdig machen:

- Da auf entfernte Objekte nur über ein Interface zugegriffen werden kann, weist der RMI auch alle Beschränkungen auf, denen Interfaces obliegen. Dies bedeutet, dass nur als öffentlich (public) gekennzeichnete Methoden entfernt aufgerufen werden können. Man kann sicherlich argumentieren, dass dies ausreicht, doch es stellt eine Einschränkung dar, die einer vollen Integration in die Objektorientierung deutlich widerspricht. Nicht möglich sind Zugriffe auf normale (default), geschützte (protected) oder statische (static)Methoden.

- Der RMI ermöglicht keine Migration von Objekten. Zwar können Kopien von Objekten als Parameter eines Methodenaufrufs übergeben werden, so dass Objekte in gewissem Sinne von einem Rechner auf einen anderen verschoben werden können. Doch die Objekte können dabei nicht ihre Identität wahren, sondern stellen eben Kopien dar. Man könnte meinen, dass dies für eine Simulation von Migration ausreicht, indem man ein Objekt auf einen entfernten Rechner kopiert und das Original löscht. Doch Referenzen, die auf das Original verweisen, können nicht auf die Kopie umgelenkt werden, was für eine echte Migration gefordert werden muss.

- Objekte können auch nicht auf einem entfernten Rechner erzeugt werden. Auf existierende entfernte Objekte kann durch den Nameservice zugegriffen werden, doch die Erzeugung von entfernten Objekten ist direkt nicht möglich. Um dies zu erreichen, müssen so genannte Factory-Objekte auf einem entfernten Rechner existieren, die dann ein (für sie lokales) Objekt erzeugen und eine Referenz darauf zurückliefern können.

- Asynchrone Aufrufe sind nicht möglich. Diese müssen mit Hilfe von Threads aufwendig simuliert werden. Ohne asynchrone Aufrufe geht aber ein wichtiger Vorteil verteilter Systeme, die natürliche Parallelität, verloren.

- Insgesamt stellt der RMI keine Unterstützung von Nebenläufigkeit zur Verfügung. Die Integration von sowohl Nebenläufigkeit als auch Verteilung

auf eine direkte und einfache Weise ist aber eine große Hoffnung, die mit der Vorstellung einer »verteilten Programmiersprache« verbunden wird.

Um zumindest einige dieser Nachteil zu beheben, besteht die Möglichkeit, den Mechanismus des RMI selbst zu erweitern. Ein Projekt, das diesen Ansatz recht erfolgreich verfolgt hat, ist JavaParty [Philippsen und Zenger 1996]. Entfernte Objekte werden hier nicht durch das Erben von speziellen Klassen und Interfaces, sondern durch ein zusätzliches Schlüsselwort gekennzeichnet, das als Klassenmodifikator der Deklaration einer Klasse vorangestellt wird. Dadurch stehen nicht nur die öffentlichen, sondern alle Methoden und Variablen einer Klasse auch entfernt zur Verfügung. Auch die Migration von Objekten und die entfernte Erzeugung ist möglich. Der Preis dafür ist, dass für jede potentiell entfernte Klasse etwa 10 zusätzliche Klassen von einem dem rmic vergleichbaren Compiler erzeugt werden müssen. Aspekte der Nebenläufigkeit werden auch hier nicht behandelt.

CORBA folgt im wesentlichen dem gleichen Ansatz wie RMI, ist zusätzlich aber auch für die Anbindung von Programmen in anderen Sprachen und zu Legacy-Anwendungen geeignet. Doch die Kritikpunkte, die für RMI gelten, finden auch hier ihre Gültigkeit. Darüber hinaus sind sie für CORBA schwerer zu überwinden. Man verliert die Vorteile von Java und kann nicht mehr auf die Plattformunabhängigkeit und Mobilität des Codes bauen, so dass Objektmigration und entfernte Erzeugung sogar unmöglich erscheinen. Zwar gibt es hier Ansätze, zum Beispiel für die Migration, doch diese bleibt noch zwingender auf das Erstellen einer Kopie und Löschen des Originals beschränkt.

Einen erheblichen Fortschritt stellt hier Voyager dar. Migration ebenso wie entfernte Objekterzeugung sind ohne weiteres möglich. Methodenaufrufe müssen nicht synchron sein, sondern können asynchron oder sogar gänzlich ungekoppelt (oneway) ablaufen. Damit bietet es ein Mindestmaß an Unterstützung von Nebenläufigkeit, auch wenn Konzepte zur Synchronisation fehlen.

Doch Voyager hat andere Nachteile, durch die auch der Kombination Java-Voyager nicht das Prädikat einer verteilten Sprache zuteil wird:

- Der entfernte Zugriff auf Voyager-Objekte ist ebenfalls auf ein Interface beschränkt und erlaubt keine Benutzung der vollständigen Schnittstelle.

- Voyager bietet zwar einen asynchronen Aufruf, doch die Syntax dieses Aufrufs folgt nicht objektorientierten Prinzipien. Die zu verwendende Schnittstelle für einen solchen Aufruf sieht folgendermaßen aus:
  `Future.invoke(Object object, String method_name, Object[] param);`
  Es wird also zum einen nicht eine Methode des entfernten Objekts aufgerufen, sondern eine statische Methode der Klasse Future. Der Methodenname wird lediglich als String angegeben. Die Verantwortung, dass das angegebene Aufrufziel object diese Methode überhaupt enthält, liegt ganz beim Programmierer und kann nicht von einem Compiler überprüft werden. Auch die Parameter werden als Array von Objekten übergeben, wobei sie die Typinformationen verlieren. Dadurch geht auch die Typsicherheit für solche Aufrufe verloren.

- Bei der Migration von Objekten muss die referentielle Integrität gewahrt bleiben, das heißt, dass alle Objekte, die über Referenzen erreicht werden können, auch nach der Migration erreichbar sein müssen. Da lokale Referenzen nicht automatisch in entfernte Referenzen umgewandelt werden können, müssen alle lokal referenzierten Objekte (die transitive Hülle) als Kopie auf dem neuen Rechner eingerichtet werden. Dadurch kann die Migration eines einzelnen Objekts zu einer Verschiebung eines ganzen Objektgraphen über das Netz führen.

- Die transitiven Hüllen von zwei Objekten können sich überlappen: Ein Objekt C, das von zwei Objekten A und B referenziert wird, gehört sowohl zur transitiven Hülle von A als auch von B. Bei einer Migration von A oder B wird von C eine Kopie erstellt, so dass es von diesem Objekt hinterher zwei Versionen gibt, deren konsistenter Zustand nicht gewährleistet ist.

Bei der Suche nach einem verteilten Java ist also noch keine befriedigende Lösung erreicht.

5.3 Eine Synthese

Was für ein Fazit lässt sich aus der in diesem Kapitel geführten Diskussion ziehen? Die angesprochenen wünschenswerten Eigenschaften und die ungelösten Probleme sollen hier in einer Anforderungsliste zusammengefasst werden:

1. Zugriffe auf entfernte Objekte sollten über den üblichen objektorientierten Mechanismus des Methodenaufrufs möglich sein. Dabei sollte aber die Sichtbarkeit der vollen Schnittstelle erhalten bleiben, wie im lokalen Fall, und nicht durch den Verteilungsmechanismus eingeschränkt werden.

2. Die Erzeugung von Objekten auf einem entfernten Rechner sollte möglich sein. Dass ein entfernter Rechner erst durch einen Daemon wie bei Voyager gestartet werden muss, ist akzeptabel und schon aus Sicherheitsgründen notwendig. Auch das vorherige Setzen von entsprechenden Rechten oder das Anmelden von Code-Servern ist kein Manko, sondern eine Notwendigkeit.

3. Lokale und entfernte Referenzen auf Objekte sollten klar voneinander unterschieden werden können, wie es Waldo fordert. Diese Unterscheidung sollte möglichst einfach und intuitiv sein und sich gut in die Sprache integrieren. Dennoch sollte der Unterschied dem Entwickler jederzeit erkennbar sein, um Entwurfsentscheidungen richtig treffen und Ausnahmen geeignet behandeln zu können. Der Unterschied sollte auch dem Compiler und dem Laufzeitsystem erkenntlich sein, so dass sie die korrekte und konsistente Anwendung entfernter Zugriffe überprüfen und den Programmierer möglichst gut unterstützen können.

4. Wenn die Aspekte der Verteilung nicht benötigt werden, sollten Programme von dem Vorhandensein dieser Aspekte möglichst vollständig unbeein-

trächtigt bleiben, so dass die Entwicklung von lokalen Anwendungen sich nicht ändert.

5. Die automatische Speicherbereinigung (Garbage Collection) sollte auch im verteilten Fall erhalten bleiben.

6. Die Migration von Objekten sollte möglich und einfach zu bewerkstelligen sein. Dabei sollte die Identität eines Objekts in vollem Umfang erhalten bleiben. Dies bedeutet, dass Referenzen auf entfernte Objekte auch nach einer Migration ihre Gültigkeit nicht verlieren.

7. Es sollte einen Mechanismus zum Zusammenfassen von Objekten geben, um sie zum Beispiel gemeinsam zu migrieren. In Emerald lassen sich Objekte einzeln bewegen oder durch eine transitive Verknüpfung (*attach*) aneinander binden. Doch dieser Mechanismus ist zu aufwendig und greift zu tief in den Code einer Anwendung hinein. Die Migration der transitiven referentiellen Hülle, wie bei Voyager, birgt aber auch Probleme, insbesondere weil sich die transitiven Hüllen überlappen können.

8. Methoden sollten sich synchron oder asynchron aufrufen lassen. Dabei sollte aber die objektorientierte Notation erhalten bleiben, um nicht, wie bei Voyager im Fall des asynchronen Aufrufs, die Typsicherheit zur Compilezeit aufgeben zu müssen.

9. Insgesamt sollte ein Mechanismus zur Behandlung von Nebenläufigkeit vorhanden sein, da dies eine natürliche Begebenheit und ein großer Vorteil verteilter Systeme ist. Er sollte möglichst direkt und einfach in die Programmiersprache integriert sein und die Prinzipien der Objektorientierung widerspiegeln, wie dies zum Beispiel bei Eiffel SCOOP vorgeschlagen wird.

10. Dies erfordert auch die Integration von Synchronisation in die Sprache. Wie Bertrand Meyer argumentiert, sollte diese auf dem Kommunikationsmechanismus aufsetzen und nicht ein zusätzlicher Mechanismus sein.

Alle diese Punkte werden einzeln in verschiedenen Systemen erfüllt und sind daher keine utopischen Forderungen. Doch keines der untersuchten Systeme kann alle diese Punkte gleichzeitig erfüllen. Gesucht ist ein Mechanismus, der möglichst viele dieser Forderungen befriedigen kann.

Doch wie kann ein Konzept aussehen, das diesen hohen Anforderungen gerecht wird? Bei der Betrachten der untersuchten Mechanismen fällt auf, dass Voyager und Eiffel SCOOP jeweils sehr viele der Punkte erfüllen. Ist es möglich diese beiden Ansätze sinnvoll zusammenzuführen und dadurch ihre guten Eigenschaften zu kombinieren? Dies soll im nächsten Kapitel untersucht werden.

Kapitel 6

Ein Konzept zur Vereinigung von Verteilung, Nebenläufigkeit und Persistenz

In diesem Kapitel soll ein Konzept entwickelt werden, das der Diskussion des letzten Kapitels standhält und die aufgestellten Forderungen erfüllt. Gesucht ist eine Abstraktion, die Verteilung, Nebenläufigkeit und Persistenz zu einem einzigen Konzept vereint. Abstraktion ist einer der wichtigsten Schlüssel zu Fortschritten in der Informatik (und natürlich nicht nur da). Hier seien drei Beispiele für solche Abstraktionen genannt, die für die Entstehung einer Sprache wie Java Voraussetzung waren oder sie zumindest begünstigt haben:

1. Daten, Funktionen und Module stellen in der prozeduralen Programmierung fundamentale, aber voneinander getrennte Konstrukte dar. Diese werden in der Objektorientierung zu einem einzigen Konstrukt, nämlich der Klasse, abstrahiert.

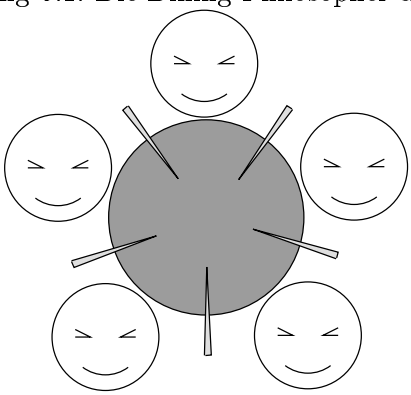
2. Die Abstraktion von physikalischen Speicheradressen (*Pointer*) hin zu logischen Referenzen hat die automatische Speicherbereinigung ebenso wie das Konzept der entfernten Referenzen erst möglich gemacht.

3. Eine der wichtigsten Abstraktionen bezüglich Java ist die virtuelle Maschine – eine Abstraktion von der tatsächlichen *physikalischen* Maschine, dem Rechner. Sie verdeckt die Unterschiede, die verschiedene Plattformen nun einmal haben. Dadurch ist Java plattformunabhängig und der ausführbare Code kann von einer Maschine auf eine andere migriert werden. Das bekannteste Beispiel hierfür sind Applets, deren Klassen von einem Webserver geladen und auf einer beliebigen Plattform in einem Browser gestartet werden können.

Was hätte eine Abstraktion von Nebenläufigkeit und Verteilung für einen Vorteil? Dazu sei hier ein Beispiel betrachtet, das sowohl in dem einen als auch in dem anderen Bereich häufig verwendet wird: das Problem der Dining Philosopher. Fünf Philosophen sitzen, wie in Abbildung 6.1 gezeigt, um einen Tisch und

speisen zusammen. Sie haben eine gemeinsame Schüssel Reis, aus der sie mit jeweils zwei Stäbchen essen können, doch jedes Stäbchen müssen sich jeweils zwei Philosophen teilen. Die Philosophen können entweder nachdenken, wozu sie die Stäbchen links und rechts von sich ablegen, oder essen, wofür sie sowohl das rechte als auch das linke Stäbchen erhalten müssen.

Abbildung 6.1: Die Dining Philosopher am Tisch

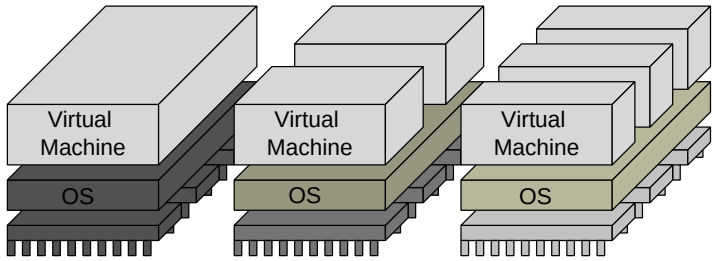


Dieses Beispiel lässt sich mit Threads realisieren, wobei jeder Philosoph einen eigenen Thread erhält. Alle Threads laufen dabei nebenläufig auf einer virtuellen Maschine. Es lässt sich aber auch in einem verteilten System verwirklichen, wobei sich (zum Beispiel) alle Philosophen auf verschiedenen Rechnern befinden und die entfernte Kommunikation über Sockets, RMI, CORBA oder Voyager implementiert werden kann. Doch diese beiden Ansätze haben nicht viel miteinander gemein. Die nebenläufige Version im nachhinein zu verteilen, bedeutet im Grunde genommen das Programm komplett umzuschreiben. Für die Umwandlung von der verteilten Version zu einer lokalen trifft dasselbe zu. Hätte man nun eine Abstraktion, die sowohl Verteilung als auch Nebenläufigkeit ausdrücken könnte, bräuchte man dieses Problem nur einmal zu implementieren und könnte sich zur Laufzeit entscheiden, ob es lokal (alle Philosophen auf einem Rechner) oder verteilt (jeder Philosoph auf einem eigenen Rechner) ausgeführt werden soll. Man könnte es sogar zur Laufzeit ändern, indem man einfach die Philosophen nach Belieben von einem Rechner zu einem anderen migrieren lässt. Sie könnten alle auf einem Rechner erzeugt und gestartet werden, ihr Mahl beginnen und zu einem beliebigen Zeitpunkt von hier auf einen anderen verschoben werden, ohne dass sie das Mahl unterbrechen müssten.

Und wie lässt sich eine solche Abstraktion finden? Dafür soll noch einmal die virtuelle Maschine von Java betrachtet werden. Die virtuelle Maschine ist, wie der Name schon sagt, ein virtuelles Konzept. Es entspricht nicht einer darunterliegenden physikalischen Realität. Auf einem Rechner können keine, eine oder aber auch mehrere virtuelle Maschinen gestartet werden, wie Abbildung 6.2 zeigt.

Zwei virtuelle Maschinen laufen zueinander nebenläufig. Befinden sie sich auf einem einzigen Rechner, so laufen sie in zwei unterschiedlichen Prozessen. Wenn sie auf zwei verschiedenen Rechnern gestartet werden, sind sie sogar zueinander echt parallel.

Abbildung 6.2: Die virtuelle Maschine verdeckt Unterschiede der Plattform



Sie sind auch verteilt. Um von einer virtuellen Maschine zu einer anderen kommunizieren zu können, muss entfernte Kommunikation eingesetzt werden. Dabei ist es völlig unabhängig davon, wo die beiden kommunizierenden virtuellen Maschinen sich physikalisch befinden (solange sie über ein Netzwerk erreichbar sind). Sie können sich auf demselben Rechner befinden oder auf zwei verschiedenen Kontinenten, für den Kommunikationsmechanismus ist dies unerheblich. Natürlich wird sich die Übertragungszeit und die Stabilität der Verbindung ändern, doch ansonsten sind beide auf dieselbe Weise füreinander erreichbar.

Doch leider lässt sich eine virtuelle Maschine nicht migrieren. Sie ist an den Ort, an dem sie gestartet wurde, fest gebunden. Einzelne Objekte können, wie Voyager zeigt, aus ihr heraus oder in sie hinein migriert werden, doch sie selbst ist statisch und unverschiebbar. Dies lässt sich auch nicht ändern, da virtuelle Maschinen selbst sehr wohl plattformabhängig sind.

Aber vielleicht lässt sich eine Abstraktionsschicht zwischen die virtuelle Maschine und die Objekte ziehen, die genau dies ermöglicht: ein ebenfalls virtuelles Konstrukt mit ähnlichen Fähigkeiten wie der virtuellen Maschine, von der es keines, eins oder ganz viele in einer virtuellen Maschine geben kann, und das sich migrieren lässt. In Anlehnung daran, dass physikalische Maschinen einen oder mehrere bzw. viele CPUs enthalten können und inspiriert von dem Begriff des Prozessors, wie Meyer ihn definiert, soll dieses Konzept als *virtueller Prozessor* bezeichnet werden.

6.1 Das Konzept des virtuellen Prozessors

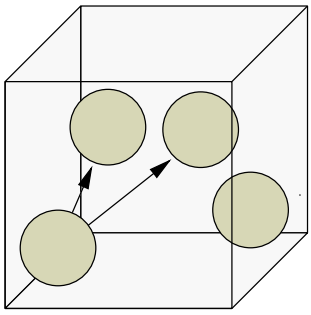
Ein virtueller Prozessor, wie er hier verstanden werden soll, ist nicht ein Stück Hardware oder ein Chip, auch nicht die Simulation oder ein Emulator einer CPU, sondern ein gewöhnliches Objekt. Doch dieses Objekt hat einige besondere Eigenschaften, die es unter anderen Objekten auszeichnet.

Ein virtueller Prozessor repräsentiert eine Ausführungseinheit. Er ist ein aktives Objekt, das also einen eigenen Kontrollfluss hat und andere Objekte verwaltet. Die Ausführung von Methoden (genauer gesagt von entfernt aufgerufenen Methoden, doch dazu später) wird vom virtuellen Prozessor verwaltet. Natürlich wird die Ausführung letztlich nach wie vor von der virtuellen Maschine bewerkstelligt, doch der virtuelle Prozessor liegt als Kontrollschicht zwischen der virtuellen Maschine und den Objekten. Innerhalb eines virtuellen Prozessors können Objekte genau so wie in einer virtuellen Maschine benutzt werden. Sie

können erzeugt, referenziert, aufgerufen und kopiert werden. Sie werden auch, wenn sie nicht mehr benötigt werden, vom Garbage Collector eingesammelt (der virtuelle Prozessor kümmert sich dabei zusätzlich um eine verteilte Garbage Collection).

Jedes Objekt ist in genau einem virtuellen Prozessor enthalten. Dadurch ist der Objektraum eindeutig aufgeteilt, so dass sich zwei Objekte entweder in dem gleichen oder in zwei verschiedenen virtuellen Prozessoren befinden. Der virtuelle Prozessor kann von einem Rechner zu einem anderen migriert werden (genau genommen kann er von einer virtuellen Maschine zu einer anderen verschoben werden, die sich durchaus auch auf demselben Rechner befinden können). Dabei wird er als Ganzes verschoben und alle in ihm enthaltenen Objekte migrieren mit. Hierfür ist es sehr wichtig, genau unterscheiden zu können, zu welcher Migrationseinheit (also zu welchem virtuellen Prozessor) ein Objekt gehört. In Abbildung 6.3 ist ein virtueller Prozessor dargestellt.

Abbildung 6.3: Der virtuelle Prozessor mit enthaltenen Objekten



Doch ehe der virtuelle Prozessor als vollständiges Konzept eingeführt wird, ist eine Middleware erforderlich, die eine Kommunikationsinfrastruktur zwischen den einzelnen virtuellen Maschinen eines verteilten Java-Systems zur Verfügung stellt. Sie muss auch einen Mechanismus zur Migration von Objekten bereithalten. Durch solch eine Schicht wird der physikalische Ort der virtuellen Maschinen weitgehend verdeckt. Eine Middleware, die dies zu leisten vermag, steht mit Voyager zur Verfügung. Es ist nicht die einzig denkbare Möglichkeit, aber derzeit diejenige, die die Anforderungen an eine solche Middleware am besten erfüllt. Voyager allein hat die in Kapitel 2.2.3 diskutierten Nachteile, doch als zugrundeliegende Infrastruktur für eine darüberliegende Schicht der virtuellen Prozessoren kann es sehr wertvolle Dienste leisten.

Passend zu dem Begriff der virtuellen Prozessoren soll diese Schicht *virtual backplane* heißen. Der Begriff der *backplane* wird normalerweise im Bereich der Hardware verwendet und bezeichnet eine Platine, die mehrere Steckplätze für CPUs bereitstellt. Davon soll hier abstrahiert werden, so dass eine Ebene entsteht, die sich über mehrere virtuelle Maschinen hinweg erstreckt und sozusagen Steckplätze für virtuelle Prozessoren bereithält. Damit eine virtuelle Maschine Teil einer *virtual backplane* wird, muss lediglich ein Voyager-Daemon gestartet werden und dessen IP-Adresse und Portnummer bekanntgegeben werden (Details im nächsten Kapitel). Anschließend können virtuelle Prozessoren auf dieser Schicht erzeugt und sozusagen »eingesteckt« werden (Abbildung 6.5).

Abbildung 6.4: Die *virtual backplane*, eine Schicht, die sich über mehrere VMs erstreckt.

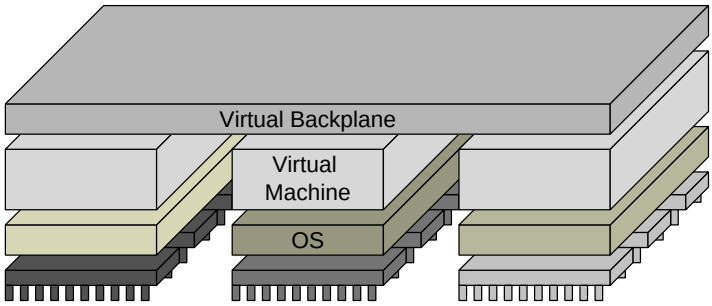
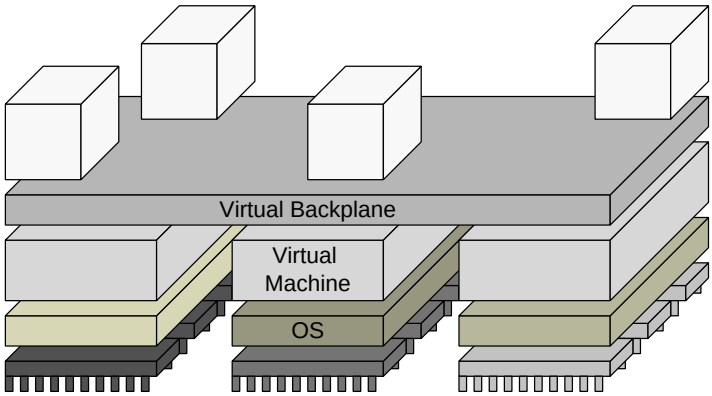


Abbildung 6.5: Virtuelle Prozessoren auf der virtual backplane

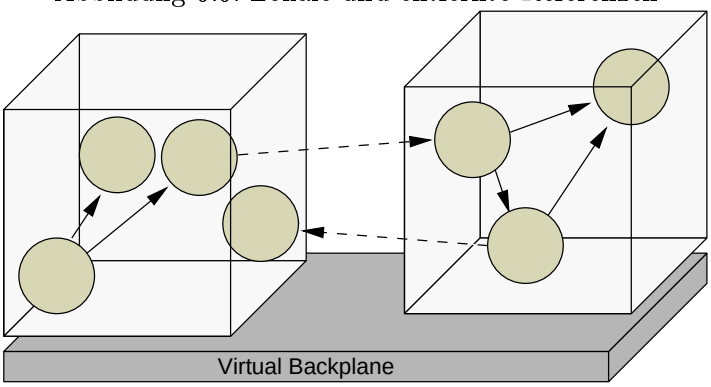


Die virtuellen Prozessoren enthalten also Objekte und können mitsamt dieser Objekte migriert werden. Die Objekte innerhalb eines virtuellen Prozessors können sich ganz normal referenzieren und aufrufen. Doch wie können Objekte in dem einen virtuellen Prozessor auf Objekte in einem anderen virtuellen Prozessor zugreifen? Hierfür soll eine besondere Sorte von Referenzen eingeführt werden, die »entfernte« Referenzen genannt werden sollen, im Gegensatz zu den gewöhnlichen, die nun auch als »lokale« Referenzen bezeichnen werden.

Über entfernte Referenzen lassen sich Objekte in anderen virtuellen Prozessoren (sie seien hier als entfernte virtuelle Prozessoren bezeichnet, auch wenn sie auf dem gleichen physikalischen Prozessor liegen können) wie gewohnt referenzieren. So lassen sich zum Beispiel Methoden eines entfernten Objektes aufrufen oder eine entfernte Referenz einer anderen Variable zuweisen.

Entfernte Referenzen müssen von den lokalen Referenzen unterscheidbar sein. Der Entwickler, ebenso wie ein Compiler oder ähnliche Werkzeuge, sollten jederzeit in der Lage sein, eine lokale von einer entfernten Referenz unterscheiden zu können. Dies kann durch verschiedene Mechanismen erfolgen. Man kann dies zum Beispiel erreichen, indem man solche Objekte von einem ausgezeichneten Interface oder einer speziellen Klasse ableitet. Bei RMI ist sogar beides nötig: das Erben vom Object RemoteObject und Implementieren des Interfaces Remote. Dies hat den Nachteil, dass nur die Fähigkeiten von Interfaces zur Verfügung stehen und zum Beispiel private Methoden nicht entfernt aufgerufen werden

Abbildung 6.6: Lokale und entfernte Referenzen



können, außerdem erfordert dies einen zusätzlichen Programmieraufwand beim Entwickeln einer Klasse. In Eiffel SCOOP wird das Voranstellen eines speziellen Schlüsselwortes (separate) vorgeschlagen. Dies erlaubt die Auszeichnungen einzelner Referenzen und erfordert keine Änderung der Klassen, doch müssen zusätzliche Regeln eingeführt werden, die zum Beispiel verhindern, dass entfernte Referenzen lokalen Variablen zugewiesen werden, was nicht erlaubt ist.

Statt dessen wird eine spezieller Typ für jede Klasse erzeugt, die entfernt benutzt werden soll. Diese Erzeugung kann, wie noch gezeigt wird, automatisch von einem Compiler geschehen, und der erzeugte Typ kann einen leicht modifizierten Namen im Vergleich zur Ursprungsklasse erhalten, zum Beispiel »Remote_A« für die Klasse A. Entfernte Referenzen ebenso wie lokale Referenzen haben einen Typ. Um den Unterschied zwischen entfernten und lokalen Referenzen zu dokumentieren, lassen sich diese beiden Typen verwenden: Wenn eine Referenz den normalen Typ trägt (A), ist es eine lokale Referenz, trägt sie den erzeugten Typ (Remote_A), ist es eine entfernte Referenz. In der im nächsten Kapitel vorgestellten Sprache – sie heißt Dejay – werden die erzeugten Typen, passend zum Namen der Sprache, das Präfix »Dj« tragen, also zum Beispiel DjA für die Klasse A. Hierdurch kann man von einer Klasse sowohl lokale als auch entfernte Referenzen erzeugen, ohne zusätzlichen Aufwand wie bei RMI betreiben zu müssen. Und man benötigt keine zusätzlichen Zuweisungsregeln, da die Typkonformitätsregeln von Java völlig ausreichen, und eine falsche Benutzung, wie etwa eine Zuweisung einer entfernten Referenz (vom Typ DjA) an eine Variable mit dem normalen Typ (A), ist nicht möglich, sondern kann schon zur Übersetzungszeit von einem Compiler erkannt werden.

6.2 Migration

Virtuelle Prozessoren können mitsamt der darin enthaltenen Objekte migriert werden. Dies ist immer dann möglich, wenn gerade keine Methode auf einem der enthaltenen Objekte ausgeführt wird. Damit der virtuelle Prozessor dies feststellen kann, werden alle Methodenaufrufe an Objekte, die in seiner Verwaltung stehen, an ihn umgeleitet. Er verwaltet diese Methodenaufrufe und leitet einen nach dem anderen an das ursprünglich aufgerufene Objekt weiter. So wird im-

mer nur eine Methode zur Zeit innerhalb des virtuellen Prozessors ausgeführt. Wenn der Befehl zur Migration eintrifft, wird dieser Aufruf ebenfalls in diese Warteschlange eingereiht, und wenn dieser Aufruf an der Reihe ist, dann ist sichergestellt, dass kein anderer Methodenaufruf innerhalb des virtuellen Prozessors ausgeführt wird. Dann kann der virtuelle Prozessor verpackt (serialisiert) werden, zu einem anderen Rechner übertragen und dort wieder ausgepackt (deserialisiert) werden. Anschließend können die Methodenaufrufe, die sich hinter dem Migrationsbefehl in der Warteschlage befanden, ausgeführt und neue Aufrufe entgegengenommen werden.

Entfernte Referenzen auf Objekte im gerade migrierten virtuellen Prozessor bleiben trotz der Migration erhalten. Dies lässt sich zum Beispiel so realisieren, dass ein Zugriff zunächst auf dem alten Standort versucht wird, der natürlich fehlschlägt, da das Objekt ja migriert ist. Doch die *virtual backplane* teilt dem Aufrufenden mit der Fehlermeldung den neuen Standort mit, so dass der Aufrufer dann einen neuen Aufruf zu diesem Standort versucht und das Objekt findet. Dies funktioniert sogar bei mehrfacher Migration eines virtuellen Prozessors hintereinander. Dies geschieht »unter der Decke« und bleibt dem Programmierer verborgen.

Man kann also den physikalischen Ort von virtuellen Prozessoren ändern, ohne dass sich die logische Struktur eines Programms, die durch die Referenzen aufgespannt wird, ändert. Um dies zu gewährleisten, dürfen virtuelle Prozessoren immer nur als ganzes verschoben werden und nicht etwa nur einzelne Objekte aus einem virtuellen Prozessor. Aber es ist möglich eine Kopie eines Objektes zu erstellen, und dies sogar entfernt, so dass zum Beispiel ein Objekt von einem virtuellen Prozessor in einen anderen kopiert wird. Das ist aber etwas anderes als die Migration (auch wenn es manchmal den gleichen Zweck erfüllt). Bei der Migration bleibt die Identität eines Objektes erhalten, beim Kopieren dagegen entsteht ein neues Objekt mit einer eigenen Identität. Die Migrationseinheit (oder die Migrationsgranularität) ist also stets der virtuelle Prozessor. Durch diesen Mechanismus werden die Probleme, die bei Voyager oder bei Emerald durch die Migration auftreten, vermieden. Bei Voyager können, wie in Kapitel 2.2.3 erläutert, ungewollt Kopien und dadurch Inkonsistenzen entstehen. Bei Emerald dagegen ist die Migrationsgranularität zu fein, so dass der Programmierer zusammengehörende Objekte durch relativ aufwendige Attachements zusammenhalten muss. Durch das Konzept der virtuellen Prozessoren ist jedes Objekt in genau einem virtuellen Prozessor enthalten und es entstehen keine Kopien durch eine Migration. Alle lokal referenzierten Objekte liegen gemeinsam in einem virtuellen Prozessor, so dass hierdurch ein einfacher Gruppierungsmechanismus gegeben ist.

6.3 Verteilung und Nebenläufigkeit

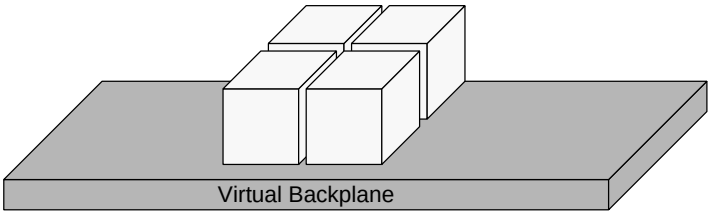
Durch die entfernten Referenzen steht ein sehr einfaches Konzept zur entfernten Kommunikation zur Verfügung. Objekte können über Rechnergrenzen hinweg angesprochen und aufgerufen werden, und zwar auf genau dieselbe Weise, wie es bei lokalen Objekten üblich ist: über Referenzen. Mit Hilfe der virtuellen

Prozessoren können Objekte auch migriert werden. Sie können sogar entfernt erzeugt werden: hierfür wird auf einem entfernten Rechner ein virtueller Prozessor erzeugt und in diesem dann ein normales Objekt angelegt, auf das man eine entfernte Referenz erhält.

Doch virtuelle Prozessoren sind auch ein Konzept, um Nebenläufigkeit auszudrücken. Virtuelle Prozessoren können nicht nur entfernt erzeugt werden, sondern auch auf dem gleichen Rechner. Zwei solche virtuelle Prozessoren sind dann nebenläufig zueinander und können gleichzeitig an zwei verschiedenen Aufgaben arbeiten. Man könnte sogar auf die Benutzung der in Java integrierten Threads verzichten, doch diese Diskussion soll hier nicht vertieft werden.

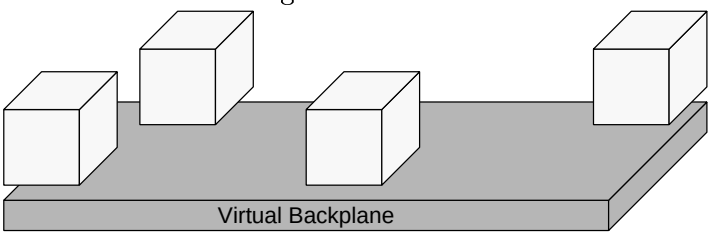
Viel interessanter ist, dass nun Verteilung und Nebenläufigkeit sich zu einem einzigen Konzept vereinen. Mehrere virtuelle Prozessoren, die gemeinsam in einer Anwendung zusammenarbeiten, können, wie Abbildung 6.7 zeigt, auf einem Rechner erzeugt werden und nebenläufig ihrer Arbeit nachgehen.

Abbildung 6.7: Virtuelle Prozessoren nebenläufig...



Zu einem beliebigen Zeitpunkt können sie dann, durch die Möglichkeit der Migration, auf einen anderen Rechner verschoben werden. Dies kann sogar im laufenden Betrieb geschehen, während die Objekte in den jeweiligen virtuellen Prozessoren sich gegenseitig aufrufen und die Anwendung vorantreiben. Dies ist in Abbildung 6.8 dargestellt.

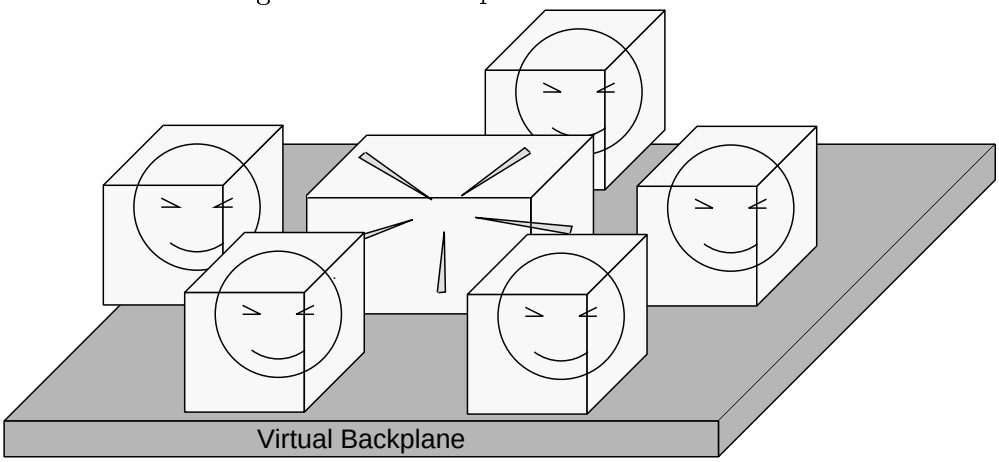
Abbildung 6.8: ... und verteilt.



Nach diesen Ausführungen soll nun noch einmal das Beispiel der dinierenden Philosophen betrachtet werden. Von diesen wurde gezeigt, dass sie entweder nebenläufig oder verteilt implementiert werden konnten, dass aber eine Änderung der einen Implementierungsart in die jeweils andere sehr aufwendig ist. Durch das Konzept der virtuellen Prozessoren ist es nun möglich, diese Anwendung so zu schreiben, dass sie entweder lokal und nebenläufig oder verteilt und echt parallel abläuft. Die Anwendung muss dafür nicht geändert werden. Es muss auch kein zusätzlicher Aufwand bei der Entwicklung getrieben werden, die Philosophen müssen lediglich in unterschiedlichen virtuellen Prozessoren erzeugt werden

und auf die ihnen zugeordneten Stäbchen über entfernte Referenzen verweisen (siehe Abbildung 6.9). Während der Entwicklung kann man völlig davon abstrahieren, ob dieses Programm später nebenläufig oder verteilt ausgeführt werden soll. Dies kann man dynamisch beim Starten der Anwendung entscheiden. Ja, es lässt sich sogar dynamisch zur Laufzeit ändern, indem die Philosophen einfach migriert werden.

Abbildung 6.9: Die Philosophen in virtuellen Prozessoren



Beide Möglichkeiten haben unterschiedliche Vorteile. Wenn die virtuellen Prozessoren möglichst nah zueinander bewegt werden, wird die Latenzzeiten erheblich reduziert und der Kommunikationsaufwand wird geringer. Wenn sich zwei virtuelle Prozessoren auf einer virtuellen Maschine befinden, findet die Kommunikation sogar über lokale Methodenaufrufe statt und nicht über das Netzwerk oder über Prozessgrenzen hinweg. Dafür benötigen aber beide die gleiche CPU, sie laufen zwar nebenläufig, aber nicht parallel. Wenn sie dagegen auf verschiedene Rechner bewegt werden, wird zwar die Kommunikation teurer, doch dafür wird die Last gleichmäßig auf mehrere Rechner verteilt und die Ausführung findet echt parallel statt.

6.4 Persistenz

Ein Problem, das der Migration in vielerlei Hinsicht ähnelt, ist die Persistenz. Bei der Persistenz werden Objekte zwar nicht verschoben, aber für einige Persistenzmechanismen müssen die Objekte, genau wie bei der Migration, serialisiert werden. Die serialisierte Form wird dann nicht über das Netz übertragen, sondern auf einer Festplatte gespeichert. Wenn diese Objekte wieder geladen werden, müssen sie, wie bei der Migration, wieder deserialisiert werden. Man könnte die Persistenz daher auch als eine Migration in der Zeit betrachten.

Aber es gibt noch mehr Ähnlichkeiten. Ein üblicher Ansatz bei objektorientierten Datenbanken besteht darin, ein Wurzelobjekt auszuzeichnen und alle von diesem Objekt über Referenzen erreichbaren Objekte gemeinsam zu speichern. Dadurch ist gewährleistet, dass alle Referenzen, die vor der Persistierung galten, auch nach dem Laden wieder ihre Gültigkeit haben. Man spricht auch

hier von referentieller Integrität. Die Granularität der Persistenz ist also die transitive Hülle des Wurzelobjektes. Das erinnert an die Migration bei Voyager (siehe Abschnitt 2.2.3), bei der ebenfalls die transitive Hülle eines Objektes migriert wird. Dabei muss man in beiden Fällen darauf achten, dass diese transitive Hülle nicht zu groß wird und Referenzen zu Objekten, die nicht mit einbezogen werden sollen, gekappt werden.

Ein weiteres Problem, das in beiden Fällen auftaucht, sind unerwünschte Kopien. Wenn ein Objekt (sei es als A bezeichnet) von zwei Wurzelobjekten (B und C) aus erreicht wird, also in der transitiven Hülle von zwei verschiedenen Objekten liegt, so wird es auch zwei Mal persistent gemacht. Angenommen, es wird erst eines der beiden Wurzelobjekte persistent gemacht. Dabei wird natürlich auch A abgespeichert. Es muss aber auch im Hauptspeicher verbleiben, damit das andere Wurzelobjekt es benutzten kann. Es existieren also zwei Kopien von A. Wird nun auch das andere Wurzelobjekt in die Datenbank gelegt und nach einer Weile beide wieder geladen, so werden auch zwei verschiedene Objekte für A angelegt. Dies lässt sich verhindern, indem ein gemeinsames Wurzelobjekt anstelle der zwei Objekte B und C verwendet wird, doch dadurch wird die transitive Hülle möglicherweise sehr groß und die Granularität der Persistenz wird sehr unhandlich. Man hätte also lieber einen Mechanismus, bei dem man die Granularität flexibler an die Anforderungen anpassen kann.

Wenn die Probleme so ähnlich sind, kann dann eine Lösung für die Migration auch eine Lösung für die Persistenz sein? Kann man die virtuellen Prozessoren als Mechanismus für die Speicherung von Objekten einsetzen? Dass dies möglich ist, soll im Folgenden gezeigt werden. Wenn man den virtuellen Prozessor als Wurzelobjekt an eine objektorientierte Datenbank übergibt, wird er zusammen mit den in ihm enthaltenen Objekten persistiert. Objekte, die in anderen virtuellen Prozessoren liegen, können über entfernte Referenzen verwaltet werden und müssen nicht verworfen werden. Die entfernten Objekte werden nicht mit in die Datenbank hineingezogen. Sie können mit dem sie umgebenden virtuellen Prozessor gespeichert werden. Da Objekte stets nur in einem virtuellen Prozessor enthalten sind, können keine ungewollten Kopien entstehen und Inkonsistenzen werden vermieden.

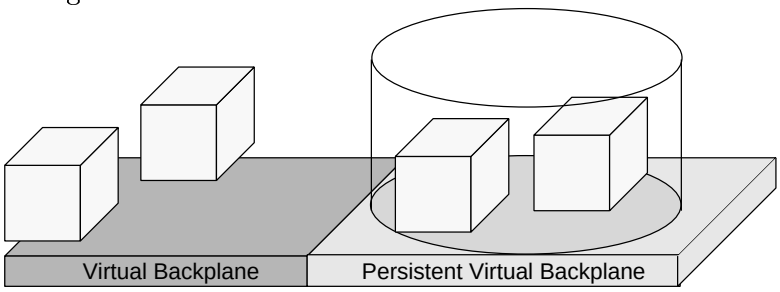
Und was geschieht mit den entfernten Referenzen? Ebenso wie bei der Migration behalten sie weiterhin ihre Gültigkeit. Wenn sowohl das referenzierte als auch das referenzierende Objekt wieder geladen sind, ist der Fall einfach: Die Referenzen können normal weiterverwendet werden. Etwas schwieriger wird es, wenn über eine entfernte Referenz auf ein Objekt zugegriffen wird, das nicht im Hauptspeicher, sondern in der Datenbank liegt. In diesem Fall kann der benötigte persistente virtuelle Prozessor automatisch wieder aktiviert werden und das Objekt steht wieder zur Verfügung. Dies geschieht ähnlich wie bei der Migration: Das referenzierende Objekt versucht das referenzierte Objekt aufzurufen. Da dies nicht verfügbar ist, sondern persistent, wird eine Ausnahme erzeugt. Durch eine geeignete Ausnahmebehandlung wird das persistente Objekt wieder aktiviert. Dann wird der Aufruf wiederholt und ist diesmal erfolgreich (siehe hierzu auch Abschnitt 4.3.2). Dies geschieht hinter den Kulissen und bleibt für den Programmierer unsichtbar.

Dadurch, dass durch dieses Konzept nur noch ein Objekttyp, nämlich der

virtuelle Prozessor, als Wurzelobjekt verwendet wird, lässt sich fast der gesamte Persistenzmechanismus hinter diesem Objekt verbergen, so dass der Programmierer nur noch den Namen der Datenbank bestimmen muss sowie den Zeitpunkt, an denen Objekte persistent gemacht werden sollen.

Datenbanken lassen sich somit in die *virtual backplane* integrieren und virtuelle Prozessoren können, je nach Bedarf, in den virtuellen Speicher verschoben werden, können wieder geladen oder automatisch reaktiviert werden. Dies zeigt Abbildung 6.10.

Abbildung 6.10: Virtuelle Prozessoren als Mechanismus für Persistenz



6.5 Zusammenfassung

Durch das in diesem Kapitel vorgestellte Konzept des virtuellen Prozessors ist eine Abstraktion gefunden, in der die Aspekte Nebenläufigkeit und Verteilung miteinander verschmelzen. Auf dieser Grundlage können alle in Abschnitt 5.3 aufgestellten Anforderungen erfüllt werden. Darüber hinaus ist dieses Konzept ebenfalls für die Persistenz ein geeigneter Mechanismus zur einfachen Speicherung und automatischen Aktivierung von Objekten und Objektgruppen. Die drei Aspekte verteilter Systeme – Nebenläufigkeit, Verteilung und Persistenz – lassen sich damit zu einem Konzept zusammenfassen. Hierdurch wird ein großes Potential zur Vereinfachung der Programmierung verteilter Systeme freigelegt, das es nun zu nutzen gilt. Im folgenden Kapitel wird eine Sprache vorgestellt, die dieses Konzept umsetzt.

Kapitel 7

Dejay: Eine verteilte Programmiersprache

Die im letzten Kapitel vorgestellten Konzepte sind im Rahmen dieser Arbeit in einer neu entwickelten, auf Java basierenden, Programmiersprache umgesetzt worden. Diese Programmiersprache trägt den Namen *Dejay* (gesprochen wie das englische Akronym für Disc Jockey – DJ) und steht für *Distributed Java.* Das Ziel, das mit Dejay verfolgt wird, ist die Entwicklung einer Programmiersprache, die die Programmierung von verteilten und nebenläufigen Anwendungen vereinfacht. Diese Sprache wurde im Rahmen dieser Arbeit entwickelt und ein voll einsatzfähiger Prototyp ist fertiggestellt. Die im Folgenden beschriebenen Sprachkonstrukte sind vollständig umgesetzt, doch dem Benutzer sollte klar sein, dass es sich noch um einen Prototypen handelt.

7.1 Ein einfaches Beispiel

Die Verwendung von Dejay soll zunächst mit einem einfachen Beispiel verdeutlicht werden. Dejay ist syntaktisch zu Java kompatibel, so dass Dejay sich wie Java liest und schreibt, doch die Aspekte Nebenläufigkeit, Verteilung und Persistenz werden durch semantische Erweiterungen diese Syntax erheblich einfacher. Die Umsetzung dieser zusätzlichen Semantik erfolgt durch einen Compiler. Dies erlaubt zwischen entfernten und lokalen Referenzen auf Ebene der Programmierung zu unterscheiden. Das zugrundeliegende Konzept ist der virtuelle Prozessor.

Das erste Beispiel in fast jeder Programmiersprache ist das so genannte HelloWorld-Programm. Und so auch hier. Die folgende Klasse bietet die Funktionalität, eine Klasse Hello zu erzeugen und eine Methode sayHello() aufzurufen, die einen Gruß auf den Bildschirm ausgibt.

```
// Hello.dj
public class Hello implements java.io.Serializable {

  public Hello() {
  }
```

```
  public void sayHello(java.lang.String name){
    System.out.println("Hello " + name);
  }
}
```

Diese Klasse ist ein legales Dejay-Programm. Es wäre auch ein legales Java-Programm, wie alle Dejay-Programme, die nur lokale Referenzen enthalten, doch es wird in der Datei Hello.dj anstatt in Hello.java abgespeichert. Anstelle des javac-Compilers wird der Compiler dejayc benutzt, der statt nur einer Ausgabedatei nun zwei erzeugt.

```
lin> dejayc Hello.dj
lin| Hello.class
lin| DjHello.class
```

Nun ist noch eine Klasse nötig, die ein Objekt von dieser Klasse erzeugen und aufrufen kann. Dafür soll eine Klasse Startup dienen, die wie folgt aussieht:

```
// Startup.dj
public class Startup {

  public static void main( String argv[] ) {
    Hello hello = new Hello();
    hello.sayHello( "Thorsten" );
  }
}
```

Auch diese Klasse wird übersetzen mit dem Compiler dejayc (allerdings, da diese Klasse nur lokal benötigen wird, mit dem Parameter -l, der die Erzeugung der Klasse DjStartup unterdrückt, näheres dazu im Abschnitt 7.9) und dann gestartet. Dejay-Programme erzeugen übrigens normalen Java-Byte-Code, so dass jeder Java-Interpreter sie ausführen kann.

```
lin> dejayc -l Startup.dj
lin| Startup.class

lin> java Startup
lin| Hello Thorsten
```

Nun soll dieses Programm verteilt ablaufen: Von dem Rechner lin aus soll ein Objekt auf dem Rechner sun erzeugt und dort aufgerufen werden und dort auch die Ausgabe machen. Nun kommt das Konzept der virtuellen Prozessoren und der entfernten Referenzen ins Spiel. Referenzen haben einen Typ, den sie bei ihrer Deklaration erhalten. Entfernten Referenzen ist ein Typ zugeordnet, dessen Name durch das Präfix Dj vor dem Namen der Klasse entsteht. Für die Klasse Hello zum Beispiel haben lokale Referenzen den Typ Hello und entfernte Referenzen den Typ DjHello.

Das Programm Startup wird folgendermaßen geändert: auf dem Rechner sun wird ein virtueller Prozessor erzeugt, wodurch eine entfernte Referenz (vom Typ DjProcessor) zurückgegeben wird. Dann wird in diesem Prozessor das Objekt Hello erzeugt und eine entfernte Referenz darauf (vom Typ DjHello) zurückgegeben. Diese Referenz kann ganz normal verwenden werden, doch es wird das entfernte Objekt aufgerufen und die Ausgabe erfolgt auf dem entfernten Rechner. Auf dem Rechner sun muss der Code für die Klasse Hello nicht vorhanden sein, es muss lediglich ein Voyager-Daemon gestartet werden (Details dazu in Abschnitt 7.10).

```
// Startup2.dj
public class Startup2 {
  public static void main( String argv[] ) {
     DjProcessor p1 = new DjProcessor("sun:8000");
     DjHello hello = new DjHello(p1);
     hello.sayHello( "Thorsten" );
  }
}

lin> dejayc -l Startup2.dj
lin| Startup2.class

sun> voyager 8000 -c http://lin:7042

lin> java Startup2

sun| Hello Thorsten
```

Nun soll dieses Objekt auch noch von einem Rechner auf einen anderen migriert werden, zum Beispiel auf den Rechner win und dort ebenfalls eine Ausgabe erzeugen. Das Programm wird noch einmal erweitert und auch auf dem Rechner win ein Voyager-Daemon gestartet:

```
// Startup3.dj
public class Startup3 {
  public static void main( String argv[] ) {
     DjProcessor p1 = new DjProcessor("sun:8000");
     DjHello hello = new DjHello(p1);
     hello.sayHello( "Thorsten" );

     hello.moveTo( "win:9000" );
     hello.sayHello( "Jan" );
  }
}

lin> dejayc -l Startup3.dj
lin| Startup3.class
```

```
sun> voyager 8000 -c http://lin:7042

win> voyager 9000 -c http://lin:7042

lin> java Startup3

sun| Hello Thorsten

win| Hello Jan
```

Damit kann dieses Programm Objekte entfernt erzeugen, entfernt referenzieren und aufrufen und Objekte migrieren lassen. Referenzen zu entfernten Objekten bleiben auch nach einer Migration erhalten (sonst könnte man die Methode sayHello() nicht ein zweites mal aufrufen), der nötige Code wird, wenn er nicht schon lokal vorhanden ist, automatisch migriert, und die verteilte Garbage Collection funktioniert ebenfalls.

7.2 Virtuelle Prozessoren

Die Grundlage für die Möglichkeiten von Dejay sind die virtuellen Prozessoren. Alle Objekte einer Dejay-Anwendung existieren in einem virtuellen Prozessor. Der erste virtuelle Prozessor wird implizit und ohne weiteres Zutun vom Programmierer bereits beim Start eines Dejay-Programms erzeugt. Daher ist dieses Konzept in einem lokal und sequentiell ablaufenden Programm, das ohne Verteilung auskommt, für den Programmierer transparent und jedes Java-Programm ist automatisch auch ein Dejay-Programm. Es müssen nur die Endungen angepasst und die Dateien neu übersetzt werden. Sobald aber Nebenläufigkeit oder Verteilung gewünscht oder benötigt wird, müssen explizit weitere virtuelle Prozessoren erzeugt werden.

Ein virtueller Prozessor ist im Prinzip ein ganz normales Objekt der Klasse Processor, das entfernt erzeugt werden kann (und noch einiges mehr, auf das erst später eingegangen wird). Es verwaltet Objekte, die sozusagen »in ihm« sind. Es wird, da es für andere Objekte ein entferntes Objekt ist, über eine entfernte Referenz verwaltet, die, konform zur Namenskonvention für entfernte Objekte in Dejay, vom Typ DjProcessor ist. Ein neuer virtueller Prozessor wird durch Anwendung des new-Operators auf den Typ DjProcessor instantiiert.

```
// erzeuge neuen virtuellen Prozessor
DjProcessor p = new DjProcessor("//sun:8000");
```

Das obige Beispiel erzeugt einen neuen virtuellen Prozessor auf dem Rechner sun auf Port 8000. Voraussetzung dafür ist ein laufender Voyager-Daemon auf diesem Port. Die Adresse des neuen virtuellen Prozessors kann eine beliebige gültige IP-Adresse (134.100.15.1) oder ein logischer Rechnername (sun.informatik.uni-hamburg.de oder innerhalb eines lokalen Netzes einfach sun) sein. Die Angabe eines Ports ist zwingend notwendig, da der anzusprechende Voyager-Daemon

nur auf dem entsprechenden Port erreichbar ist. Auf einem Rechner können durchaus mehrere solche Daemons gestartet werden, die dann untereinander auch mit localhost:<portnummer> angesprochen werden können. Zusätzlich zur Adresse sind noch folgende, optionale Parameter bei der Erzeugung eines neuen virtuellen Prozessors zugelassen:

- Ein Wert vom Typ boolean, mit dem die Ausgabe von Debug-Informationen gesteuert werden kann. Wird dieser Parameter nicht angegeben, werden keine Debug-Informationen ausgegeben.
- Ein Wert vom Typ String, der den Voyager-Daemon, auf dem der virtuelle Prozessor erzeugt wird, darüber informiert, auf welchem Rechner er nach fehlenden Klassen suchen soll. Auf dem angegebenen Rechner wird die Umgebungsvariable CLASSPATH dazu benutzt, die entsprechenden Klassen zu finden und nachzuladen. Der String muss eine gültige http-Adresse enthalten. Wird dieser Parameter nicht angegeben, wird localhost als Quelle für fehlende Klassen angenommen.

Beide Parameter sind optional und können beliebig kombiniert werden. Der boolean-Wert muss allerdings vor dem String-Wert angegeben werden.

```
// erzeuge neuen virtuellen Prozessor mit allen Parametern
DjProcessor p = new DjProcessor("//localhost:8000", true,
      "http://www.dejay.org/");
```

Der erzeugte virtuelle Prozessor liegt auf dem lokalen Rechner auf Port 8000 und soll Debug-Informationen ausgeben. Fehlende Klassen werden von Voyager automatisch auf dem Webserver www.dejay.org gesucht.

Der virtuelle Prozessor verfügt noch über einige Methoden, mit denen auf lokale Informationen zugegriffen werden kann.

- void printAddress(String) gibt die aktuelle Adresse des angegebenen virtuellen Prozessors auf der Konsole des Rechners aus, auf dem der virtuelle Prozessor liegt. Als Parameter wird der lokale Name des entfernten virtuellen Prozessors angegeben.
- String getAddress() liefert die aktuelle Adresse des virtuellen Prozessors zurück. Der Rückgabestring enthält dabei eine Adresse und Portnummer in der Form tcp://<Rechnername>:<Portnummer>.
- void printStorage() gibt eine Stringdarstellung der Referenzen, die im virtuellen Prozessor enthalten sind. Damit erhält man eine Aufstellung der Objekte, die zu diesem Prozessor gehören. Die Ausgabe erfolgt auf der Konsole des Rechners, auf dem sich der virtuelle Prozessor befindet.

Die Möglichkeiten, einen virtuellen Prozessor von einem Rechner zu einem anderen zu bewegen, bereits existierende Objekte aufzurufen und einen virtuellen Prozessor persistent zu machen, werden in den Abschnitten 7.4, 7.5 und 7.6 detailliert beschrieben. Doch zunächst wird auf die Erzeugung und Verwendung entfernter Objekte eingegangen.

7.3 Entfernte Objekte

Objekte können in Dejay entweder lokal oder entfernt erzeugt werden. Die Erzeugung eines lokalen Objektes geschieht genau wie in Java durch den new-Operator. Für die entfernte Erzeugung eines neuen Objektes wird ebenfalls der new-Operator benutzt, allerdings auf einen anderen Typ. Für die Erzeugung eines Objektes vom Typ A wird der Typ mit dem Präfix Dj, also DjA verwendet. Alle Konstruktoren von A stehen auch für DjA zur Verfügung, und die Parameter dieser Konstruktoren bleiben gleich, werden aber um einen Parameter ergänzt, der bestimmt, in welchem virtuellen Prozessor das Objekt angelegt werden soll.

```
DjA a1 = new DjA ( <Parameter1, ..., ParameterN,> DjProcessor );
```

Als Ergebnis erhält man eine entfernte Referenz auf das neu erzeugte entfernte Objekt. Der Mechanismus der entfernten Referenzen ist durch Proxies implementiert, so dass man genau genommen nicht eine entfernte Referenz, sondern eine lokale Referenz auf einen automatisch generierten Proxy erhält, der die Funktionalität der entfernten Benutzung zur Verfügung stellt. Dieser Mechanismus wird in Dejay weitestmöglich verborgen, so dass der Programmierer tatsächlich das Gefühl hat, mit entfernten Referenzen umzugehen. Nur an wenigen Stellen treten diese Proxies zu Tage und erfordern eine andere Behandlung als tatsächliche Referenzen. Ein Beispiel hierfür ist der Vergleich von Referenzen. Mit dem Vergleich zweier entfernter Referenzen, zum Beispiel

```
DjA a2 = a1;
if (a1 == a2) ...
```

wird also geprüft, ob beide Referenzen lokal auf denselben Proxy zeigen, nicht, ob sie auf dasselbe entfernte Objekt zeigen. Da auch diese Überprüfung nützlich sein kann, wurde sie beibehalten. Um die inhaltliche *Gleichheit* von zwei unterschiedlichen entfernten Objekten zu prüfen, wurde die Methode equals() überladen, so dass diese entfernt wirkt. Für die Überprüfung, ob zwei entfernte Referenzen auf *dasselbe* entfernte Objekt verweisen, steht die Methode isRemoteIdentical() zur Verfügung:

```
if ( a1.isRemoteIdentical(a2) ) ...
```

Um zwei unterschiedliche Referenzen auf dasselbe entfernte Objekt zu erhalten, also zwei unterschiedliche Proxies und nicht zwei lokale Referenzen auf ein Proxy, existiert ein weiterer Konstruktor (der Copy-Konstruktor), der als Parameter eine entfernte Referenz auf ein Objekt erhält.

```
DjA a2 = new DjA ( a1 );
```

Nachdem entfernte Objekte erzeugt wurden, lassen diese sich auch benutzen. Die Beispiele des vorigen Abschnitts haben bereits erste einfache Aufrufe von Methoden an lokalen und entfernten Objekten gezeigt. Einen Unterschied zwischen dem Aufruf von sayHello() auf dem lokalen und dem entfernten Objekt gibt es weder in der Aufrufsyntax noch in den Ausgaben. Lediglich der Ort der Ausgaben ist unterschiedlich.

```
Hello hello_local = new Hello();
hello_local.sayHello( "Toby" );

DjHello hello_remote = new DjHello(p1);
hello_remote.sayHello( "Thorsten" );
```

Auch Parameter können beim Aufruf von Methoden wie gewohnt übergeben werden. Wenn eine lokale Referenz auf einen Parameter angegeben wird, so wird dieser Parameter in den Adressraum des entfernten Objektes kopiert. Bestehende Methoden müssen hierfür nicht geändert werden, so dass zum Beispiel alle Objekte und ihre Methoden aus dem JDK oder aus bestehenden Projekten einfach verwendet werden können, ohne sie zu ändern oder sie neu kompilieren zu müssen. Übergibt man eine entfernte Referenz, wird auf diesen Parameter entfernt zugegriffen. Eine entfernte Referenz auf ein lokales Objekt zu erhalten ist allerdings nicht möglich.

Als Rückgabewert bei entfernten Aufrufen wird, wenn möglich, eine entfernte Referenz auf das entfernt vorliegende Ergebnis zurückgeliefert. Dies funktioniert genau dann, wenn für den Rückgabewert der entfernte Typ existiert und der zugehörige Code vom umgebenden virtuellen Prozessor erreichbar ist. Der entfernte Typ, zum Beispiel DjA zur Klasse A, wird vom dejayc-Compiler erzeugt. Der entfernte Typ kann auch für eine in Bytecode vorliegende Klasse erzeugt werden, so dass die Klassendefinition nicht im Source vorliegen muss, um sie entfernt zu benutzen. Findet der virtuelle Prozessor den entfernten Typ, liefert er eine entfernte Referenz auf das Ergebnisobjekt zurück.

Gibt es keine entsprechende Proxy-Klasse, wird eine Kopie des Ergebnisobjektes zurückgegeben. Dies ist genau dann möglich, wenn alle betroffenen Klassen serialisierbar sind, was für fast alle Standardklassen zutrifft. So wird zum Beispiel für eine Methode, die einen String zurückliefert, eine Kopie des Strings zurückgegeben, die dann lokal beim Aufrufenden bearbeitet werden kann. Auch die eigenen Klassen sollten stets serialisierbar sein, was durch den Zusatz implements java.io.Serializable erreicht wird.

Auch wenn es häufig wünschenswert ist, eine entfernte Referenz auf ein Objekt zu erhalten, wird man gelegentlich eine lokale Kopie dieses Objektes erstellen wollen. Hierfür existiert für jede entfernte Referenz eine Methode getCopy(), die dies ermöglicht.

```
// Erzeugen einer lokalen Kopie eines entfernten Objektes
DjA a_remote = new DjA(p1);
A a_local_copy = a_remote.getCopy();
```

Dejay unterscheidet drei Arten von Aufrufsemantiken: synchrone, asynchrone und Einwegaufrufe. Bei synchronen Aufrufen wartet das rufende Objekt so lange, bis das gerufene Objekt das Ergebnis zurückgeliefert hat, wie es bei lokalen Aufrufen immer der Fall ist. Bei asynchronen Aufrufen wartet das rufende Objekt nur auf eine Bestätigung, dass der Aufruf beim gerufenen Objekt angekommen ist. Anschließend kehrt der Kontrollfluss zurück und das Programm läuft weiter. Wenn das Ergebnis dieses Aufrufes benötigt wird, wenn also auf

dieses Objekt Methoden aufgerufen werden, zum Beispiel um Ergebniswerte auszulesen, wird überprüft, ob das Ergebnis bereits vorliegt oder noch nicht. Ist das Ergebnis zu diesem Zeitpunkt noch nicht eingetroffen, wird an dieser Stelle blockierend gewartet. Diese Semantik wird als »wait-by-neccessity« bezeichnet.

Bei Einwegaufrufen wird nur der Aufruf abgeschickt, aber überhaupt nicht auf eine Rückmeldung oder ein Ergebnis gewartet. Diese Aufrufart ist daher nur sinnvoll, wenn kein Rückgabewert erwartet wird. Von der weiteren Bearbeitung des Aufrufs bekommt das rufende Objekt nichts mehr mit. Das gilt insbesondere auch für den Fall, dass der Aufruf nicht ordnungsgemäß weitergeleitet werden konnte, z.B. aufgrund von Netzwerkproblemen. Man spricht daher auch von einer »fire-and-forget«-Semanik. Dafür entstehen für das aufrufende Objekt keinerlei Wartezeiten, so dass dieser Aufruf zwar nicht sehr sicher aber sehr schnell ist und für einige Fälle sehr sinnvoll sein kann.

Die Aufrufsemantik eines Methodenaufrufes kann durch einen zusätzlichen Parameter angegeben werden. Standardmäßig verwendet Dejay synchrone Aufrufe, die daher nicht weiter gekennzeichnet werden müssen. Für asynchrone oder Einwegaufrufe wird eine Konstante aus der Klasse Dejay als zusätzlicher Parameter angegeben. Solch eine Konstante existiert auch für den synchronen Aufruf, sie muss aber nicht angegeben werden.

```
DjA a = new DjA("lin:8000");

// synchroner Aufruf
B b1 = a.method1( 42 );
if (b1 instanceOf DjB)
  DjB b2 = (DjB) b1;

// asynchroner Aufruf
DjB b3 = a.method1( 42, Dejay.ASYNC );

// Einwegaufruf
a.method1( 42, Dejay.ONEWAY );
```

Asynchrone und Einwegaufrufe verursachen eine nebenläufige Ausführung. Bei asynchronen Methodenaufrufen kann daher das Ergebnis eher angefordert werden, als es tatsächlich bereitsteht. Normalerweise wird bei der ersten Benutzung des Ergebnisses blockierend gewartet. Es gibt aber Mechanismen, um flexibler und kontrollierter auf diese Situation zu reagieren. Das Ergebnis eines asynchronen Aufrufs muss daher als entfernte Referenz zurückgeliefert werden. Durch diese ist es dann möglich, nach dem Vorhandensein des Ergebnisses zu fragen, was man als *Polling* bezeichnet, oder an einer definierten Stelle zu warten, ohne das Ergebnis selbst zu benutzen. Dafür stellen entfernte Referenztypen die Methoden isAvailable() und waitForResult() zur Verfügung. isAvailable() liefert true zurück, wenn das Ergebnis vorliegt, sonst false. Damit kann dynamisch entschieden werden, ob das Ergebnis verarbeitet oder in der Wartezeit etwas anderes bearbeitet werden soll. Mit waitForResult() kann explizit auf das Ergebnis gewartet werden. Es liefert immer true zurück, blockiert aber so lange

bis das Ergebnis vorliegt. Die folgenden zwei Beispiele sind vom Verhalten her äquivalent. Bei beiden wird ein asynchroner entfernter Aufruf ausgelöst und auf das Ergebnis gewartet. Während das erste Beispiel sehr einfach zu schreiben ist, kann bei der zweiten Version der Kontrollfluss sehr viel flexibler gesteuert werden.

```
// Beispiel für wait-by-necessity
DjB remote_result = a.method1( 42, Dejay.ASYNC);
remote_result.method2();

// Beispiel für Polling und explizites Warten
DjB remote_result = a.method1( 42, Dejay.ASYNC);
if ( remote_result.isAvailable() ) {
  remote_result.method2();
} else {
  remote_result.waitForResult();
  remote_result.method2();
}
```

Verschiedene virtuelle Prozessoren sind zueinander nebenläufig, da jeder virtuelle Prozessor seinen eigenen Kontrollfluss in sich trägt. Durch die Verwendung von mehreren virtuellen Prozessoren besteht die Möglichkeit, Dinge nebenläufig oder parallel auszuführen. Durch den asynchronen Aufruf wird diese Möglichkeit genutzt. Damit wird in Dejay auf einfache Weise Nebenläufigkeit nutzbar gemacht. Dabei ist es aus Sicht des Dejay-Programms unerheblich, ob es sich um scheinbare Nebenläufigkeit oder echte Parallelität handelt.

7.4 Migration

Der vorige Abschnitt hat gezeigt, wie man in Dejay entfernte Objekte erzeugen und aufrufen und wie man mit Hilfe von virtuellen Prozessoren Nebenläufigkeit ausdrücken kann. Damit lassen sich bereits sehr einfach verteilte Anwendungen programmieren. Dabei ist bisher davon ausgegangen worden, dass entfernte Objekte und virtuelle Prozessoren an einer bestimmten Stelle erzeugt werden und während der Laufzeit eines Programms an dieser Stelle bleiben. Doch in verteilten Systemen ist es oft wünschenswert, dass Objekte ihren Ort ändern. Dies wird als Objektmigration bezeichnet und wurde schon in den Abschnitten zu Voyager und zu Emerald diskutiert. Doch bei beiden Ansätzen traten Probleme auf, die vor allem mit der Migrationsgranularität begründet sind.

In Dejay ist die Einheit der Migration der virtuelle Prozessor. Das heißt, ein virtueller Prozessor ist migrierbar und nimmt dabei alle in ihm enthaltenen Objekte mit sich. So bleiben alle lokalen Referenzen der in ihm enthaltenen Objekte gültig. Und auch die entfernten Referenzen behalten ihre Gültigkeit, wie gezeigt werden wird.

Um eine Migration zu veranlassen, steht für entfernte Referenztypen eine Methode moveTo() zur Verfügung. Sie kann entweder bei einem virtuellen Prozessor aufgerufen werden oder bei einem beliebigen (entfernt referenzierten) Ob-

jekt in diesem. In beiden Fällen migriert der gesamte virtuelle Prozessor mit allen enthaltenen Objekten. Die Methode moveTo() existiert in mehreren Versionen, die mit jeweils verschiedenen Parametern aufgerufen werden können.

- moveTo(String) führt zu einer Bewegung an die im Parameter angegebene Adresse inklusive Portnummer. Die Adresse muss eine gültige IP-Adresse oder einen gültigen logischen Rechnernamen enthalten. Unter der angegebenen Adresse muss bereits ein Voyager-Daemon laufen.
- moveTo(DjProcessor) löst eine Bewegung zu der Adresse aus, unter der der angegebene virtuelle Prozessor erreicht werden kann.
- moveTo(DjObject) führt eine Bewegung zu der Adresse aus, unter der der virtuelle Prozessor, der das angegebene Objekt enthält, erreicht werden kann.

Damit ergeben sich eine Reihe von Möglichkeiten, um dasselbe zu erreichen. Hier seien ein paar Beispiele aufgeführt. Dabei wird angenommen, dass jeweils ein Voyager-Daemon auf dem lokalen Rechner (auf Port 7000 und 8000) und auf dem Rechner sun (Port 8000 und 9000) gestartet sind.

```
// verschiedene Aufrufmöglichkeiten von moveTo()
DjProcessor p1 = new DjProcessor( "//localhost:8000" );
DjProcessor p2 = new DjProcessor( "tcp://sun.uni-hamburg.de:7000" );
DjHello hello1 = new DjHello( p1 );
DjHello hello2 = new DjHello( p2 );

// bewege p1 von Port 8000 nach Port 9000
p1.moveTo( "//localhost:9000" );

// bewege p2 von Port 7000 nach Port 8000
hello2.moveTo( "tcp://sun.uni-hamburg.de:8000" );

// bewege p1 nach tcp://sun.uni-hamburg.de:8000
hello1.moveTo( p2 );

// bewege p2 von tcp://sun.uni-hamburg.de nach localhost
p2.moveTo( "//localhost:8000" );

// bewege p1 von tcp://sun.uni-hamburg.de nach localhost
hello1.moveTo( hello2 );
```

Einzelne Objekte können nicht migriert werden. Statt dessen kann man eine lokale Kopie von entfernten Objekten erstellen, bei der nicht nur das einzelne Objekt, sondern dessen transitive Hülle kopiert wird. Die kopierten Objekte haben anschließend eine eigene Identität und Referenzen auf das Original zeigen weiterhin auf das Original. Das Kopieren eines Objektes kann sehr sinnvoll sein, um teure entfernte Aufrufe zu vermeiden, insbesondere dann, wenn auf diese Objekte ohnehin nur lesend zugegriffen werden soll.

```
Hello copy_of_hello1 = hello1.getCopy();
```

7.5 Namensdienst

In den meisten Kommunikationsmechanismen wie RMI oder CORBA ist es nicht möglich, Objekte entfernt zu erzeugen. Statt dessen müssen Objekte auf einem entfernten Rechner erst erzeugt werden, anschließend über einen geeigneten Mechanismus bekannt gemacht und kontaktiert werden. Ein hierfür häufig verwendeter Mechanismus ist der Namensdienst. In Dejay ist nicht nur die entfernte Erzeugung möglich, sondern auch die Verbindung zu existierenden Objekten. Dejay bietet ebenfalls einen Namensdienst. Dafür sind verschiedene Realisierungen möglich, zum Beispiel ein netzweiter Namensdienst, wie ihn CORBA oder Jini ermöglichen, oder als rechnerbezogener Dienst, der nur Objekte auf einem Rechner (bzw. für eine virtuelle Maschine) verwalten kann, wie bei RMI und Voyager.

Die bisher in Dejay realisierte Variante ist zweistufig. Da Objekte in Dejay stets von einem virtuellen Prozessor umgeben sind, verwaltet eine erste Stufe des Namensdienstes nur virtuelle Prozessoren, und zwar derzeit begrenzt auf die virtuellen Prozessoren, die auf einem Voyager-Daemon laufen. Einem virtuellen Prozessor kann damit ein Name erteilt werden, unter dem er beim lokalen Namensdienst angemeldet und bei Bedarf wieder abgerufen werden kann. Das Anmelden und Abrufen ist jeweils entfernt möglich. Zum Anmelden steht sowohl für entfernte Referenzen als auch für lokale Referenzen auf einen virtuellen Prozessor die Methode registerByName() zur Verfügung.

Innerhalb eines virtuellen Prozessors gibt es eine zweite Stufe des Namensdienstes, über die die enthaltenen Objekte intern an einen Namen gebunden werden können. Dafür dient ebenfalls eine Methode registerByName(). Eine Anmeldung kann also etwa folgendermaßen aussehen:

```
DjProcessor p1 = new DjProcessor("lin:8000");
DjA a1 = new DjA(p1);
a1.registerByName("DemoA");
p1.registerByName("DemoProcessor");
```

Von einem anderen Rechner aus kann nun über den Namensdienst auf diesen virtuellen Prozessor und anschließend auf das Objekt A zugegriffen werden. Dafür bieten einerseits die Klasse Dejay die statische Methode getProcessorByName() und der virtuelle Prozessor die Methode getObjectByName().

```
DjProcessor q1 = Dejay.getProcessorByName("DemoProcessor","lin:8000");
DjA a2 = (DjA) q1.getObjectByName("DemoA");
```

Die Referenzen a1 und a2 zeigen nun von unterschiedlichen Rechnern aus auf dasselbe Objekt der Klasse A auf dem Rechner lin:8000.

7.6 Persistenz

Objekte können in Dejay auf einfache Weise persistiert werden. Dabei wird, wie auch bei der Migration, stets der gesamte virtuelle Prozessor mit allen in ihm enthaltenen Objekten gemeinsam persistiert. Dadurch bleiben lokale Referenzen immer gültig, da alle lokal referenzierten Objekte sich immer entweder gemeinsam im Hauptspeicher oder im persistenten Speicher befinden. Aber auch entfernte Referenzen behalten ihre Gültigkeit. Wenn ein entfernt referenziertes Objekt nicht im Hauptspeicher ist, aber persistiert wurde, wird es automatisch wieder aktiviert. Man spricht hier von Activation. Sowohl der in Dejay verwendete Speichermechanismus als auch der Aktivierungsmechanismus basieren auf Voyager.

Um einen virtuellen Prozessor persistent machen zu können, muss er darauf vorbereitet sein und einen Namen (ein Memento) und den Namen einer Datenbank erhalten. Dies geschieht schon bei der Erzeugung durch einen speziellen Konstruktor. Anschließend kann sein Inhalt durch den Aufruf der Methode persist() in dieser Datenbank festgehalten werden. Mit der Methode isPersistable() kann überprüft werden, ob ein virtueller Prozessor entsprechend erzeugt oder vorbereitet wurde.

```
DjProcessor p1 = new DjProcessor("PersistProc","ProcessorDB","lin:8000");
DjA a1 = new DjA();
if ( p1.isPersistable ) {
  p1.persist();
  p1.flush();
}
```

Nach der Persistierung verbleibt der Prozessor im Hauptspeicher, und es kann weiterhin auf seine Objekte zugegriffen werden. Doch häufig wird die Persistenz gerade dafür eingesetzt, den Hauptspeicher von selten benötigten Objekten zu befreien. Durch die Methode flush() kann der virtuelle Prozessor für die automatische Speicherbereinigung freigegeben werden, so dass der von ihm belegte Speicherbedarf frei werden kann.

Wenn ein entferntes Objekt nun auf ein Objekt in einem so behandelten virtuellen Prozessor zugreift, ist das Objekt entweder immer noch im Hauptspeicher und kann direkt genutzt werden, oder es ist nicht mehr im Hauptspeicher, dafür aber in der Datenbank, dann wird es automatisch wieder reaktiviert und steht nach einer kurzen Verzögerung wieder bereit. Dies bedeutet in dem obigen Beispiel, dass die Referenz a1 zu einem gegebenen Zeitpunkt wieder verwendet werden kann, auch wenn das entsprechende Objekt längst nicht mehr im Hauptspeicher ist. Dies ist sogar möglich, wenn die virtuelle Maschine oder sogar der Rechner zwischenzeitlich heruntergefahren wird und lediglich die *virtual backplane* wieder aufgesetzt wird. Um dabei die Aktivierung neu zu ermöglichen, reicht allerdings das einfache Starten des Voyager-Daemons nicht, sondern es muss statt dessen die Klasse Backplane gestartet werden.

Eine andere Möglichkeit ähnelt dem Namensdienst. Nach der Persistierung können die entfernten Referenzen auf einen virtuellen Prozessor verworfen wer-

den. Zu einem späteren Zeitpunkt kann der virtuelle Prozessor durch die Methode getProcessorFromDB() der Klasse Dejay per Namen wieder aus der Datenbank erweckt werden. Objekte innerhalb des virtuellen Prozessors sind dann über den normalen internen Namensdienst wieder zu erreichen.

```
DjProcessor q1 = Dejay.getProcessorFromDB("PersistProc","ProcessorDB",
      "lin:8000");
```

7.7 Komposition virtueller Prozessoren

Durch das Konzept des virtuellen Prozessors werden Objekte zu Gruppen zusammen gefasst, die unter Aspekten der Verteilung, Nebenläufigkeit, und Persistenz eine funktionale Einheit bilden. Die Objekte dieser Einheit verhalten sich bezüglich der Verteilung ko-lokal, weisen bezüglich der Nebenläufigkeit einen gemeinsamen untereinander geteilten Ausführungkontrollfluss auf und werden gemeinsam persistent gemacht. Dieser Mechanismus wirkt als Gruppierungsmechanismus auf Ebene der Objekte. Dies wirft aber die Frage auf, ob eine Notwendigkeit für einen ähnlichen Gruppierungsmechanismus auch auf der Meta-Ebene, der Ebene der virtuellen Prozessoren, notwendig ist und wie ein solcher aussehen könnte. Dieser Frage soll in diesem Abschnitt nachgegangen werden. Der Fokus wird dabei auf den Aspekt der Verteilung gelegt. Für andere Aspekte, insbesondere für die Persistenz sind die Ergebnisse aber übertragbar.

Die Problemstellung aus Sicht der Verteilung besteht in der räumlichen Zusammengehörigkeit von Objekten, die sich insbesondere durch die Möglichkeit der Migration ergibt. Die Zusammengehörigkeit von Objekten wird durch das Konzept der virtuellen Prozessoren statisch ausgedrückt. Solch eine Objektgruppe ist fest miteinander verbunden. Erfordert eine Anwendung eine gewisse Flexibilität dieser Gruppierung, so ist ein Mechanismus auf höherer Ebene nötig.

Als einfaches Beispiel sei hier die Modellierung eines Autos genannt. Zwar wird das Auto mit seinem Motorblock die meiste Zeit eine Einheit bilden, so dass ihr Abbild in einem Softwaremodell die Gruppierung von Auto und Motorblock in einer gemeinsamen virtuellen Prozessor vorsehen mag. Dies macht unter dem Aspekt der Migration Sinn, da das Auto typischerweise zusammen mit seinem Motorblock migriert werden soll. Doch wenn die Anwendung eine Trennung von Auto und Motorblock notwendig macht, etwa zur Reparatur oder zum Auswechseln des Motors, so ist es sinnvoller beide in getrennten virtuellen Prozessoren zu gruppieren. Dies erfordert allerdings die Migration beider Gruppen im normalen Fall, es sei denn es liegt ein Mechanismus zur Komposition von virtuellen Prozessoren vor.

Ein solcher Mechanismus erleichtert auch die Modellierung in solchen Fällen, in denen die Aspekte Verteilung, Nebenläufigkeit und Persistenz für eine jeweils andere Granularität sprechen würden. Durch einen Gruppierungsmechanismus auf höherer Ebene kann die jeweils kleinste geforderte Granularität gewählt werden und für die anderen Aspekte durch die Meta-Gruppierung wieder zusammengefasst werden.

Ein Mechanismus, der dies erlaubt, wurde für die Sprache Dejay entwickelt und soll im Folgenden vorgestellt werden. Dynamische Gruppierungskonzepte

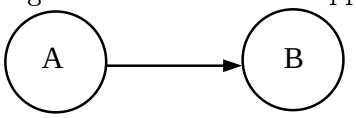
auf Ebene der Objekte wurden bereits in den Projekten Emerald [Black et al. 1986], [Jul et al. 1987], [Hutchinson 1987] und FarGo [Holder et al. 1999] untersucht und implementiert. Doch auf Ebene von Objekten scheint eine dynamische Gruppierung zu feingranular und der nötige Aufwand zur Steuerung und Verwaltung dieser Dynamik ist auf dieser Ebene kaum zu rechtfertigen. Doch nach einer Reduzierung der Granularität durch eine Gruppierung, wie durch das Konzept der virtuellen Prozesse, scheint eine Gruppierung oder Komposition solcher statischer Objektgruppen sinnvoll. Dadurch entsteht der Aufwand zur Handhabung der Dynamik nur dort, wo er wirklich benötigt wird. Daher wurden in Dejay Mechanismen der dynamischen Gruppierung von der Ebene der Objekte auf die Ebene der Gruppen übertragen.

Die Komposition von Objektgruppen hat Auswirkungen auf die Migration und auf die Persistenz. Durch eine geeignete Komposition lässt sich ausdrücken, wie sich andere Objektgruppen bei Migration oder Speicherung einer Gruppe verhalten sollen. Um dieses Verhalten differenziert beschreiben zu können sind verschiedene Kompositionsbeziehungen nötig, die im Folgenden vorgestellt werden.

Unidirektionale Komposition

Die einfachste Gruppierungsart ist die einseitige Zuordnung eines Prozessors B zu einem Prozessor A. Dies bedeutet, dass Prozessor B sich immer auf der logischen Umgebung aufhalten muss, auf der Prozessor A sich gerade befindet. Daraus folgt natürlich direkt, dass auch A sich immer bei B befindet. Erst bei der Migration wird die Einseitigkeit erkennbar: Prozessor A darf bewegt werden, und zieht Prozessor B mit sich. Prozessor B kann dagegen nicht bewegt werden - jeder Versuch, dies zu tun, führt zu einer Laufzeit-Exception.

Abbildung 7.1: Einfache Uni-Gruppierung



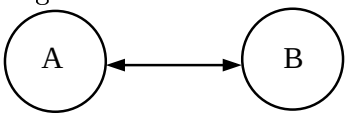
In der graphischen Darstellung dieser Gruppierung (Abbildung 7.1) zeigt der Pfeil von A nach B. Umgangssprachlich bedeutet dies: "A zieht B mit sich." Diese Art der Verknüpfung ist transitiv. Es ist daher möglich, eine ganze Reihe von Prozessoren zu verketten. Allerdings kann nur derjenige Prozessor bewegt werden, der nicht selbst an einen anderen Prozessor gebunden wird. Diese Art der Komposition wird im Folgenden als UniPull-Relokator bezeichnet. Die Bewegung dieses "ersten" Prozessors zieht die gesamte Kette der verknüpften Prozessoren mit sich. Es ist nicht möglich einen Prozessor gleichzeitig an zwei unterschiedliche Prozessoren per UniPull zu binden.

Symetrische Komposition

Um eine beidseitige Verknüpfung zweier Prozessoren zu erreichen, ist eine weitere Kompositionsart nötig, die symetrisch ist. Diese Kompositionsart soll hier

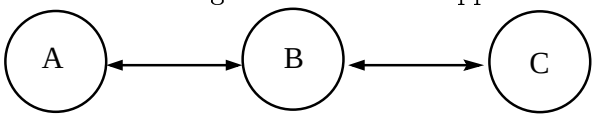
als BiPull-Relokator bezeichnet werden. Wie bei UniPull müssen sich die beiden virtuellen Prozessoren in der gleichen Laufzeitumgebung aufhalten, doch im Gegensatz zu UniPull sind die Prozessoren nun "gleichberechtigt". Wenn einer der beiden Prozessoren migriert, zieht er den jeweils anderen mit sich, soweit andere Einschränkungen dies nicht verhindern.

Abbildung 7.2: Einfache Bi-Verknüpfung



Diese Kompositionsart eignet sich, um größere Verbände von Prozessoren zu bilden (siehe Abbildung 7.3). Bei der Migration verhalten sich die Objekte in den Prozessoren so, als ob sie sich alle im selben Prozessor befänden - migriert ein Objekt, werden alle damit verknüpften Prozessoren mitmigriert.

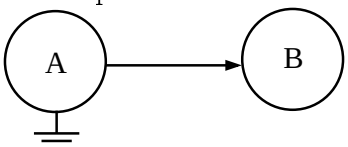
Abbildung 7.3: Einfache Gruppe



Fixierung

In einem System, in dem Migration von äußeren Umständen abhängt, kann bei der Entwicklung nicht vorhergesagt werden, wann ein Prozessor sich auf einer bestimmten Umgebung aufhält, beziehungsweise wann andere Objekte die Migration dieses Prozessors anstoßen werden. Daher muss es möglich sein, einen Ortswechsel durch Migration zu unterbinden. Dieses Verbot muss sowohl die aktive Aufforderung zur Migration durch ein enthaltenes oder anderes Objekt als auch das passive Migrieren durch eine Kompositionsbeziehung einschließen. Zu diesem Zweck wird die Fixierung eingeführt. Erfolgt während des fixierten Zustands eine Aufforderung zur Migration, wird eine Laufzeit-Exception ausgelöst.

Abbildung 7.4: Beispiel für einen fixierten Prozessor



Auch bei der Fixierung von Prozessoren gilt es Regeln zu beachten: Ein Prozessor A, der per UniPull von Prozessor B gezogen wird, darf nicht festgesetzt werden. Der Versuch, dies doch zu tun, endet in einer Exception, da sonst sich widersprechende Relationen zwischen A und B entstehen würden. Fixierungen haben auf die Gruppierung von Prozessoren Auswirkungen. Eine Gruppe

von Prozessoren, die mittels UniPull und BiPull verknüpft ist, kann nicht migrieren, sobald ein einziger Prozessor aus der Gruppe festgesetzt wird. Es gibt aber auch Konstellationen, in denen einige Prozessoren bei ihrer festgesetzten Gruppen bleiben müssen, andere aus der Gruppe jedoch die Möglichkeit erhalten sollten, sich im Fall von Anforderungen bewegen zu können. Es soll also die Fixierung eines Gruppenmitglieds ignoriert werden, ohne die Gruppe grundsätzlich zu verlassen. Hierzu ist ein Gruppierungsmechanismus nötig, der sich bei Gruppenzerlegung tolerant verhält.

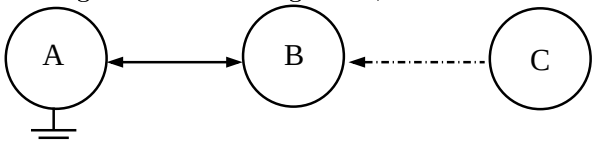
Schwache Komposition

Die beiden bisher besprochenen Kompositionsarten erlauben eine statische und verlässliche Bindung von Prozessoren. Wenn die Bindung ohne Exception ausführbar war, dann befinden sich anschließend alle miteinander verbundenen Prozessoren auf dem selben Rechner. Doch gelegentlich ist eine schwächere Bindung wünschenswert, die eine Ko-Lokation zwar nicht garantiert, sie aber herstellt, wenn dies möglich ist und nicht durch andere Regeln unterbunden wird. Daher ist es nötig, eine schwache Komposition einzuführen, die in speziellen Situationen ein flexibleres Verhalten ermöglicht. Sie wird als RubberPull-Relokator bezeichnet.

Abbildung 7.5: Einfache RubberPull-Verknüpfung

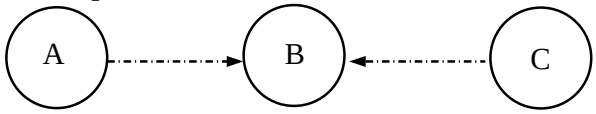
Wird ein Prozessor B mittels RubberPull mit einem Prozessor C verknüpft, wie in Abbildung 7.6 dargestellt, so versuchen Prozessor A und B mit C zu migrieren. Sollte die Migration von A oder B unmöglich sein, zum Beispiel weil, wie in der Abbildung gezeigt, A fixiert ist, migriert C alleine. Es wird auch keine Exception geworfen. Wenn nach der Migration von C die Fixierung aufgehoben wird, folgen A und B nicht automatisch dem migrierten C. Erst bei einer erneuten Bewegung von C zu einer anderen Laufzeitumgebung werden A und B ihrem Relokator C folgen.

Abbildung 7.6: C kann migrieren, obwohl A fixiert ist



Ein weiteres Beispiel ist in Abbildung 7.7 gezeigt: Sowohl A als auch C sind durch einen RubberPull-Relokator mit Prozessor B verknüpft. Wenn Prozessor A bewegt wird, wird auch B mitmigriert - obwohl auch C eine Verknüpfung mit B hat. Bewegt sich danach Prozessor C, wird B mit zum neuen Aufenthaltsort von C migriert, egal wo B sich zu diesem Zeitpunkt befindet.

Abbildung 7.7: Alle Prozessoren dürfen migrieren - wenn A oder C migrieren, wird B mit ans Ziel migriert



Schnittstellenbeschreibung

Dieses Konzept wurde in die Sprache Dejay integriert. Die Verknüpfung von Prozessoren miteinander ist durch eine Schnittstelle zur Laufzeit möglich. Der Aufruf der Methoden geschieht - da Referenzen auf virtuelle Prozessoren immer entfernte Referenzen sind - über die Proxyklasse DjProcessor. Die wichtigste hierfür nötige Methode ist die Methode addRelocator().

```
public void addRelocator(RelocatorType r,DjProcessor x)
```

Die Methode addRelocator() aus DjProcessor verknüpft den Prozessor, an dem sie aufgerufen wird, mit dem Prozessor x. Dabei wird der RelocatorTyp r benutzt. Beispielsweise der folgende Codeabschnitt

```
DjProcessor djA= new DjProcessor("tcp://vsys1:8000");
DjProcessor djB= new DjProcessor("tcp://vsys5:9600");
djA.addRelocator(RelocatorType.RUBBERPULL,djB);
```

dazu führen, dass Prozessor djA von Prozessor djB per RubberPull mitgezogen wird, wenn djB migriert - soweit djA nicht aus anderen Gründen verhindert ist (z.B. durch den Status "fixiert").

Um eine Kompositionsbeziehung wieder aufzuheben, wird die folgende Methode verwendet:

```
public void removeRelocator(RelocatorType r,DjProcessor x)
```

Diese Methode funktioniert analog zur addRelocator().

Mittels der Methode moveTo() wird die Bewegung des DjProzessors ausgelöst, wie bereits vor der Einführung des Gruppierungsmechanismus. Doch nun prüft sie zusätzlich, ob eine Migration aus gruppierungstechnischen Gründen (Fixierungen, UniPulls) möglich ist, und wirft im Falle eines Verbots eine MovementException. Dabei wird im Fehlerfall eine Auflistung aller am Fehler beteiligten Prozessoren gemeldet. Im Erfolgsfall sorgt die Methode dafür, dass alle verketteten Prozessoren rekursiv durchlaufen werden, und bewegt sie mit zum Ziel des Prozessors.

```
public void moveTo(String a) throws MovementException
```

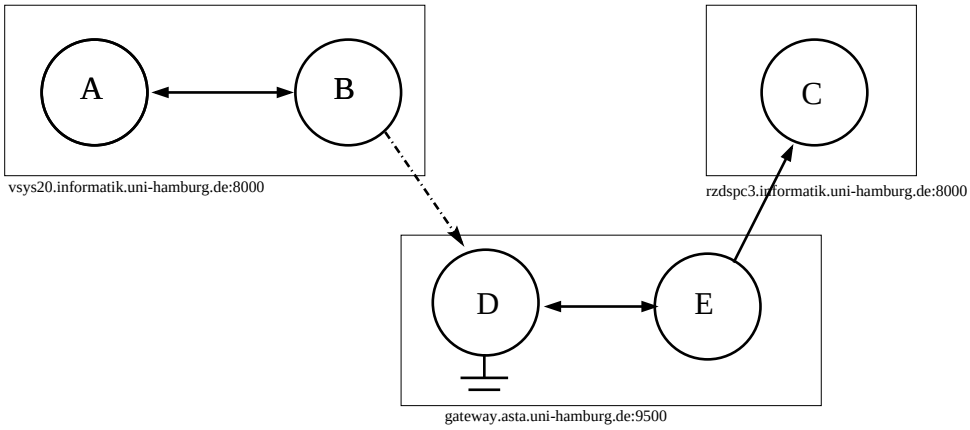
Die Methoden setFixed()/getFixed() dienen dem Festsetzen, Lösen und der Abfrage des Status eines Prozessors. Nach dem Aufruf von setFixed(true) kann der

Prozessor (und damit auch alle Prozessoren, die ihn direkt oder indirekt referenzieren) nicht mehr verschoben werden. Durch ein einfaches Nacheinanderschalten von moveTo() und setFixed(true) lässt sich in einem verteilten System nicht gewährleisten, dass sich nicht ein anderer Prozessor mit einem weiteren moveTo() zwischen die beiden Befehle schiebt, und der setFixed(true) an einer vollkommen unerwarteten Adresse ausgeführt wird. Daher bedarf es der atomaren Methode fixAt(...), die eine Bewegung sowie eine Fixierung am Zielort garantiert.

```
public void setFixed(boolean f)
public boolean getFixed()
public void fixAt(String a) throws MovementException
```

Um die Benutzung der API noch weiter zu verdeutlichen, wird im Folgenden ein etwas komplexeres Beispiel präsentiert, das in Abbildung 7.8 grafisch dargestellt ist und durch die Ausführung des folgenden Codes erreicht wird.

Abbildung 7.8: Beispiel für Anwendung aller Gruppierungsmechanismen




```
DjProcessor a,b,c,d,e;
a= new DjProcessor("tcp://vsyspc2:8000");
b= new DjProcessor("tcp://vsyspc20:8000");
c= new DjProcessor("tcp://gateway:9500");
d= new DjProcessor("tcp://gateway:9500");
e= new DjProcessor("tcp://rzdspc3:8000");
a.addRelocator(RelocatorType.BIPULL,b);
d.addRelocator(RelocatorType.RUBBERPULL,b);
d.addRelocator(RelocatorType.BIPULL,e);
c.addRelocator(RelocatorType.UNIPULL,e);
d.setFixed(true);
// doSometing()...
c.removeRelocator(RelocatorType.UNIPULL,e);
d.setFixed(false);
d.removeRelocator(RelocatorType.BIPULL,e);
```

```
a.removeRelocator(RelocatorType.BIPULL,b);
d.removeRelocator(RelocatorType.RUBBERPULL,b);
```

7.8 Ausnahmebehandlung

Bei der Behandlung von Ausnahmen wird in Dejay zwischen zwei Arten von Ausnahmen unterschieden. Zum einen gibt es die, die in der Schnittstelle eines Objektes definiert sind und regulär von einem Objekt erzeugt oder »geworfen« werden können. Diese Ausnahmen müssen natürlich auch im verteilten Fall abgefangen und behandelt werden. Bei Methodenaufrufen auf entfernte Objekte müssen diese genau wie lokale Aufrufe durch einen entsprechenden try-catch-Block abgefangen werden. Auch einige Methoden, die erst durch die entfernte Referenzierung zur Verfügung stehen, wie etwa die Methode moveTo(), können Ausnahmen erzeugen, entsprechend zur Definition ihrer Schnittstelle im entfernten Typ.

Zum zweiten gibt es Ausnahmen, die nicht in der Schnittstelle des Objektes definiert sind, sondern erst durch den entfernten Zugriff entstehen können, zum Beispiel durch einen Netzwerkfehler. Solche Fehler können, müssen aber nicht behandelt werden. In Anwendungen, die eine hohe Robustheit aufweisen müssen, kann damit auf solche Ausnahmen explizit reagiert werden. Doch in Fällen, in denen man sich hinreichend auf das Netzwerk verlassen kann oder den erhöhten Aufwand für die Ausnahmebehandlung ohnehin nicht als gerechtfertigt ansieht, ist man nicht dazu gezwungen. Dadurch ist das Erlernen und das Erstellen von kleineren Beispielen in Dejay diesbezüglich sehr viel leichter als beispielsweise beim RMI, bei dem jeder entfernte Aufruf durch einen try-catch-Block umgeben sein muss.

Der Bereich der Ausnahmebehandlung ist noch in der Entwicklung, so dass die entgültige Menge der Ausnahmen noch nicht gegeben werden kann. Bisher sind die folgenden Ausnahme – sozusagen exemplarisch – zusätzlich in Dejay definiert:

- StartProgramFailedException: Die Ausführung der main()-Methode oder die Erzeugung des ersten impliziten virtuellen Prozessors ist fehlgeschlagen.
- ConstructProcessorFailedException: Die Erzeugung eines virtuellen Prozessors ist fehlgeschlagen.
- NoRemoteReferenceClassException: Bei einem asynchronen Aufruf konnte das Ergebnis nicht als entfernte Referenz übergeben werden, zum Beispiel weil der Dj-Typ nicht verfügbar war.

7.9 Der Compiler dejayc

Dejay hat einen eigenen Compiler namens dejayc, der ähnlich zu verwenden ist wie der javac des JDK. Er übersetzt Dejay-Klassen, die in einem File, das den Namen der enthaltenen Klasse und die Endung .dj trägt, in Java-Bytecode. Er erzeugt dabei zwischenzeitlich zwei Java-Klassen, die dann automatisch in

Bytecode weiterübersetzt werden. Durch Direktiven lässt sich der Compiler aber auch so steuern, das nur Java-Code erzeugt wird oder dieser Code erhalten bleibt. Doch im Normalfall erzeugt er aus einer Datei Hello.dj die zwei Klassen Hello.class und DjHello.class.

```
lin> dejayc Hello.dj
lin| Hello.class
lin| DjHello.class
```

Die Klasse Hello.class entspricht der Klasse, von der eine lokale Instanz erzeugt wird. Die Klasse DjHello.class entspricht einem Proxy, das den entfernten Zugriff auf ein Objekt ermöglicht. Wenn eine Klasse die besonderen Eigenschaften von Dejay wie Migration und asynchrone Aufrufe verwenden soll, muss sie auf diese Weise übersetzt werden, um entsprechend zu funktionieren. In Dejay können aber auch Java-Klassen integriert werden, wie zum Beispiel die aus dem JDK. Sie können sogar entfernte Referenzen, etwa als Parameter bei einem Methodenaufruf, erhalten und so einen Teil der Dejay-Funktionalität nutzen. Methodenaufrufe werden dann entfernt ausgeführt, ohne dass die Java-Klasse bewusst auf Verteilung vorbereitet wurde. Allerdings können sie keine Migration oder asynchrone Methodenaufrufe verwenden und Zugriffe auf Datenfelder sind nicht möglich.

Es ist auch möglich, entfernte Referenzen auf bestehende Java-Klassen zu erzeugen. Der Dejay-Compiler ist in der Lage, die Proxy-Klasse nur aus der .class-Datei einer Java-Klasse zu erstellen, sie muss also nicht einmal in Sourcecode vorliegen. Zum Beispiel kann man ein Proxy der Klasse java.util.Vector erzeugen und damit Vektoren entfernt erzeugen, zugreifen und migrieren.

```
lin> dejayc java.util.Vector.class
lin| DjVector.class
```

Hierfür gibt es allerdings eine Einschränkung: Bei nicht weiter ableitbaren Klassen (final) ist dies nicht möglich, wie zum Beispiel bei der Klasse java.lang.String.

7.10 Programmstart

Die erzeugten Klassen können anschließend von jedem gewöhnlichen Java-Interpreter ausgeführt werden. Die Klasse, in der die zur Ausführung kommende main()-Methode enthalten ist, muss allerdings ebenfalls mit dejayc übersetzt worden sein. Mit der Direktive -l kann angegeben werden, dass diese Klasse nur lokal verwendet und nicht entfernt referenziert wird.

```
lin> javac -l Startup.dj
lin> java Startup
```

Die main()-Methode wird vom Dejay-Compiler so verändert, dass automatisch ein Voyager-Daemon gestartet und darin ein virtueller Prozessor angelegt wird, wodurch die Erzeugung des ersten virtuellen Prozessors für den Programmierer transparent ist. Über den Parameter -p <Port> kann der Port dieses Voyager-Daemons bestimmt werden.

```
lin> java Startup -p 8000
```

Wird der Parameter -p nicht angegeben, wird der Voyager-Daemon auf Port 7042 gestartet. Natürlich können noch weitere Parameter angegeben werden, die von der main()-Methode ausgelesen und verarbeitet werden, doch dieser Parameter muss, wenn er angegeben wird, der erste sein.

Um ein Programm verteilt ablaufen zu lassen, müssen auf den beteiligten Rechnern ebenfalls Voyager-Daemons auf entsprechenden Ports gestartet werden. Dies kann sowohl aus einem Dejay-Programm heraus als auch von der Kommandozeile aus getan werden. Von der Kommandozeile aus kann der Voyager-Daemon mit dem folgenden Befehl gestartet werden.

```
sun> voyager 8000
```

Voyager benutzt die Umgebungsvariable CLASSPATH auf dem jeweiligen lokalen Rechner, um Klassen-Code zu finden. Da es bei verteilten Anwendungen zweckmäßig sein kann, den Sourcecode der Klassen nicht auf alle beteiligten Rechner zu verteilen, gibt es die Möglichkeit, weitere Quellen für die Suche nach Klassen anzugeben. Diese Quellen können sowohl Verzeichnisse auf dem lokalen Rechner als auch entfernte Rechner sein. Entfernte Rechner müssen dabei allerdings über http ansprechbar sein und zum Beispiel einen Webserver oder einen anderen Voyager-Daemon auf dem entsprechenden Port betreiben. Im folgenden Beispiel werden die Suchmöglichkeiten für den Voyager-Daemon um das lokale Verzeichnis /home/my-code und den entfernten Rechner lin erweitert.

```
sun> voyager 8000 -c file:///home/my-code/ -c http://lin:9000
```

7.11 Einschränkungen

Die bisherigen Abschnitte haben sich mit den Erweiterungen von Java befasst. In den meisten Fällen ist es gelungen, die Benutzung möglichst intuitiv zu gestalten und die Semantik daran zu orientieren, was man als Java-Programmierer erwarten würde. Aber es gibt auch Fälle, in denen, aus technischen Gründen oder aus Zeitmangel, dies nicht erreicht werden konnte, außerdem gibt es einige Einschränkungen in Dejay gegenüber der normalen Verwendung von Java. Die folgende Auflistung kann leider keinen Anspruch auf Vollständigkeit erheben, versucht aber möglichst alle bekannten Besonderheiten aufzuzählen.

- Viele Klassen in den Standardbibliotheken von Java sind nicht weiter ableitbar (final), so zum Beispiel die Klasse String. Da der Mechanismus, mit dem in Dejay Proxy-Klassen erzeugt werden, auf der Ableitung von der lokalen Klasse beruht, ist dieser für nicht ableitbare Klassen nicht anwendbar. Dementsprechend können von diesen Klassen keine entfernten Referenzen erzeugt werden. Um diese Einschränkung zu umgehen, ist die Benutzung von Wrapper-Klassen nötig. Dabei wird per Aggregation eine Instanz der eigentlichen Klasse (zum Beispiel String) in einer sie umgebenden Klasse verwaltet (zum Beispiel eine Klasse Adresse). Die umgebende

Klasse bildet dabei die Schnittstelle der eigentlichen Klasse nach oder stellt eine ähnliche Schnittstelle zur Verfügung. Von außen ist nur die umgebende Klasse mit ihrer öffentlichen Schnittstelle nutzbar. Ein Methodenaufruf kann von der umgebenden Klasse an den entsprechenden Methodenaufruf der eigentlichen Klasse weitergeleitet werden. Im verteilten Programm werden dann Instanzen der Wrapper-Klasse benutzt, für die nun entfernte Referenzen (zum Beispiel DjAdresse) erzeugt werden können.

- Ebenso sind Methoden, die als final deklariert sind, nicht in der Schnittstelle einer Proxy-Klasse enthalten.

- Variablen eines Objektes lassen sich nicht entfernt zugreifen, da die Proxy-Klasse nur Methoden zum entfernten Zugriff bietet. Direkte Zugriffe auf Variablen sollten aber ohnehin vermieden werden.

- Der Geltungsbereich von statischen Variablen ist die virtuelle Maschine. Sie werden nicht mit migriert. Nach einer Migration werden beim Zugriff auf solche statischen Variablen also evtl. andere Werte ausgegeben.

- Die Methoden der Klasse Object werden allesamt von einer Proxy-Klasse überschrieben und an das entfernte Objekt weitergeleitet. Nicht zuletzt dadurch wirken sie wie entfernte Referenzen, doch die gewohnte Semantik dieser Klassen ist verändert.

- Java ermöglicht Nebenläufigkeit durch Abspaltung von Threads aus dem eigentlichen Kontrollfluss. In Dejay wird Nebenläufigkeit üblicherweise durch virtuelle Prozessoren ersetzt. Aus Gründen der Kompatibilität zu Java und aus praktischen Überlegungen (z.B. macht die Programmierung von Benutzeroberflächen in Java intensiven Gebrauch von Threads), ist die Verwendung der Klasse Thread auch in Dejay weiterhin möglich. Doch die Verwendung von Threads in einem virtuellen Prozessor führt dazu, dass in diesem virtuellen Prozessor mehrere Kontrollflüsse existieren. Eine sequentielle Bearbeitung der eingehenden Aufrufe ist damit nicht mehr gewährleistet. Probleme treten bei der Migration auf: Bei der Migration eines virtuellen Prozessors kommt es zu unterschiedlichem Verhalten, je nachdem ob der virtuelle Prozessor auf einen anderen Rechner bzw. physikalischen Prozessor bewegt wird oder nicht. Bleibt der virtuelle Prozessor auf dem gleichen Rechner, bleibt die Verbindung zu dem abgespaltenen Kontrollfluss weiterhin erhalten. Wechselt der virtuelle Prozessor allerdings auf einen anderen Rechner, wechselt zwar das Thread-Objekt ebenfalls den Rechner, der eigentliche Thread bleibt aber auf dem alten Rechner. Die Verbindung zu ihm geht verloren. Die Migration von virtuellen Prozessoren, die Thread-Objekte enthalten, sollte daher vermieden werden.

- Mit der Einführung des JDK 1.1 wurde für Java auch die Verwendung von inneren und lokalen Klassen definiert. Diese speziellen Klassen werden in Dejay derzeit nicht angemessen behandelt.

- Die Synchronisation ist in Dejay bisher nur eingeschränkt behandelt. Die vorhandenen Synchronisationsmechanismen basieren auf denen von Java, so dass zum Beispiel das Schlüsselwort synchronized auf den Proxy wirkt und nicht auf das entfernte Objekt. Es ist geplant, auch die Synchronisation auf das entfernte Objekt wirken zu lassen.

- Für den Fall, dass ein Objekt nur lokal referenziert wird, aber eine entfernte Referenz benötigt wird, zum Beispiel als Parameter bei einem entfernten Methodenaufruf, gibt es derzeit keine Möglichkeit eine entfernte Referenz zu erhalten, da Objekte ihren virtuellen Prozessor nicht kennen. Dies ist nur entfernt über den Copy-Konstruktor für schon entfernt referenzierte Objekte möglich.

- Proxy-Klassen erben von der Klasse, auf die sie entfernt referenzieren sollen. Dies hat viele Vorteile, birgt aber auch ein paar Nachteile. So wird zum Beispiel beim Aufruf von new DjA der Konstruktor einer Klasse A zweimal ausgeführt, einmal bei der Erzeugung des Objektes DjA, der den Konstruktor seiner Superklasse (lokal) ausführt, und ein zweites mal bei der entfernten Erzeugung (entfernt). Dadurch werden etwa Ausgaben des Konstruktors sowohl lokal als auch entfernt ausgegeben. Ausgaben oder die Erzeugung von GUIs sollten daher nicht im Konstruktor stattfinden.

Kapitel 8

Modellierung und Visualisierung von Verteilung

Das in den vorhergehenden Kapiteln entwickelte Konzept der virtuellen Prozessoren konnte im Kapitel 7 auf eine konkrete textuelle Syntax, auf die Sprache Dejay, abgebildet werden. In diesem Kapitel sollen die Möglichkeiten untersucht werden, wie die bisher entwickelten Konzepte auf eine konkrete grafische Syntax abgebildet werden können und welche Vorteile sich daraus für die Modellierung und die visuelle Darstellung ergeben.

Die grafische Modellierung bietet als Komplement zur textuellen Darstellung in Programmiersprachen einige Vorteile. Komplexe Zusammenhänge werden häufig durch eine grafische Darstellung für den menschlichen Betrachter leichter zugänglich. Sie bietet eine abstraktere Sicht auf den gleichen zu modellierenden Gegenstand und erlauben dadurch eine bessere Fokussierung auf spezielle Aspekte. Grafische Modelle sind auch leichter auf ein semantisches Modell abbildbar, während die Bewahrung der Synchronität zwischen einer textuellen Darstellung und den semantischen Modellen ein schwierigeres Problem darstellt. So befinden sich zum Beispiel Texteditoren, die eine semantische Unterstützung für die Entwicklung von Programmcode bieten, erst in der Entwicklung, wie von [Vanter 2000] dargestellt, während die üblichen Editoren eine rein syntaktische Unterstützung wie Syntaxhighlighting oder Textvervollständigung bieten. Werkzeuge zur Unterstützung von grafischen Modellierungssprachen sind dagegen eher in der Lage die semantische Bedeutung unmittelbar zu erfassen und entsprechende Unterstützung zu bieten [Robbins 1999].

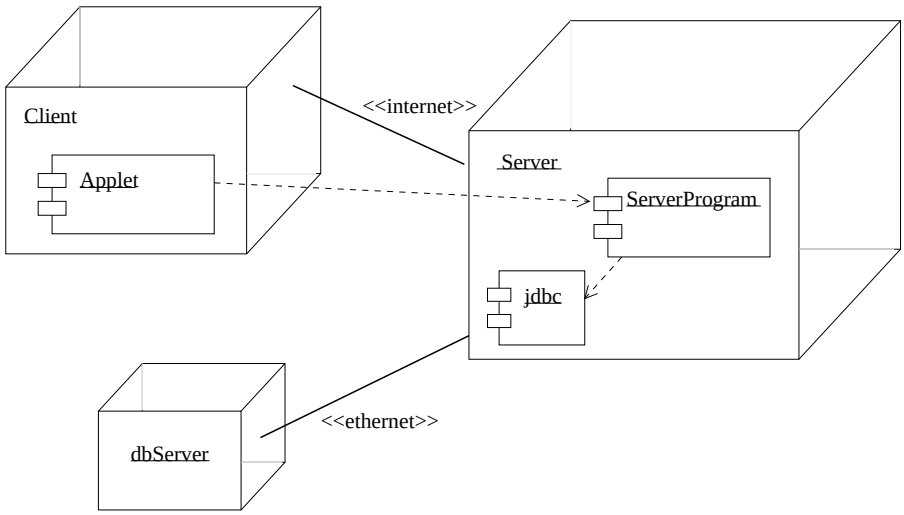
Für die Analyse und das Design komplexer, objektorientierter Anwendungen hat sich der Einsatz der Modellierungssprache UML in starkem Umfang durchgesetzt. UML bietet eine Reihe unterschiedlicher Diagramme an, die verschiedene Aspekte einer Anwendung veranschaulichen und somit sowohl als Dokumentation, als auch als Kommunikationgrundlage für die beteiligten Entwickler dienen. Während das Klassendiagramm das am meisten verwendete Diagramm ist, bietet die UML mit dem Komponenten- und Verteilungsdiagramm auch Diagrammtypen, die gerade für die Entwicklung verteilter Anwendungen nützlich sein können, da mit ihnen die Verteilungsaspekte einer Anwendung darstellbar sind.

8.1 Modellierung von Verteilung mit UML

Das UML-Verteilungsdiagramm (Deployment Diagram) dient dazu die Konfiguration eines verteilten Systems zu beschreiben. Es werden Beziehungen zwischen Verarbeitungseinheiten, wie Rechnern, und Software-Elementen, wie Komponenten, Prozessen und Objekten dargestellt. Zusätzlich zeigt das Verteilungdiagramm das Netz zwischen den Verarbeitungeinheiten, das von den Softwarekomponenten zur Kommunikation genutzt wird. Das Verteilungsdiagramm ist somit ein Graph in dem die Knoten Verarbeitungseinheiten repräsentieren, während die Kanten Kommunikations-Assoziationen repräsentieren.

Knoten werden im Verteilungsdiagramm durch Quader dargestellt. Komponenten, die in dem Knoten existieren und von diesem ausgeführt werden, werden in den Quader gezeichnet. In der UML stellen Komponenten austauschbare Software-Elemente eines verteilten Systems dar. Im Verteilungsdiagramm werden sie in Form eines Rechtecks mit zwei kleineren Rechtecken auf der linken Kante dargestellt. Die Komponenten können neben der Angabe eines Namens näher spezifiziert werden, indem Klassen in die Komponenten gezeichnet werden, die von der Komponente implementiert werden. Außerdem können Komponenten weitere Komponenten enthalten. Komponenten können durch gestrichelte Pfeile miteinander verbunden werden um anzuzeigen, dass eine Komponente den Dienst einer anderen Komponente nutzt. Die genaue Beziehung der Komponenten zueinander kann durch einen Stereotyp spezifiziert werden.

Abbildung 8.1: Ein Verteilungsdiagram nach UML



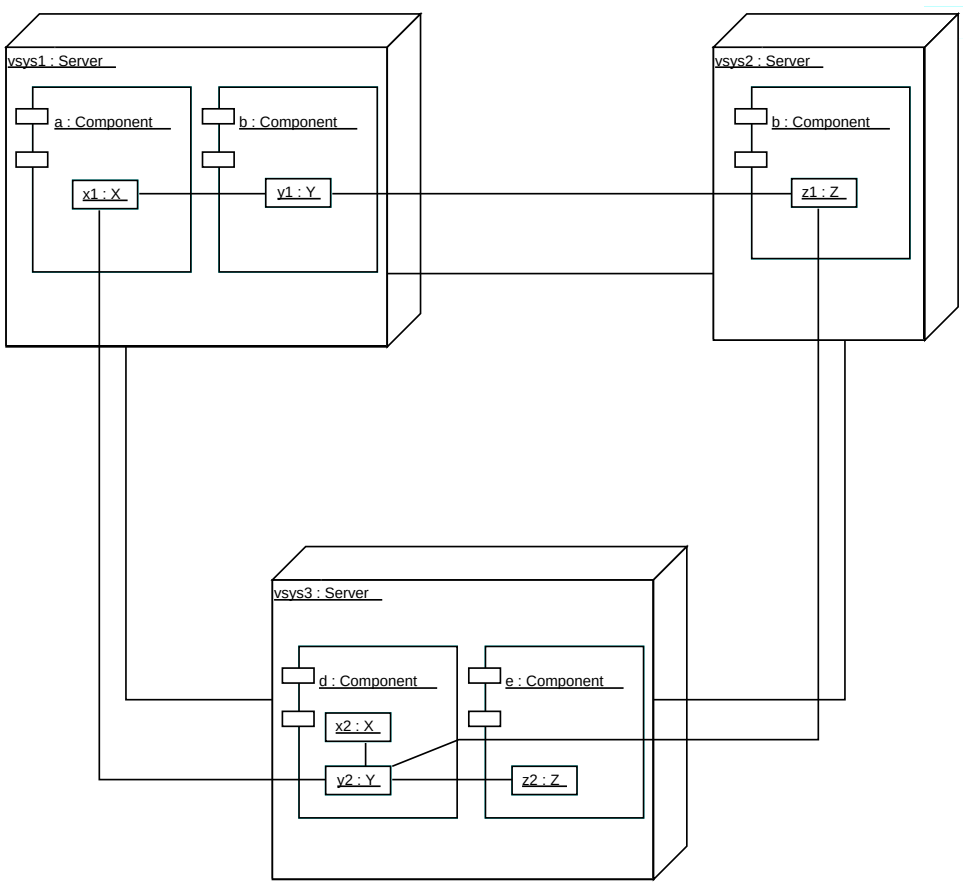
Auch die Migration, also das Verschieben von Komponenten und Objekten zwischen Knoten unter Beibehaltung der Referenzen, lässt sich im Verteilungsdiagramm ausdrücken. Ein gestrichelter Pfeil zwischen zwei Komponenten mit dem »become« Stereotyp drückt die Migration einer Komponente aus. Gleichermaßen lässt sich auch die Migration eines Objektes zwischen zwei Komponenten darstellen.

8.1.1 Abbildung von UML auf Dejay

Die in UML in Verteilungsdiagramm und Komponentendiagramm modellierbare Information läßt sich einfach auf Konstrukte von Dejay abbilden. Das in der UML verwendete Konzept der Komponente findet eine direkte Abbildung auf den virtuellen Prozessor in Dejay. Objekte, die laut einer UML-Spezifikation in einer Komponente enthalten sein sollen, werden in einem entsprechenden virtuellen Prozessor erzeugt. UML erlaubt die Spezifikation von sowohl synchroner als auch asynchroner Kommunikation, die beide in Dejay unmittelbar unterstützt werden.

Um die unmittelbare Abbildbarkeit zu demonstrieren, soll hier ein Beispiel gezeigt werden. In der Abbildung 8.2 ist ein allgemein gehaltenes Beispiel dargestellt, in dem 3 Rechnerknoten, 5 Komponenten und 6 Objekte verwendet werden. Die Rechnerknoten sind mit vsys1 bis vsys3 bezeichnet und können realexistierende Rechner repräsentieren. Für die Verwendung von Dejay muss auf diesen Rechnern lediglich der virtuelle Backplane als Daemon gestartet sein. Auf diesen Rechnern können dann die Komponenten a bis e erzeugt werden. Diese wiederum können dann Objekte enthalten.

Abbildung 8.2: Beispiel eines Verteilungsdiagrams nach UML



Das in der Abbildung dargestellte Szenario kann mit Dejay mit dem folgenden Codefragment erzeugt werden. Dabei werden zunächst die Komponenten als virtuelle Prozessoren erzeugt und anschließend die entsprechenden Objekte in diese Komponenten platziert. Die Zuweisung von Referenzen geschieht wie in Java üblich durch die Übergabe als Parameter in Konstruktoren oder Methoden. Dieser Code kann von jedem Rechner aus ausgeführt werden, der mit den drei genannten Rechnern über ein TCP/IP-basiertes Netz verbunden ist. Der nötige Code kann lokal installiert oder über einen Webserver automatisch verteilt werden.

```
DjProcessor a= new DjProcessor("tcp://vsys1:8000");
DjProcessor b= new DjProcessor("tcp://vsys1:8000");
DjProcessor c= new DjProcessor("tcp://vsys2:8000");
DjProcessor d= new DjProcessor("tcp://vsys3:8000");
DjProcessor e= new DjProcessor("tcp://vsys3:8000");
DjX x1= new DjX(a);
DjZ z1= new DjZ(c);
DjY y1= new DjY(x1, z1, b);
DjX x2= new DjX(d);
DjZ z2= new DjZ(e);
DjY y2= new DjY(x2, z2, d);
y2.add(z1);
```

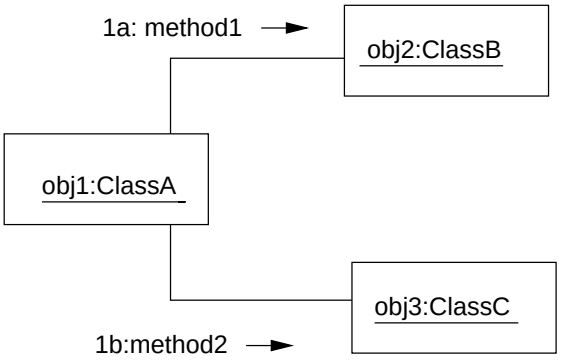
Eine Abbildung dieser Verteilungskonzepte von UML in eine Programmiersprache ohne ein geeignetes Konzept, insbesondere zur Abbildung der Komponenten und der asynchronen Kommunikation ist bedeutend schwieriger. Insgesamt ist die Abbildung von Komponenten auf Code nicht klar. Für die Abbildung von Komponenten, die nur eine Schnittstelle (nach dem obigen Beispiel ein Objekt) anbieten, kommt die Abbildung auf eine Java-Bean in Frage. Diese erlauben allerdings nicht die Behandlung von Nebenläufigkeit oder Verteilung, sie bieten lediglich eine standardisierte Funktionalität an. Komponenten mit mehreren Schnittstellen können auf JARs abgebildet werden, die aber nur eine Sammlung von zusammengehörigen Klassen darstellen, die gemeinsam ausgeliefert werden können. Zur Beschreibung der Laufzeitsituation sind sie aber ebenfalls ungeeignet. Das COM-Modell von Microsoft und die Enterprise JavaBeans von Sun können zur Abbildung verwendet werden, sind aber ungleich komplizierter. Siehe hierzu auch [Griffel 1998].

Aspekte der Nebenläufigkeit werden in UML unter anderem durch die Diagrammarten Aktivitäts-, Kollaborations- und (das zum letzteren nahezu äquivalente) Sequenzdiagramm ausgedrückt. In Kollaborations- und Sequenzdiagrammen werden die Aufrufe durch eine Sequenznummer zeitlich sortiert. Wenn diese Nummer mehrfach verwendet wird und durch einen angefügten Buchstaben unterschieden wird, so sollen die dazugehörigen Aufrufe nebenläufig ausgeführt werden. Abbildung 8.3 zeigt ein solches Beispiel in einem Kollaborationsdiagramm.

Dieses Beispiel kann in Dejay durch den folgenden Code realisiert werden

```
public class ClassA {
```

Abbildung 8.3: Beispiel eines Kollaborationsdiagramms nach UML




```
  ...
  public void method() {
    Object a= obj2.method1(Dejay.ASYNC);//nebenläufig
    Object b= obj3.method2();
    System.out.println("Result: "+a+ " "+b);
  }
}
```

Eine Umsetzung in Java erfordert den Einsatz von Threads und von Synchronisation, wie es in 3.2 erläutert und in Abbildung 3.3 dargestellt ist und könnte in etwa folgendem Code resultieren.

```
public class ClassA {
  Object a;

  public void method{
    class Caller extends Thread {

      Client client;
      private boolean returned = false;

      public Caller (Client cl) {
        client = cl;
        this.start();
      }

      public void run() {
        Object a = client.method1();
        returned = true;
        if (client != null) {
          client.callback(a);
        }
      }
```

```
        public boolean returned() {
          boolean result = returned;
          returned = false;
          return result;
        }
      }
      Caller caller = new Caller(obj2); //nebenläufig
      Object b= obj3.method2();
      while(caller.returned()==false) {
         try {
           Thread.sleep(1000);
         } catch (InterruptedException e) {
           e.printStackTrace();
         }
      }
      System.out.println("Result: "+a+ " "+b);
   }

   public void callback(Object res) {
      a=res;
   }
}
```

Durch eine solche einfache Abbildbarkeit von Verteilung und Nebenläufigkeit von den im Design mit UML entworfenen Systemen kann die Komplexität der Implementierung deutlich vermindert werden.

Doch auch die Konsistenz der Designdokumente mit der tatsächlichen Implementierung kann dadurch gewährleistet werden. Dieser Aspekt soll im folgenden Abschnitt ausgeführt werden.

8.1.2 Abbildung von Dejay nach UML

Die Abbildung in umgekehrter Richtung, von einem lauffähigen oder laufenden Programm auf eine grafische Darstellung, ist erheblich schwieriger und soll daher hier ausführlicher diskutiert werden. Diese Abbildung, auch als reverse engineering oder als Visualisierung bezeichnet, lässt sich für objektorientierter Programme in eine statische und eine dynamische Ebene trennen. Diagramme der statischen Ebene beschreiben Strukturen und Interaktionen, die unabhängig vom Beobachtungszeitpunkt Gültigkeit haben. In der UML sind dies das Klassendiagramm, Zustands- und Aktivitätsdiagramm. Konkrete Zustände und Programmabläufe sind der dynamischen Ebene zugeordnet und werden in der UML durch Objektdiagramme sowie Sequenz- und Kollaborationsdiagramme und durch Verteilungsdiagramme dargestellt.

Statische Aspekte lassen sich in der Regel leicht durch Inspektion des Programmcodes gewinnen, woraufhin direkt entsprechende Diagramme generiert

werden können. Um jedoch das dynamische Verhalten eines Programms zu visualisieren, reicht es nicht aus den Programmcode zu untersuchen. Es gibt im wesentlichen zwei Möglichkeiten Interaktions-Informationen eines Programms zu gewinnen: Instrumentierung und Verhaltensreflektion.

Bei der Instrumentierung wird der Programmtext so verändert, dass bei Eintreffen interessanter Ereignisse zusätzliche Befehle ausgeführt werden. Diese Befehle können einfache Textausgaben oder Methodenaufrufen zu einem externen Programm sein, dass die Erzeugung und Darstellung eines Diagramms übernimmt. Für die Visualisierung sind im allgemeinen Konstruktion und Dekonstruktion von Objekten, Methodenaufrufe und -rücksprünge sowie Zuweisung von Objektreferenzen interessant. Im Hinblick auf verteilte Systeme sind insbesondere entfernte Methodenaufrufe von Bedeutung.

Vorteil der Instrumentierung ist, dass der für die Visualisierung erforderliche Code im Programm integriert ist und genauso effizient ausgeführt wird. Der Laufzeit-Overhead ist somit relativ gering. Nachteile sind die gesteigerte Programmgröße und insbesondere die erforderliche Neukompilierung.

Bei der Verhaltensreflektion wird das zu untersuchende Programm mit Debug-Informationen kompiliert und unter der Kontrolle des Visualisierungssystems ausgeführt. Innerhalb des Visualisierungssystems können Ereignisse angegeben werden, die zu einer Unterbrechung des untersuchten Programms führen. Auf eine Unterbrechung wird reagiert, indem das Ereignis identifiziert wird und vorhandene Diagramme aktualisiert werden. Danach wird das untersuchte Programm fortgesetzt.

Der Vorteil dieser Methode ist, dass der Quelltext nicht geändert werden muss. Es können somit auch Programme untersucht werden, deren Code nicht vorliegt. Der Nachteil gegenüber der Instrumentierung liegt im hohen Laufzeit-Overhead, der durch die Unterbrechungen und das Interpretieren der Ereignisse entsteht.

Während zur Visualisierung verteilter Anwendungen sehr wenig Arbeiten existieren, gibt es für die Visualisierung objektorientierter Programme im Allgemeinen sogar Produkte. *JinSight* [IBM 2000] zum Beispiel ist ein von IBM entwickeltes Werkzeug zur Visualisierung von Java Programmen. Die zu analysierenden Programme müssen hier nicht instrumentiert werden, stattdessen wird eine instrumentierte virtuelle Maschine benutzt in der Programme ausgeführt werden. Die virtuelle Maschine erstellt eine Trace-Datei mit allen beobachteten Ereignissen, die danach von dem Visualisierungssystem geladen werden kann, um den Programmablauf darzustellen. Durch die Verwendung einer Trace-Datei ist es leider nicht möglich, Diagramme zur Laufzeit zu erstellen.

Das dynamische Verhalten wird hier durch ein Sequenzdiagramm dargestellt. Die Lebenslinien des Sequenzdiagramms sind gefärbt. Instanzen der gleichen Klasse besitzen die gleiche Farbe. Das Tool bietet gute Filtermöglichkeiten für die Darstellung dieses Diagramms. Es lassen sich allerdings nur Klassen filtern, so dass entweder alle Instanzen einer Klasse oder keine angezeigt werden. JinSight kann außerdem sich wiederholende Aufrufe, wie sie in Schleifen vorkommen, erkennen und zusammengefasst darstellen.

Statistische Eigenschaften werden in einem Histogramm dargestellt, in dem die Instanzen nach ihren Klassen gruppiert sind. Die Instanzen werden durch

farbige Rechtecke repäsentiert. Die Farbe zeigt dabei die Höhe der Aktivität innerhalb der jeweiligen Instanz an. Als zusätzliche Funktion lassen sich auch Beziehungen zwischen Instanzen, wie Referenzierungen und Aufrufe, darstellen.

Für die Darstellung des dynamischen Verhaltens verteilter Anwendungen gibt es solche Produkte bisher nicht. Durch die Kapselung von verteilten Objekten in virtuellen Prozessoren liegt aber in Dejay ein Konzept vor, dass eine Verhaltensreflexion für Aspekte der verteilten Kommunikation erlaubt.

8.1.3 Visualisierung des dynamischen Verhaltens von Komponenten

Ein Problem der Programmvisualisierung ist die große Datenmenge, die bei der Beobachtung eines laufenden Programms anfällt und für den Benutzer auf eine sinnvolle und verständliche Weise dargestellt werden muss. Es sind in der Regel Filter und Abstraktionen nötig, um zu einer übersichtlichen Darstellung zu gelangen.

Im Zusammenhang mit dieser Arbeit wurde in Kooperation mit der TU Harburg ein Konzept entwickelt, das durch die Verwendung von dynamischen Komponenten zu einer Vergröberung gelangt, die genutzt werden kann, um sowohl statische als auch dynamische Aspekte eines Programms in angemessener Weise darzustellen. Einige Ergebnisse wurden in [Wienberg et al. 1999] beschrieben.

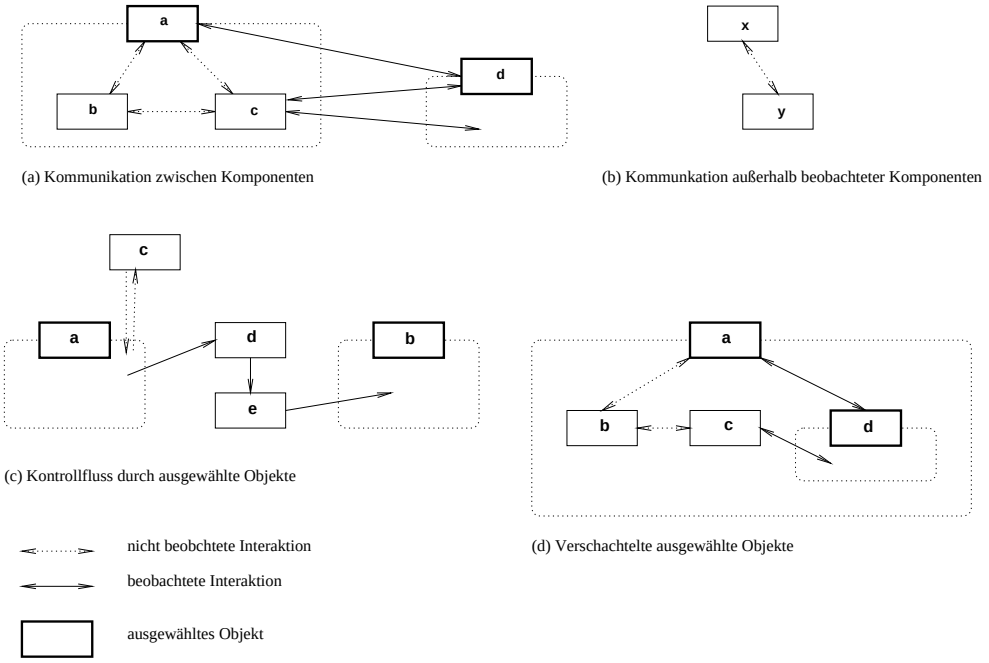
Softwarekomponenten werden in der Regel als Elemente der statischen Ebene angesehen. Sie fassen Programmtexte zu wiederverwendbaren Teilen zusammen. Die statische Struktur eines Softwaresystems entsteht, indem die Softwarekomponenten zusammengesetzt werden. In [Wienberg et al. 1999] werden die konkreten Exemplare einer statischen Softwarekomponente dynamische Komponenten genannt. Diese Unterscheidung ist wichtig, da nur unter Verwendung dynamischer Komponenten auch dynamisches Verhalten modellierbar und visualisierbar ist.

Dynamische Komponenten haben eine hierarchische Struktur, das heißt eine Komponente kann weitere Komponenten enthalten. Diese Eigenschaft kann in der Visualisierung des Systemzustandes ausgenutzt werden, indem zunächst nur die Elemente der obersten Ebene angezeigt werden. Die inneren Komponenten werden erst angezeigt, wenn sich der Benutzer für diese interessiert.

Bei der Visualisierung der Interaktion kann von der Kommunikation innerhalb dynamischer Komponenten abstrahiert werden. Der Benutzer wählt die dynamischen Komponenten, die beobachtet werden sollen. Kommunikation, die vollständig innerhalb einer Komponente stattfindet, wird ebensowenig angezeigt, wie Kommunikation, die vollständig außerhalb der beobachteten Komponenten liegt (Abbildung 8.4 a und b). Es ist allerdings auch möglich, eine Komponente und eine enthaltene Subkomponente auszuwählen, falls man an der Kommunikation innerhalb einer Komponente interessiert ist (Abbildung 8.4 d). Erfolgt ein Aufruf von einer beobachteten Komponente zu einer nicht ausgewählten, wird dieser Aufruf nur gemeldet, wenn der Kontrollfluss über weitere Komponenten eine andere ausgewählten Komponente erreicht. Tritt der Kontrollfluss wieder in die ursprüngliche Komponente ein, wird der Aufruf dagegen ignoriert (Abbildung 8.4 c).

In dem beschriebenen System wird das beobachtete Programm präpariert, indem Variablen, die Komponentenbindungen enthalten ausgezeichnet werden. Auf diese Weise kann zur Laufzeit die Komponentenstruktur des Programms ermittelt werden. Die Gewinnung der Interaktionsinformation erfolgt durch Verhaltensreflektion und die Ergebnisse können als Sequenzdiagramm oder Verteilungsdiagramm dargestellt werden.

Abbildung 8.4: Kommunikationsarten zwischen Komponenten



8.1.4 Integration von Verhaltensreflexion in Dejay

Durch das Konzept des virtuellen Prozessors werden alle Methodenaufrufe zwischen Objekten in unterschiedlichen Prozessoren nie direkt ausgeführt, sondern stets die `execute()` Methode des Ziel-Prozessors aufgerufen. Dadurch ergibt sich eine einfache, effiziente und elegante Möglichkeit eine Verhaltensreflexion in ein Dejay-System zu integrieren. Dies erlaubt die Beobachtung der entfernten Kommunikation von außen, ohne, dass der Code des Dejayprogramms hierfür geändert werden muss. Der virtuelle Prozessor muss lediglich so modifiziert werden, dass innerhalb der `execute()` Methode ein Event erzeugt wird, das von außenstehenden Werkzeugen aufgefangen werden kann. Die `execute()` Methode erhält als Parameter ein Job-Objekt. Die Klasse `Job` hat folgende Struktur:

```
public class Job implements java.io.Serializable {

   String methodName;
   String[] ParameterList;
```

```
Object[] arguments;
int VectorID;
String proxyAddress;
RemoteIndicator returnReference;
```

Für die Verhaltensreflexion ist vor allem der Methodenname von Interesse. Der Prozessor sendet also während der `execute()` Methode ein Signal, das als Parameter den Namen der gerade auszuführenden Methode sowie eine Referenz auf sich selbst enthält.

Desweiteren werden noch vier weitere Signale gesendet, die für externe Beobachter von Interesse sein könnten. Die Methode `newProcessor()` signalisiert die Erzeugung eines neuen Prozessors und wird nach seiner Erzeugung aufgerufen. `processJob()` wird aufgerufen, wenn ein entfernter Methodenaufruf stattfindet, `processResult()` wenn der Methodenaufruf beendet ist. `processorMoved()` wird von einem Prozessor aufgerufen, wenn er zu einem anderen Rechner migriert ist.

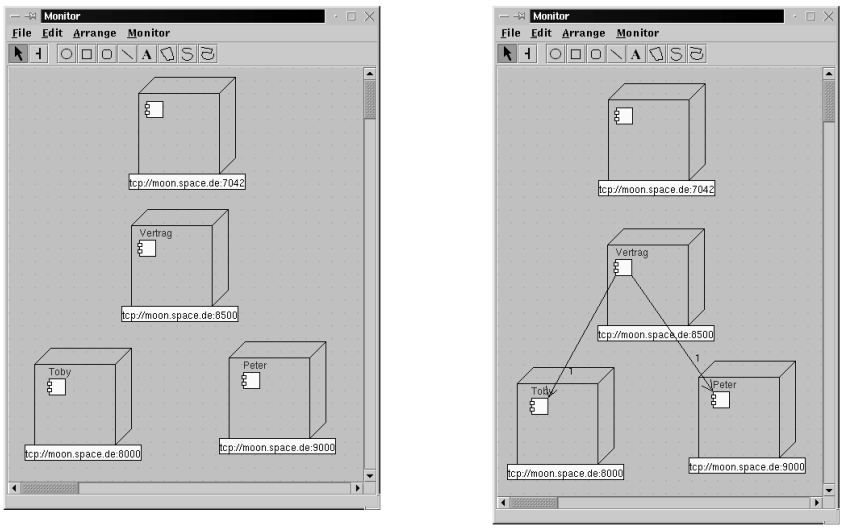
8.2 Entwicklung eines Monitors und Konfigurationsmanagers für Dejay-Programme

Aufbauend auf den Ergebnissen, die im letzten Abschnitt beschrieben wurden, war es möglich ein grafisches Werkzeug zu entwickeln, welches die Kommunikation zwischen Dejay-Prozessoren zur Laufzeit darstellt. Dieses Tool wird im Folgenden als Monitor bezeichnet. Der Dejay-Monitor stellt die entfernten Methodenaufrufe auf zwei verschiedene Weisen dar. Zum einen wird ein UML-Verteilungsdiagramm erzeugt. Die Knoten repräsentieren in diesem Fall einen Voyager-Daemon, der auf einem bestimmten Rechner gestartet wurde. Komponenten sind virtuelle Prozessoren. Einzelne Objekte innerhalb der Prozessoren werden nicht angezeigt, lassen sich aber durch einen Doppelklick auf den Prozessor abfragen. Entfernte Methodenaufrufe werden durch Pfeile zwischen zwei Komponenten gekennzeichnet. Erfolgt ein entfernter Methodenaufruf wird der entsprechende Pfeil farblich kurz hervorgehoben. Zusätzlich wird die Anzahl der Methodenaufrufe seit Beginn des Monitorings mitgezählt. Natürlich erkennt der Monitor auch neu erzeugte und migrierende Prozessoren und stellt diese korrekt innerhalb ihres neuen Knotens dar.

Die zweite Möglichkeit, die der Monitor bietet, um einen Programmablauf zu visualisieren, besteht in einem automatisch generiertem Sequenzdiagramm. Da wiederum nur entfernte Methodenaufrufe berücksichtigt werden, sind die Lebenslinien des Sequenzdiagramms nicht einzelnen Objekten zugeordnet, sondern den virtuellen Prozessoren. Auch in diesem Diagramm ist es möglich, durch einen Doppelklick auf eine Lebenslinie zu erfahren, welche Objekte sich momentan in einem virtuellen Prozessor befinden. Es ist allerdings nur möglich den aktuellen Zustand abzufragen. Man kann also z.B nicht feststellen, welche Objekte sich zu Beginn im Prozessor befanden.

Wie üblich wird durch einen durchgezogenen Pfeil ein Methodenaufruf gekennzeichnet. Ein gestrichelter Pfeil stellt hingegen die Rückgabe an den auf-

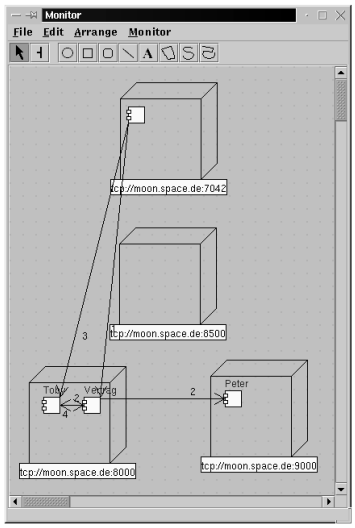
Abbildung 8.5: Der Monitor vor (a) und während (b) einer Kommunikation



rufenden Prozessor dar. Da sich in einem Prozessor sehr viele verschiedene Objekte befinden können, wird ein Pfeil nicht nur mit der aufgerufenen Methode, sondern zusätzlich mit der Klasse des aufgerufenen Objektes in der Form `Klasse.Methode()` beschriftet. Die Sequenzdiagramme werden, ebenso wie die Verteilungsdiagramme, dynamisch zur Laufzeit generiert. Nur im Verteilungsdiagramm lässt sich aber bestimmen, welche Rechner überwacht werden sollen. Wird während eines Programmablaufs ein neuer Rechner hinzugefügt auf dem sich ein virtueller Prozessor befindet, wird auch im Sequenzdiagramm automatisch eine neue Lebenslinie auf der richtigen Höhe hinzugefügt.

Das Verteilungsdiagramm, als Mischung aus Momentaufnahme (die Verteilung der Prozessoren auf unterschiedliche Rechner) und vergangenen Ereignissen (Methodenaufrufe), ist eher für das eigentliche Monitoring, also das Überwachen eines Programmablaufes geeignet. Eine ungünstige Verteilung der Prozessoren oder Fehler bei der Migration werden hier schnell offensichtlich. Das Sequenzdiagramm hingegen stellt den gesamten Programmablauf zeitlich geordnet dar und bietet somit erweiterte Debug-Möglichkeiten. So lässt sich z.B. auch überprüfen, ob das generierte Sequenzdiagramm dem während der Spezifikation des Programms erstellten Sequenzdiagramm entspricht. Verteilungsaspekte werden im Sequenzdiagramm allerdings nicht berücksichtigt. So ist nicht ersichtlich auf welchem Rechner sich ein Prozessor an einem Punkt in der Vergangenheit befand. Auch die Migrationsbefehle werden im Sequenzdiagramm nicht dargestellt, da sie direkt an den virtuellen Prozessor gerichtet sind und nicht an Objekte innerhalb des Prozessors.

Abbildung 8.6: Der Monitor nach einer Migration



8.2.1 Konfigurationsmanagement

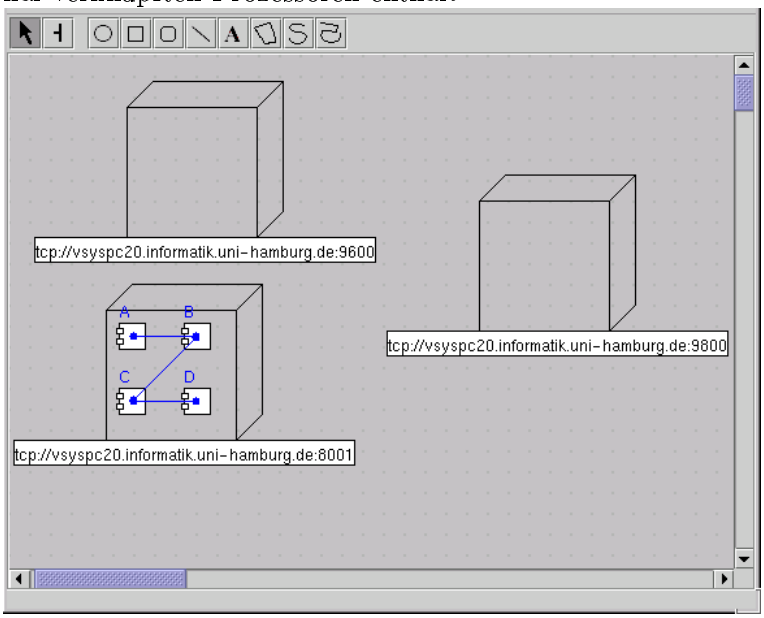
Neben der Möglichkeit die Kommunikation zwischen virtuellen Prozessoren zu überwachen, bietet der Monitor zusätzlich die Möglichkeit virtuelle Prozessoren per *drag-and-drop* zwischen Rechnern zu migrieren. Dies ist jederzeit zur Laufzeit des überwachten Programms möglich. Muss z.B ein Rechner heruntergefahren werden, kann man vorher die Teile eines Programms, die auf dem Rechner ausgeführt werden zu einem anderen Rechner verschieben. Nicht implementiert, aber eine denkbare Erweiterung, ist ein automatisiertes Konfigurationsmanagement. Der Benutzer könnte dann beispielsweise angeben, auf welchen Rechnern sich ein virtueller Prozessor befinden darf und welche Prozessoren zusammengehören. Der Konfigurationsmanager würde dann automatisch für eine optimale Verteilung der Prozessoren, unter Berücksichtigung der Zeit, die ein Prozessor verbraucht und des Kommunikationsaufwandes zwischen Prozessoren, sorgen.

8.2.2 Visualisierung der Komposition

Um die Funktion des Kompositionsmechanismus zu visualisieren, wurde ebenfalls der Monitor verwendet. Der Monitor erkennt Kompositionen und stellt sie grafisch dar. Wenn der Monitor als Konfigurationsmanagementtool eingesetzt wird und einzelne Komponenten verschoben werden, verschieben sich auch die damit verknüpften Komponenten gemäß der in Kapitel 7 beschriebenen Regeln. Dies soll in einem kleinen Beispiel dokumentiert werden. Abbildung 8.7 zeigt drei Knoten von denen einer eine Gruppe von vier bidirektional verknüpften

Prozessoren enthält.

Abbildung 8.7: Ansicht auf drei Knoten, von denen einer eine Gruppe von vier bidirektional verknüpften Prozessoren enthält



Einer dieser Prozessoren soll nun per drag-and-drop auf einen anderen Knoten verschoben werden. Abbildung 8.8 zeigt einen Snapshots zu einem Zeitpunkt, wo einer der Prozessoren gerade verschoben wird, wobei der Maus-Cursor nicht dargestellt ist.

Nachdem der Prozessor auf den neuen Prozessor verschoben worden ist, werden die damit verknüpften Prozessoren automatisch und den Regeln konform nachgezogen, was in der Abbildung 8.9 dargestellt ist.

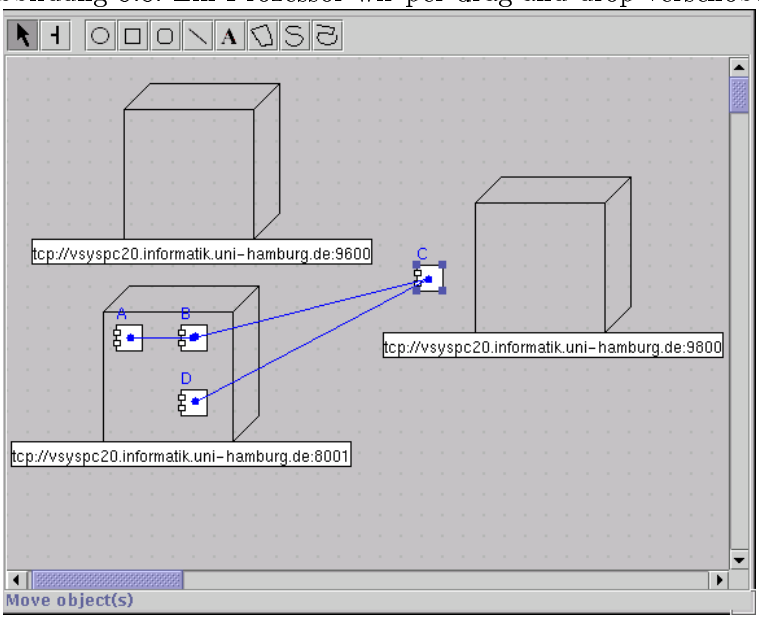
8.2.3 Anbindung des Monitors

Der Monitor ist ein eigenständiges Programm, dass optional zu einem Dejay-Programm hinzugeschaltet werden kann. Ein Dejay-Programm muss für die Verwendung des Monitorings weder neu kompiliert werden, noch sollte es sich während des Monitorings anders verhalten. Der Monitor kann unabhängig von einem Dejay-Programm gestartet werden und das Monitoring der Prozessoren zu einem beliebigen Zeitpunkt während der Programmausführung aufnehmen oder abbrechen. Lediglich der Monitor braucht Referenzen auf alle zu überwachenden Prozessoren.

Voyager bietet zu diesem Zweck einen Naming-Service, mit Hilfe dessen sich Objekte registrieren lassen. Auf diese Objekte kann dann später über den während der Registrierung angegebenen Namen zugegriffen werden. Allerdings hat der Namensraum keine rechnerübergreifende Gültigkeit, sondern gilt immer nur lokal für einen Rechner.

Es lassen sich zwei Fälle voneinander unterscheiden: 1. Der Monitor wird gestartet, es werden Rechner angegeben, die überwacht werden sollen und auf

Abbildung 8.8: Ein Prozessor wir per drag-and-drop verschoben



diesen Rechnern werden von einem Dejay-Programm virtuelle Prozessoren erzeugt. 2. Das Dejay-Programm läuft bereits und der Monitor wird hinzugeschaltet um einzelne Rechner zu überwachen auf denen bereits virtuelle Prozessoren existieren.

Die Anbindung des Monitors erfolgt für beide Fälle mittels folgendem Algorithmus: Wird ein neuer virtueller Prozessor erzeugt, prüft dieser zunächst, ob der Rechner bereits von einem Monitor überwacht wird. Dazu führt er einen Lookup auf den Namen "Monitor" aus. Findet er einen Monitor, benachrichtigt er diesen. Der Monitor setzt den Prozessor daraufhin in den Monitoring-Modus. Als nächstes registriert sich der Prozessor selbst im Namensraum. Es ist möglich, den Namensraum hierarchisch zu gliedern. Dies wird in diesem Fall ausgenutzt und der Prozessor unter "Processors/id" registriert, wobei die id eine eindeutige Integer-Zahl ist, die dem Prozessor während der Erzeugung zugewiesen wird.

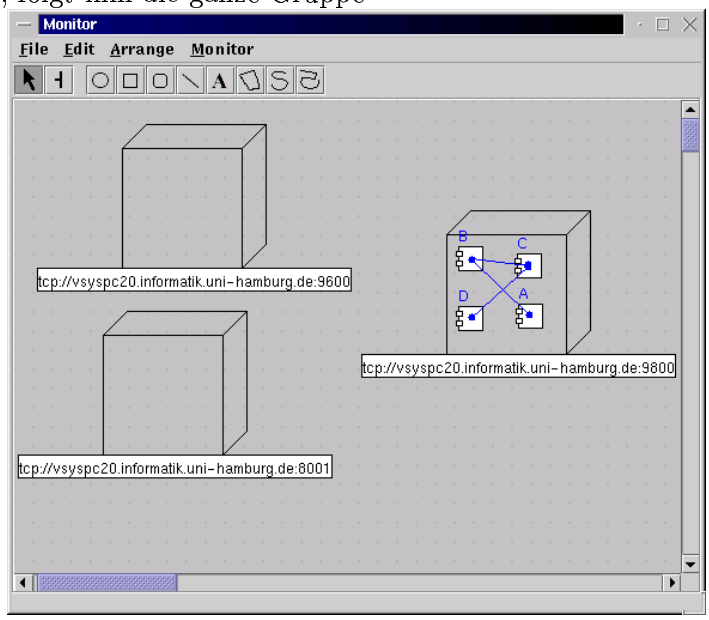
Der Monitor verhält sich folgendermaßen: Soll ein Rechner überwacht werden, registriert sich der Monitor zunächst im Namensraum des entsprechenden Rechners. Danach führt er ein Lookup auf das Verzeichnis "Processors/" im Namensraum des Rechners aus. Als Ergebnis erhält er ein Array mit allen registrierten Prozessoren. Der Monitor setzt daraufhin alle gefundenen Prozessoren in den Monitoring-Modus.

Der Aufruf des Monitors erfolgt durch den Aufruf

```
java dejay.monitor.Monitor [zu überwachende Rechner...]
```

Weitere Rechner können später jederzeit über den Menüpunkt `Monitor|Add Computer` hinzugefügt werden. Existieren auf diesem Rechner schon virtuelle Prozessoren, werden diese automatisch als Komponenten dargestellt. Die Knoten können nach Belieben angeordnet werden.

Abbildung 8.9: Nachdem der Prozessor per Maus auf die rechte Umgebung platziert wurde, folgt ihm die ganze Gruppe



Kapitel 9

Implementierung von Dejay

Nachdem in den vorangegangenen Kapiteln ausführlich auf die zugrunde liegenden Konzepte und auf die Sprache, in der diese umgesetzt sind, eingegangen worden ist, sollen nun noch einige Details der Implementation von Dejay und des zugehörigen Compilers beschrieben werden.

Die Implementation umfasst eine Klassenbibliothek, deren zentraler Bestandteil der virtuelle Prozessor ist, und den Compiler, der aus Dejay Java-Code generiert. Zunächst wird auf die Architektur und Funktionsweise des virtuellen Prozessors in Abschnitt 9.1 eingegangen. Anschließend werden die generellen Mechanismen des Compilers in Abschnitt 9.2 sowie die Realisierung der entfernten Referenzen, die vom Compiler erzeugt werden in Abschnitt 9.3 erläutert. Auf die Implementation der Komposition und der Ausnahmebehandlung wird in Abschnitt 9.4 bzw. 9.5 eingegangen.

9.1 Der virtuelle Prozessor

Der virtuelle Prozessor verwaltet eine Gruppe von Objekten und sorgt für eine von außen konsistente Sicht auf diese Gruppe. Alle Aufrufe von außen an Objekte dieser Gruppe werden an den virtuellen Prozessor geleitet, von ihm entgegengenommen, deren Ausführung abgewickelt und das Ergebnis zurückgeliefert. Seine Aufgaben umfassen die Erzeugung von Objekten, die Verwaltung von Aufrufen, die Garbage-Collection, sowie die konsistente Migration und Persistenz.

Die zentrale Aufgabe des virtuellen Prozessors ist die Verwaltung von Objekten. Damit der virtuelle Prozessor diese Aufgabe erfüllen kann, benötigt er Referenzen auf all jene Objekte, die durch ihn verwaltet werden sollen. Für die Speicherungen dieser Referenzen wird ein Objekt vom Typ ReferenceStorage verwendet. Das entsprechende Objekt wird innerhalb des Kontruktors erzeugt. In diesem Objekt werden nicht nur die eigentlichen Referenzen abgespeichert, sondern gleichzeitig für jede Referenz auch ein Referenzzähler angelegt. Dieser Zähler gibt an, wie viele Dejay-Objekte auf das gespeicherte Objekt verweisen. Durch die Methoden increaseCounter() und decreaseCounter() kann der Wert des Referenzzählers verändert werden. Referenzen und Zähler werden intern in zwei Vektoren gespeichert. Der Zugriff erfolgt mit Hilfe einer so genannten vectorID,

die angibt, an welcher Position die Referenz abgespeichert wurde.

Bei der Erzeugung eines virtuellen Prozessors besteht außerdem die Möglichkeit, zwei Argumente zu übergeben. Diese werden den Instanzvariablen classLoaderSource und debug zugewiesen. Die erste Variable kann die Adresse eines Voyager *Resource Servers* enthalten und wird in der Methode fetchClass() für das Nachladen von .class-Dateien benötigt. Hat die zweite Variable den Wert true, so werden bei der Benutzung des virtuellen Prozessors diverse Statusmeldungen ausgeben.

Für die Erzeugung und Registrierung von Objekten innerhalb des virtuellen Prozessors wird die Methode createObject() durch den Konstruktor einer Dejay-Klasse aufgerufen. Bei ihrem Aufruf werden eine Anzahl von Parametern mit übergeben. Hierbei handelt es um einen String mit dem Namen der Klasse, von der das entfernte Objekt erzeugt werden soll, je eine Liste mit Parametertypen und Konstruktorargumenten und die Adresse des Dejay-Objekts.

Die eigentliche Objekterzeugung erfolgt durch die Verwendung des Reflection-API. Zuvor müssen jedoch bestimmte Voraussetzung geschaffen werden. Zuerst erfolgt eine Aufarbeitung der Parametertypen. Als nächstes wird die Liste der Konstruktorargumente überprüft. Falls sich darunter Dejay-Objekte befinden und der Aufruf aus einer anderen JVM kommt, müssen die Referenzzähler der entsprechenden Dejay-Objekte erhöht werden. Der Grund hierfür ist, dass bei solchen Aufrufen die Argumente nur als Kopie ankommen und sich somit die Anzahl der Dejay-Objekte erhöht. Im darauffolgenden Schritt wird versucht, die .class-Datei des zu erzeugenden Objekts zu laden. Dazu wird die Methode fetchClass() verwendet. Ihr zweistufiger Mechanismus sucht diese Datei zuerst lokal. Wenn dies ohne Erfolg war, wird versucht, die Datei durch den Voyager *ClassLoader* zu laden. Hierbei wird die Instanzvariable classLoadingSource benutzt, welche (möglicherweise) die Adresse eines *Resource Servers* enthält.

Waren die bisherigen Schritte erfolgreich, wird unter Zuhilfenahme des Arrays mit den Parametertypen und der geladenen Klasse ein Constructor-Objekt erzeugt. Dieses wird dann zusammen mit der Liste der Konstruktorargumente an die Methode create() überreicht. Dort wird das Objekt letztendlich erzeugt und als Ergebnis zurückgegeben.

Der letzte Schritt ist die Registrierung des erzeugten Objekts. Es wird hiermit dem virtuellen Prozessor zur Verwaltung übergeben. Dies geschieht durch den Aufruf der register()-Methode. Das erzeugte Objekte wird dabei als Parameter übergeben. Als Ergebnis liefert der Aufruf die vectorID zurück. Mit ihr ist es möglich, das gespeicherte Objekt zu einem späteren Zeitpunkt wiederzufinden. Diese vectorID wird schließlich als Resultat des createObject()-Aufrufs an das aufrufende Dejay-Objekt zurückgeben.

Anschließend können Methoden dieses Objektes aufgerufen werden. Aufrufe von außerhalb des virtuellen Prozessors werden an die execute()-Methode geleitet. Der Parameter Job-Objekt enthält alle hierfür benötigten Informationen. Sie werden zu Beginn der execute()-Methode extrahiert und dann lokalen Variablen zugewiesen. Einige Daten werden aber zuvor noch modifiziert. Dazu gehört zum Beispiel die Liste mit den Parametertypen. Wie schon bei der *Objekterzeugung*, wird auch hier die Methode buildClassArray() benutzt, um wieder ein class-Array zu erhalten. Des weiteren wird die vectorID dazu verwendet, um eine Referenz

auf das entsprechende Objekt zu erhalten.

Der nächste Schritt ist die Überprüfung der Liste mit den Methodenargumenten, wie es auch bei der Liste mit den Konstruktorargumenten der Fall war. Danach wird an einem Method-Objekt, das zuvor erzeugt wurde, die gewünschte Methode aufgerufen. Dies geschieht mit Hilfe des Reflection-API.

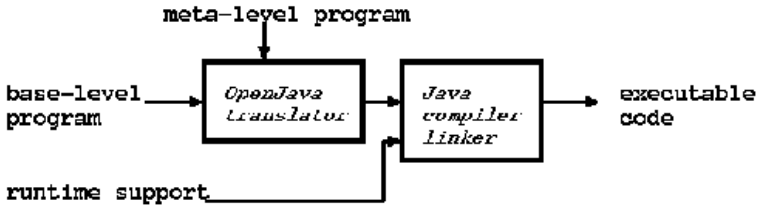
Falls der Aufruf einen Rückgabewert hat, wird dieser dem Objekt result zugewiesen und anschließend für die Rückgabe analysiert. Handelt es sich dabei um einen einfachen Datentyp, wie zum Beispiel int oder bool, wird das Ergebnis einfach durchgereicht. Dies geschieht auch mit Dejay-Objekten, wobei hier aber zusätzlich überprüft wird, ob der Aufruf aus einer anderen JVM kommt. Trifft dies zu, muss der Referenzzähler des Dejay-Objekts erhöht werden. Wenn aber das Ergebnis ein normales Objekt ist, wird versucht, hierfür ein Dejay-Objekt zu erzeugen und dieses anstelle des ursprünglichen Resultats zurückzugeben, so dass die referentielle Integrität gewahrt bleibt. Bei der Erzeugung des Dejay-Objekts wird zuerst geprüft, ob das Objekt schon registriert ist, sonst wird dies jetzt nachgeholt. In beiden Fällen liegt anschließend eine vectorID vor. Als Nächstes wird versucht, die Dejay-Klasse des Objekts zu laden. Wenn dies nicht gelingt, kann kein Dejay-Objekt erzeugt werden. Folglich muss das Ergebnis doch durchgereicht werden. War dagegen das Laden der Dejay-Klasse erfolgreich, wird unter Verwendung des speziellen Proxy-Konstruktors der Dejay-Klasse, ein neues Dejay-Objekt erzeugt. Das Ergebnis wird zurückgegeben und der Aufruf ist damit abgearbeitet.

9.2 Dejay-Compiler

Für die Implementierung des Compilers wurde auf ein existierendes Werkzeug zurückgegriffen, das für die Entwicklung von Java-Dialekten entwickelt wurde. Es handelt sich dabei um ein Werkzeug, dass die Manipulation der Sprache auf einer Meta-Ebene erlaubt. Dieses Konzept wurde im Projekt OpenJava von Michaki Tatsubori entwickelt ([Tatsubori 1997]) und steht frei zur Verfügung. In OpenJava wird durch ein Metaobjekt-Protokoll jeder Bestandteil der Sprache Java auf eine Klasse abgebildet. Ein Parser erzeugt aus einem eingelesenen Quelltext eine Baumstruktur desselben unter Verwendung der Klassen des Metaobjekt-Protokolls. Der Baum und seine Objekte werden Basisprogramm genannt. Jeder Knoten im Baum repräsentiert also einen Bestandteil des Quelltextes als Java-Objekt. Eine Programmierschnittstelle ermöglicht es, auf diese Objekte zuzugreifen und mit weiteren Java-Klassen, so genannten *Metaprogrammen*, zu manipulieren. Die so durchführbaren Veränderungen wirken sich direkt auf den zugrunde liegenden Quelltext aus.

Wird ein Metaprogramm auf ein Basisprogramm angewendet, entsteht ein neues Basisprogramm. Dieses kann durch einen Java-Compiler übersetzt und dann ausgeführt werden. Art und Umfang der Manipulationen sind nicht festgelegt, solange das geänderte Basisprogramm wieder ein Java-Programm ist. Für die Benutzung von OpenJava in Dejay ist besonders wichtig, dass die Manipulation des Basisprogramms von der eigentlichen Übersetzung des Programms getrennt ist. Dadurch ist es möglich Sprachelemente im Basisprogramm zu nut-

Abbildung 9.1: OpenJava-Compiler



zen, die in Java nicht zur Verfügung stehen, solange sie im Metaobjekt-Protokoll abbildbar sind.

Der Dejay-Compiler besteht aus zwei Teilen, einem vorbereitenden Analysewerkzeug, das die Dateien auf die Verarbeitung mit OpenJava vorbereitet, und einem Sprachcompiler, der das Metaprogramm enthält. Die Abarbeitung und Steuerung wird durch ein gemeinsames Skript gesteuert. Die Realisierung des Skriptes ist plattformabhängig für die Unix-Kommandozeilenumgebung bash erfolgt. Damit ist der Dejay-Compiler in seiner bestehenden Version nur für Unix-Systeme nutzbar, eine vollständige Umsetzung in Java ist aber möglich. Dadurch ist es allerdings möglich, das aus Sicht des Benutzers das Skript dejayc in einer ähnlichen Weise wie der Java-Compiler javac verwendet werden kann.

Das Skript startet zunächst das Java-Programm DistributableClass, dass eine Voranalyse der übergebenen Klasse und die Erzeugung einer neuen Klasse für die entsprechende entfernte Referenz vornimmt. Die übergebene Klasse wird um zusätzliche Informationen für die nachfolgende Bearbeitung mit OpenJava erweitert, die neu erzeugte Klasse enthält die komplette Schnittstelle der übergebenen Klasse, Informationen für die weitere Bearbeitung und einige nötige Import-Anweisungen. DistributableClass macht intensiven Gebrauch von der *Java Core Reflection API* (siehe [Sun 1999]). Diese ermöglicht es, zur Laufzeit eines Java-Programms detaillierte Informationen über eine Java-Klasse abzufragen. Neben der Signatur der Klasse und ihren Vererbungsbeziehungen sind sowohl die öffentliche als auch die private Schnittstelle der Klasse zugänglich. DistributableClass untersucht Bytecode- oder Quelltext-Dateien. Die Java Core Reflection API wird benutzt, um aus den Bytecode-Dateien die Informationen über die Klassen zu erhalten, die für die Erzeugung einer entfernten Referenz-Klasse notwendig sind. Bei Quelltext-Dateien wird diese Aufgabe über einfache Texterkennung wahrgenommen.

Ergebnis dieser Vorverarbeitung ist einerseits die manipulierte Eingabedatei sowie eine neue Quelltext-Datei, die nur Signatur und Schnittstelle der übergebenen Klasse enthält. Der Name der neuen Datei erweitert den ursprünglichen Namen um den Prefix Dj. Liegt die Beschreibung der übergebenen Klasse als Quelltext vor, wird dieser um einen speziellen Eintrag für OpenJava erweitert. Der Eintrag kennzeichnet das Metalevel-Programm, das auf den Quelltext angewendet werden soll. Für Quelltext-Dateien handelt es sich dabei um das Programm DjCodeCompiler. Auch die neu erzeugte Datei enthält einen entsprechenden Eintrag für OpenJava. Hier handelt es sich um das Programm DjCreateProxy. Nachdem die Ausgangsklasse vorbereitet und eine neue Klasse für die entfernte Referenz erzeugt wurde, wird durch das Kommandozeilen-Skript die

Übersetzung der Dejay-Quelltexte in Java-Quelltexte durch OpenJava ausgelöst.

Eine der Hauptaufgaben von DjCodeCompiler ist die Aufbereitung der Methode main(). Die einfache Ausführung dieser Methode würde wichtige Eigenschaften von Dejay unterlaufen. In Dejay dürfen Objekte nur innerhalb von virtuellen Prozessoren existieren. Objekte, die in main() erzeugt werden, würden aber in keinem virtuellen Prozessor liegen. Daher wird die Methode main() vollständig ersetzt. Anstelle der eigentlichen Befehle in main() wird ein standardisierter Befehlsblock zum Start eines Dejay-Programms in die Methode eingetragen. Die ursprünglichen Befehle werden in eine neue Methode djmain() übertragen. Um ein Dejay-Programm zu starten, wird zunächst eine Voyager-Umgebung gestartet. In dieser wird ein initialer virtueller Prozessor erzeugt, der wiederum dafür Sorge trägt, dass die Methode djmain() aufgerufen und bearbeitet wird. Somit ist sichergestellt, dass alle im ursprünglichen main() erzeugten Objekte innerhalb eines virtuellen Prozessors erzeugt werden. Die eigentlichen Befehle werden dabei nicht verändert.

Des weiteren analysiert DjCodeCompiler die Implementation aller Methoden der zu übersetzenden Klasse. Die einzelnen Befehle werden dabei solange in ihre Bestandteile zerlegt, bis sie auf atomare Bestandteile heruntergebrochen worden sind. Dies dient der Umsetzung von dejay-spezifischen Sprachelementen in Java-Sprachelemente. Die Fähigkeiten von Dejay werden im wesentlichen über eine Klassenbibliothek bereitgestellt. Für die zukünftige Entwicklung von Dejay ist es aber durchaus möglich, neue syntaktische Änderungen in die Sprache zu integrieren. Durch die bereits vorgenommene Zerlegung der Befehle ist eine derartige Erweiterung mit verhältnismäßig geringem Aufwand realisierbar.

9.3 Entfernte Referenzen

Die Hauptaufgabe des Compilers ist die Erzeugung der entfernten Referenzen. Die durch DistributableClass vorgenommene Vorbereitung einer neuen Klasse für eine entfernte Referenz beinhaltet ausschließlich die Bereitstellung der Signatur der Klasse und ihrer Schnittstelle. Um daraus eine funktionsfähige Klasse zu erzeugen, müssen alle Methoden mit einer Implementation versehen werden. Zudem ist noch eine Anpassung von Klassennamen und Vererbungsbeziehungen erforderlich. Letztlich benötigt eine entfernte Referenz noch eine Reihe von zusätzlichen Methoden, um ihre Eigenschaften vollständig zur Verfügung stellen zu können. All diese Aufgaben erledigt die Übersetzung der neuen Quelltext-Datei mit dem Metalevel-Programm DjCreateProxy.

Der Klassenname der entfernten Referenz besteht aus dem ursprünglichen Klassennamen und dem Prefix Dj. Auch die Vererbungsbeziehungen verändern sich. Die entfernte Referenz ist eine Ableitung der Ausgangsklasse. Zudem implementiert sie die Schnittstelle RemoteReference, die die zusätzlichen Methoden für entfernte Referenzen festlegt. Durch die geänderte Vererbungsstruktur ist es einerseits möglich, eine entfernte Referenz wie eine Instanz der Ausgangsklasse zu behandeln. Andererseits haben alle entfernten Referenzen eine gemeinsame Basis, über die sie von Instanzen ihrer Ausgangsklassen unterscheidbar sind.

```
public class DjA extends A implements dejay.base.RemoteReference {
```

```
  private int vectorID;
  private dejay.base.DjProcessor processor;
  private boolean isMoving = false;
  private java.lang.Object result = null;
  ...
}
```

Signatur und Felder einer entfernten Referenz

Jede entfernte Referenz benötigt zudem noch einen Satz von Instanzvariablen. Diese dienen der Identifikation des entfernten Objektes (vectorID und processor), der Erhaltung der Konsistenz der verteilten Speicherverwaltung während einer Migration (isMoving) und der Bearbeitung von asynchronen Aufrufen (result).

Auch die entfernten Referenzen benötigen eigene Konstruktoren. Mindestausstattung einer entfernten Referenz sind ein Konstruktor mit der Referenz auf einen virtuellen Prozessor und ein Konstruktor mit den Verbindungsinformationen zu einem entfernten Objekt. Der erstgenannte Konstruktor ist dabei eine Erweiterung des Standardkonstruktors der Ausgangsklasse. Er erzeugt ein entferntes Objekt über eben diesen Standardkonstruktor in dem angegebenen virtuellen Prozessor und speichert die Verbindungsinformationen zu diesem Objekt in den Instanzvariablen der zu erzeugenden entfernten Referenz.

```
public DjA( DjProcessor p ) {
  this.processor = p;
  String[] parameterTypes = new String[]{ };
  Object[] arguments = new Object[]{ };
  try {
    vectorID = processor.createObject("A", parameterTypes,
            arguments, getProxyAddress());
  } catch ( Exception e ) {
    e.printStackTrace();
  }
}

public DjA( DjProcessor p, int vectorID ) {
  this.processor = p;
  this.vectorID = vectorID;
  increaseProxyCounter();
}
```

Quelltext der Standardkonstruktoren von entfernten Referenzen

Für jeden weiteren Konstruktor der Ausgangsklasse wird ein zusätzlicher Konstruktor für die entfernte Referenz erzeugt. Die Parameterliste wird dabei um die Referenz auf einen virtuellen Prozessor erweitert. Die Aufgabe des Konstruktors ist auch hier die Erzeugung eines entfernten Objektes unter Verwendung des zugrunde liegenden Konstruktors der Ausgangsklasse im übergebenen virtuellen Prozessor.

9.3.1 Anpassung ererbter Methoden

Alle entfernten Methodenaufrufe sollen von den entfernten Referenzen aufgenommen und an das entsprechende entfernte Objekt weitergeleitet werden. Die Implementation der Methode beim entfernten Objekt ist den entfernten Referenzen unbekannt. Die Implementation der Methoden bei den entfernten Referenzen beschränkt sich dementsprechend auf die Aufbereitung des Aufrufs, dessen Durchführung und die Behandlung des Ergebnisses. Die Logik dieser ist daher für alle Methoden gleich. Lediglich die genutzten Werte und Typen unterscheiden sich von Methode zu Methode. Daher liegt es nahe, die Implementation einer derartigen Methode allgemein zu definieren und die variablen Bestandteile für jede Methode anzupassen. Als ererbte Methoden sind dabei aber nicht nur die Methoden zu verstehen, die in der Schnittstelle der Ausgangsklasse stehen. Vielmehr werden alle Methoden der kompletten Vererbungshierarchie bis hin zur Klasse Object aufgeführt und behandelt.

Die variablen Bestandteile gehen bereits aus der Signatur der Methode hervor. Methodenname, Typ des Rückgabewertes und Parameter der Methode stehen hier zur Verfügung. Auch das zu rufende entfernte Objekt ist durch die Verbindung zwischen ihm und der rufenden entfernten Referenz bekannt. Für jede ererbte Methode wird daher bei der Übersetzung der Quelltext-Datei für die entfernte Referenz beim Metalevel-Programm DjCreateProxy die Methode translateMethod() aufgerufen. Diese Methode erweitert die Parameterliste der zu bearbeitenden Methode zunächst um einen Eintrag der Klasse Messenger. Die Klasse Messenger kapselt die Information darüber, ob der Aufruf synchron, asynchron oder als Einwegaufruf ausgeführt werden soll.

```
public B getB( dejay.base.Messenger messageType ) throws de-
jay.base.NoRemoteReferenceClassException {
  try {
    if (!(result == null)) {
      ((DejayResult) result).waitForResult();
      DjA proxy = (DjA) ((DejayResult) result).getResult().readObject();
      this.processor = proxy.getProcessor();
      this.vectorID = proxy.getVectorID();
      increaseProxyCounter();
      result = null;
    }
    String[] parameterTypes = new String[]{ };
    Object[] arguments = new Object[]{ };
    dejay.base.Job job = new dejay.base.Job( "getB", parameterTypes, arguments,
vectorID, getProxyAddress(), Dejay.LocalReferenceAllowed );
    if (messageType.equals( Dejay.ASYNC )) {
      job.setReturnReference( Dejay.RemoteReferenceOnly );
      Class tmpClass = Class.forName( "DjB" );
      java.lang.reflect.Constructor tmpCon = tmp-
Class.getConstructor( new Class[]{ this.processor.getClass() } );
      Object fuValue = tmpCon.newInstance( new Object[]{ this.processor } );
      java.lang.reflect.Method tmpMeth = tmpClass.getMethod( "getDjResult",
```

```
            new Class[]{ } );
        DejayResult tmpResult = (DejayResult) tmpMeth.invoke( fuValue,
              new Object[]{ } );
        tmpResult.setResult( Future.invoke( processor, "execute",
              new Object[]{ job } ) );
        return (B) fuValue;
      } else {
        if (messageType.equals( Dejay.SYNC )) {
          B syValue = null;
          result = new DejayResult();
          ((DejayResult) result).setResult( Sync.invoke( processor, "execute",
               new Object[]{ job } ) );
          syValue = (B) ((DejayResult) result).getResult().readObject();
          result = null;
          return syValue;
        }
      }
    } catch ( ClassNotFoundException c ) {
      c.printStackTrace();
      throw new NoRemoteReferenceClassException( "No remote reference class
          description found for class B" );
    } catch ( Exception e ) {
      e.printStackTrace();
    }
    return null;
}
```

Quelltext einer übersetzten Methode mit einstellbarem Ausführungsverhalten

Entscheidenden Einfluss auf die Implementation der Methode hat lediglich der Typ des Rückgabewertes. Für die primitiven Datentypen boolean, int, long, char, float und double ist es nicht möglich, die Methode asynchron auszuführen. Eine Einwegausführung (*oneway*) ist hingegen ausschließlich mit Methoden ohne Rückgabewert erlaubt. Ist der Rückgabewert ein Objekt, sind sowohl synchrone als auch asynchrone Ausführung möglich. Die asynchrone Ausführung unterliegt zur Laufzeit allerdings einer strengeren Überprüfung. Nur bei Rückgabewerten, für die eine entfernte Referenz erzeugt werden kann oder die selbst eine entfernte Referenz sind, ist die asynchrone Ausführung zugelassen. Um dies zu überprüfen, ein solches Objekt zu erzeugen und sein Ergebnis-Objekt zu benutzen, wird auch an dieser Stelle wieder sehr stark Gebrauch von der Java Core Reflection API gemacht. Im Gegensatz zur Nutzung in DistributableClass wird die Überprüfung hier zur Laufzeit des Dejay-Programms durchgeführt.

Ein Aufruf an das entfernte Objekt findet nur indirekt statt. Stattdessen wird eine Instanz der Klasse Job erzeugt, mit den nötigen Informationen versehen und an den verbundenen virtuellen Prozessor geschickt. Erst dieser führt den eigentlichen Aufruf durch. Die Verbindung zum virtuellen Prozessor wird

mittels Voyager realisiert. Dies ist eine der wenigen Stellen, an denen die Existenz von Voyager tatsächlich sichtbar wird. Neben der Erzeugung eines neuen virtuellen Prozessors erscheint Voyager ansonsten nur noch beim Start eines Dejay-Programms. Die Verschickung des Aufrufs, die Weiterleitung eventuell auftretender Ausnahmen auf der gerufenen Seite und die Rücksendung der Antwort wird vollständig an Voyager delegiert.

Um die Ausführungsreihenfolge bei mehreren, aufeinander folgenden asynchronen Aufrufen auf das gleiche entfernte Objekt zumindest innerhalb eines virtuellen Prozessors zu erhalten, wird die Antwort eines noch unbeantworteten Aufrufs abgewartet. Erst nach Eintreffen der Antwort kann der neue Aufruf bearbeitet werden.

Nach Bearbeitung durch translateMethod() hat die bearbeitete Methode eine Implementation, die eine Weiterleitung eingehender Aufrufe zu einem entfernten Objekt ermöglicht. Durch die Erweiterung der Parameterliste ist aus dieser Methode aus Sicht der Vererbung allerdings eine neue Methode geworden. Ein Aufruf ohne zusätzlichen Messenger-Parameter würde somit die ursprüngliche, ererbte Implementation der Methode rufen. Neben der unerwünschten lokalen Ausführung können durch die geänderte Struktur der Klasse unerwartete Fehler auftreten. Daher muss die Methode mit ihrer ursprünglichen Signatur wieder hergestellt und mit einer neuen Implementation versehen werden. Dazu wird die Methode addWithDefaultMessenger() aufgerufen.

```
public B getB() {
  Messenger messageType = Dejay.SYNC;
  try {
    return getB( messageType );
  } catch ( NoRemoteReferenceClassException e ) {
    e.printStackTrace();
  }
  return null;
}
```

Quelltext einer Methode mit standardisiertem Ausführungsverhalten

Diese Methode erzeugt eine zusätzliche Methode in der zu bearbeitenden Klasse mit der originalen Signatur. Ein Aufruf der Methode soll ebenfalls zu einer Weiterleitung des Aufrufs an das verbundene, entfernte Objekt führen. Als Aufrufsemantik wird dabei eine synchrone Bearbeitung angenommen, da dies in Dejay als Standard benutzt wird. Der Einfachheit halber wird eine Instanz der Klasse Messenger erzeugt, die eine synchrone Ausführung zur Folge hat. Diese wird gemeinsam mit den übrigen Parametern an die oben beschriebene Methode weitergeleitet.

9.3.2 Zusätzliche Methoden

Um die volle Funktionalität von Dejay auszunutzen, benötigen die entfernten Referenzen noch eine Reihe zusätzlicher Methoden. Die allgemein gültigen Methoden werden in der Schnittstellenbeschreibung RemoteReference definiert. Alle entfernten Referenzen müssen die dort definierten Methoden implementieren.

RemoteReference dient somit gleichzeitig als gemeinsame Basis und Identifikationsmerkmal aller entfernten Referenzen.

```
public interface RemoteReference extends IMobile, Serializable {
  public void increaseProxyCounter();
  public String getRemoteAddress();
  public String getProxyAddress();
  public void moveTo(String target);
  public void moveTo(RemoteReference target);
  public void moveTo(DjProcessor target);
  public void printVectorID(String objectName);
}
```

Quelltext der Schnittstellenbeschreibung RemoteReference

Die Methode increaseProxyCounter() erhöht die Anzahl der entfernten Referenzen auf das verbundene entfernte Objekt um eins. Da diese Methode direkten Einfluss auf die verteilte Speicherverwaltung hat, sollte sie nur verwendet werden, wenn man die Auswirkungen genau abschätzen kann. Die Migration von Objekten kann auf verschiedene Weise angefordert werden. Die Methode moveTo() kann daher unterschiedliche Parameter verwenden. Gemeinsam ist allen Versionen, dass der referenzierte virtuelle Prozessor zu der Adresse bewegt wird, an der sich das Parameter-Objekt befindet. Die Migration wird auch nicht von der entfernten Referenz ausgeführt. Wie alle anderen Aufrufe auch, wird die Anforderung der Migration von der entfernten Referenz entgegengenommen. Aufgrund des angegebenen Parameter-Objektes wird die Zieladresse identifiziert. Anschließend ruft die entfernte Referenz die Methode moveTo() an dem verbundenen virtuellen Prozessor. Die eigentliche Migration wird dabei an Voyager delegiert. Durch die speziellen Eigenschaften des virtuellen Prozessors bewegt sich dadurch die gesamte Objektgruppe, ohne das Anpassungen an Voyager notwendig sind. Mit den Methoden getRemoteAddress() und getProxyAddress() kann die aktuelle Adresse des entfernten Objektes beziehungsweise der entfernten Referenz herausgefunden werden.

Die Adresse der entfernten Referenz zu bestimmen, stellt sich dabei in der Implementation als interessantes Problem dar. Ein Voyager-Objekt ist dazu in der Lage herauszufinden, welche Adresse die Voyager-Umgebung hat, in der es sich befindet. Voyager-Objekte sind in Dejay aber nur die virtuellen Prozessoren. Entfernte Referenzen sind weder Voyager-Objekte, noch kennen sie den virtuellen Prozessor in dem sie selbst leben. Einzig der virtuelle Prozessor des entfernten Objektes ist ihnen bekannt. Über den gerade aktiven Thread lässt sich aber auch eine Verbindung zum ausführenden Prozessor herstellen. Dieser kann als Voyager-Objekt wiederum seine Adresse in Erfahrung bringen.

Die Methode printVectorID() dient dazu, die Identifikation des entfernten Objektes auszugeben. Der virtuelle Prozessor lagert alle in ihm erzeugten Objekte in einem internen Behälter. Mit printVectorID() erhält der Aufrufer die eindeutige Identifikation des entfernten Objektes innerhalb des Behälters. Diese Methode dient vorwiegend der Fehlersuche in den Basisklassen von Dejay und hat für die Entwicklung eines Dejay-Programms untergeordnete Bedeutung.

Es gibt aber auch noch eine Reihe von Methoden, die aus technischen Gründen nicht in RemoteReference aufgenommen werden konnten, aber trotzdem in der Schnittstelle der entfernten Referenz auftauchen müssen. Diese Methoden lassen sich in drei Gruppen aufteilen. Die erste Gruppe besteht aus Methoden deren Signatur vom Klassennamen abhängt oder nicht in einer Schnittstellenbeschreibung auftauchen darf, während die zweite Gruppe zur Erhaltung eines konsistenten Objektzustandes und die dritte Gruppe der Steuerung von wait-by-necessity dient.

```
public DjA( DjA copyObject )
protected void finalize()
public A getCopy()
protected dejay.base.DjProcessor getProcessor()
protected int getVectorID()
public boolean isRemoteIdentical( DjA opponent )
```

Signaturen der Methoden der ersten Gruppe

Zur ersten Gruppe zählt ein so genannter Copy-Konstruktor. Auf Basis des Zustandes eines Objektes der gleichen Klasse erzeugt ein Copy-Konstruktor eine inhaltlich identische Kopie mit anderer Identität. Der Copy-Konstruktor kann explizit oder implizit, beispielsweise durch eine Zuweisung, aufgerufen werden. Bei entfernten Referenzen kopiert der Copy-Konstruktor die Informationen über die Verbindung zum entfernten Objekt vom Parameter-Objekt auf das neue Objekt. Zudem wird die Anzahl der entfernten Verbindungen auf das entfernte Objekt angepasst, um eine korrekte Funktion der verteilten Speicherverwaltung zu gewährleisten.

Ein einfacher Vergleich auf Gleichheit mittels des Operators == überprüft nur die Gleichheit der entfernten Referenzen. Die Methode equals() hingegen prüft, ob zwei entfernte Objekte den gleichen Inhalt haben. Es fehlt somit ein Vergleich darauf, ob zwei entfernte Referenzen auf das gleiche entfernte Objekt zeigen. Dazu wird die Methode isRemoteIdentical() implementiert. Geprüft wird hierbei, ob die Verbindungsdaten der entfernten Referenzen den gleichen Inhalt haben. Ein entfernter Zugriff ist dazu nicht notwendig. Da der Typ des Vergleichsobjekts von der aktuellen Klasse abhängt, zählt auch diese Methode zur ersten Gruppe. Auch die Methode getCopy() gehört zur ersten Gruppe, weil der Typ des Rückgabewertes von der Ausgangsklasse abhängt. Die Methode erzeugt lokal eine Kopie des entfernten Objektes. Das entfernte Objekt wird dabei nicht migriert.

Die Methoden finalize(), getProcessor() und getVectorID() sind allesamt als protected gekennzeichnet und dürfen daher nicht in einer Schnittstellenbeschreibung auftauchen. Die Methode finalize() sorgt dafür, dass vor Zerstörung der entfernten Referenz die Anzahl der entfernten Verbindungen auf das entfernte Objekt angepasst wird. Die Methoden getProcessor() und getVectorID() bieten lesenden Zugriff auf die einzelnen Bestandteile der Verbindungsdaten und sollten daher nicht von außen zugreifbar sein.

```
public void preDeparture( java.lang.String source, java.lang.String destination )
```

```
public void postDeparture()
public void preArrival()
public void postArrival()
```

Signaturen der Methoden der zweiten Gruppe

Die zweite Gruppe sorgt für einen konsistenten Objektzustand während einer Migration. Voyager stellt dafür die Methoden preDeparture(), postDeparture(), preArrival() und postArrival() zur Verfügung. Jede dieser Methoden ermöglicht es, nötige Befehle zu einem definierten Zeitpunkt auszuführen. Die Methode preDeparture() blockiert solange, bis ein noch unbeantworteter asynchroner Aufruf auf das entfernte Objekt beantwortet ist. Erst danach kann die entfernte Referenz migriert werden. Die Methode postArrival() gibt das Objekt wieder zur weiteren Verwendung frei, nachdem es am Zielort angelangt ist. Die weiteren Methoden sind in Dejay derzeit ohne Bedeutung und daher leer implementiert.

```
public boolean waitForResult()
public dejay.base.DejayResult getDjResult()
public boolean isAvailable()
```

Signaturen der Methoden der dritten Gruppe

Die dritte Gruppe steuert das Verhalten der entfernten Referenz als Ergebnis eines asynchronen Aufrufs. Die Methode getDjResult() ermöglicht den Zugriff auf das interne Ergebnis-Objekt der entfernten Referenz. Innerhalb eines asynchronen Aufrufs wird dieses Ergebnis-Objekt mit dem Aufruf verbunden und erhält später die Antwort. Mit isAvailable() wird der Status des internen Ergebnis-Objektes abgefragt. Ist die Antwort bereits eingetroffen, lautet das Ergebnis true, ansonsten false. Die Methode waitForResult() blockiert solange, bis das Ergebnis eingetroffen ist. Dadurch ist es dem Entwickler eines Dejay-Programms möglich, eine explizite Synchronisation mit dem Eintreffen des Ergebnisses eines asynchronen Aufrufs vorzunehmen.

9.4 Komposition in Dejay

In diesem Abschnitt soll auf einige Details der Realisierung der Komposition eingegangen werden. Zu deren Implementierung war im wesentlichen eine Erweiterung der Klassenhierarchie um vier zusätzliche Klassen nötig. Für das Konzept des Relokators wurde eine abstrakte Klasse, die Klasse RelocatorType eingeführt, von der für jeden der Relokatorbeziehungen eine konkrete Klasse eingeführt wurden, die Klassen RubberPull, UniPull und BiPull. Die eigentliche Herausforderung bestand aber in der Gewährleistung der Einhaltung der durch diese Beziehungen eingeführten Regeln. Daher soll auf diese Aspekte an dieser Stelle eingegangen werden.

Locking

Die Verkettung von Prozessoren sowie die Bewegung sind nicht-atomare Aktionen, bei denen konkurrierend auf Ressourcen zugegriffen wird. Dies macht die

Einführung eines Lockingmechanismus notwendig, der die Atomarität von Kompositionsoperationen sicherstellt. Die Verknüpfung mittels BiPull beispielsweise garantiert, dass zwei Prozessoren sich immer in der selben Voyager-Laufzeitumgebung aufhalten. Derartige Garantien müssen auch durchgesetzt werden. Durch die Nebenläufigkeit könnte es durchaus passieren, dass ein Prozessor A den Auftrag erhält, sich mit (dem lokal vorliegenden) B zu verknüpfen, während ein Objekt innerhalb von C die Migration auf einen anderen Rechner plant. Der Fall, dass A zunächst überprüft, ob B lokal ist (und eine positive Antwort erhält), dann B sich fortbewegt, und erst danach A und B verknüpft werden, muss ausgeschlossen werden. Ebenfalls denkbar ist, dass innerhalb einer BiPull-verketteten Gruppe zwei Prozessoren gleichzeitig den Auftrag bekommen, zu verschiedenen Laufzeitumgebungen zu migrieren, und durch rekursive unkontrollierte Migration am Ende die eine Hälfte der Gruppe auf einer, und die andere Hälfte auf der anderen Umgebung landet. Für beide Fälle muss ein Mechanismus eingeführt werden, um die Garantien, die dem Anwender gegeben werden, erfüllen zu können. Entweder müssen die Befehle serialisiert werden, oder dem einen der beiden Befehle muss die derzeitige Unerfüllbarkeit signalisiert werden.

Daher ist ein zweiphasiges Locking eingeführt worden. Locking birgt die Gefahr von Deadlocks - um diese zu umgehen, können beispielsweise Deadlock Prevention oder Deadlock Detection Mechanismen eingesetzt werden. Aufgrund des hohen Aufwandes für Deadlock Detection wurde in dieser Arbeit das leichter zu implementierende Deadlock Prevention eingesetzt: Wenn festgestellt wird, dass Ressourcen bereits anderweitig gesperrt sind, wird nicht auf die Freigabe derselben gewartet, sondern es werden alle eigenen Ressourcen freigegeben und der Methodenaufruf mit einer Exception abgebrochen.

Bewegung

Innerhalb verknüpfter Gruppen lassen sich kleinere Untergruppen ausmachen, innerhalb derer alle Prozessoren durch die "harten" Verknüpfungen UniPull und BiPull transitiv miteinander verbunden sind. Diese sollen im Folgenden als Kerne bezeichnet werden. Innerhalb solcher Kerne können zusätzlich auch RubberPulls vorhanden sein, ändern aber nichts am Charakter des Kerns. Kerne können untereinander mittels RubberPull verbunden sein, und ergeben so eine Gesamtgruppe.

Gemäß Definition kann ein Kern migrieren, wenn keines ihrer Mitglieder fixiert bzw. anderweitig reserviert ist. Daher werden beim Locking zunächst alle Elemente eines Kerns mit einem eindeutigen Key (welcher von dem Prozessor generiert wird, an dem der Move ausgeführt wurde) gesperrt. Eventuelle Verbindungen zu anderen Kernen werden temporär gespeichert. Wenn klar ist, dass die Gruppe einen fixierten Prozessor enthält, oder mindestens ein Element mit einem anderen Key gesperrt wurde, wird das Locking dieser Gruppe beendet, und die reservierten Elemente werden freigegeben. Kann dagegen die gesamte Gruppe reserviert werden, wird sukzessive versucht, auch die per RubberPull verbundenen Nachbargruppen zu allozieren. Falls eine Nachbargruppe nicht reservierbar ist, wird die von ihr geworfene Exception gefangen und die Gruppe wird ignoriert.

Nachdem das Locking beendet ist, wird vom Ausgangsprozessor aus die Bewegungssequenz gestartet. Dabei werden rekursiv alle verknüpften Prozessoren gefragt, ob sie mit dem urprünglichen Key gesperrt wurden. Wenn ja, wird auf diesen Prozessor der rekursive moveTo-Befehl angewandt, ansonsten wird der Prozessor ignoriert. Ein dabei auftretendes Problem ist, dass, wenn mehrere RubberPulls die selben Gruppen verbinden, der Test mehrmals ausgeführt werden muss. Andererseits ist es wenig sinnvoll, dass ein Entwickler im Layout einer Anwendung solche Mehrfachverknüpfungen vornimmt, daher wird dieser Fall nicht gesondert behandelt oder optimiert. Nach der Migration eines jeden Prozessors wird per Callback-Methode die Sperrung aufgehoben.

Wenn zwei Prozessoren verknüpft werden sollen, werden sie erst gesperrt, dann die Verknüpfung angelegt, und danach die Reservierung aufgehoben. Falls sich einer der beiden Prozessoren nicht sperren lässt, schlägt die Verknüpfung mit einer Exception fehl.

Gruppierung

Jeder Prozessor verwaltet selber, mit welchen Prozessoren er in welcher Verbindung steht. Es gibt, wie es sich für ein verteiltes System ratsam ist, keine zentrale Instanz. Dies bedeutet aber auch, dass alle Verknüpfungen doppelt vorliegen müssen, in jedem Prozessor einmal. Ein Prozessor verweist auf seinen UniPull-Relokator, damit sichergestellt werden kann, dass er nicht zu Zweien zugeordnet wird, und der referenzierte Prozessor kennt den mit ihm verknüpften Prozessor, um es im Falle einer Bewegung benachrichtigen zu können. Ebenso wird bei BiPull in beiden Prozessoren der jeweils andere gespeichert. Es gibt daher in jedem Prozessor folgende Datenstrukturen:

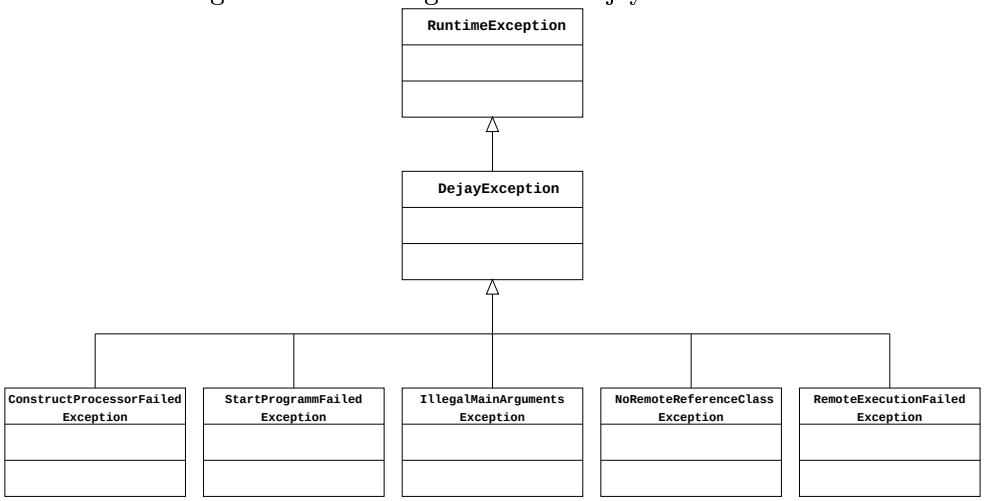
```
private Vector uniPullVec;
private DjProcessor uniPulledByProcessor;
private Vector biPullVec;
private Vector rubberPullVec;
private Vector rubberPulledVec;
```

9.5 Ausnahmebehandlung

Zu den Aufgaben einer verteilten Programmiersprache zählt auch die Behandlung von Fehlern, die zur Laufzeit der Anwendung auftreten. Java bietet zu diesem Zweck eine Ausnahmebehandlung an. Dejay bedient sich dieses Konzeptes und erweitert es um Funktionalität, die eine verteilte Ausnahmebehandlung ermöglicht. Unterstützt wird Dejay dabei durch Voyager. In Voyager sind bereits Mechanismen integriert, die die Weiterleitung von entfernt auftretenden Ausnahmen ermöglicht. Die Ausnahme wird dabei auf der entfernten Seite geworfen, um die dort vorhandene Ausnahmebehandlung nicht auszuhebeln. Gibt es keine angemessene Behandlung der Ausnahme, wird sie von der Voyager-Umgebung abgefangen. Diese verpackt die Ausnahme, verschickt sie an die rufende Seite und wirft sie dort erneut. Zudem stellt Voyager eine Reihe von Ausnahmen zur

Verfügung, die eine Vielzahl von Fehlerszenarien während entfernter Aufrufe abdecken.

Abbildung 9.2: Klassendiagramm der Dejay-Ausnahmenklassen



Unbehandelt bleibt dabei eine Gruppe von Fehlern, die durch die Verwendung von speziellen Dejay-Eigenschaften auftreten können. Um diese Fehler ebenfalls in entsprechenden Ausnahmen zu behandeln, wurde für Dejay eine eigene Hierarchie von Ausnahmen entwickelt, die sich in die bestehende Hierarchie von Java-Ausnahmen nahtlos einbindet. Basisklasse dieser Hierarchie ist die Klasse java.lang.RuntimeException. Die Besonderheit dieser Ausnahme liegt darin, dass sie und alle von ihr erbenden Ausnahmen nicht in der Signatur der Methode auftauchen müssen. Da es sich um Laufzeitbedingungen handelt, können sie von jeder Methode geworfen werden. Neben den vorgegebenen Ausnahmen müssen keine weiteren mehr in der Signatur der Methode definiert werden. RuntimeException ist Bestandteil der Java-Bibliotheken.

Als gemeinsame Basis aller Dejay-Ausnahmen dient die Klasse DejayException. Sie ist von RuntimeException abgeleitet. DejayException bietet keine weitergehende Funktionalität, sondern dient lediglich als übergeordnetes Identifikationsmerkmal für alle Ausnahmen, die in Dejay definiert werden. So ist es dem Programmierer leicht möglich eine Behandlung aller Dejay-Ausnahmen zu programmieren, ohne diese einzeln angeben zu müssen. Alle speziellen Dejay-Ausnahmen werden von DejayException abgeleitet. Entsprechend der Vorgehensweise in Java erweitern diese Ausnahmen nicht die Funktionalität der Basisklasse. Sie dienen über ihren eigenständigen Typ lediglich als Identifikation für eine bestimmte Klasse von Ausnahmen. Das obige Klassendiagramm zeigt die Vererbungshierarchie der derzeit existierenden Dejay-Ausnahmen.

Kapitel 10

Vergleich mit anderen Forschungsprojekten

In diesem Kapitel sollen aktuelle Forschungsansätze für verteilte Programmiersprachen vorgestellt und zu den in dieser Arbeit vorgestellten Konzepten und deren Umsetzung in Bezug gesetzt werden. Die Zahl der Forschungsprojekte mit dem Ziel einer Vereinfachung der Programmierung durch die Entwicklung von neuen Sprachen oder von Sprachdialekten ist verhältnismäßig groß. Einen Überblick über solche Projekte mit Fokus auf Aspekte der Verteilung findet man in [Briot et al. 1997] und [Vitek und Tschudin 1997]. Hier wurde daher notwendigerweise eine Auswahl getroffen, die an konzeptioneller Nähe und an der Möglichkeit der gegenseitigen Beeinflussung ausgerichtet war. Ein weitergehender Vergleich der in Dejay verwendeten Konzepte mit anderen Forschungsarbeiten kann in [Fragemann 2000] nachgelesen werden, der die nachfolgende Darstellung weitgehend folgt.

10.1 FarGo

FarGo [Holder et al. 1999] ist ein Projekt der Distributed Systems Group am Israel Institute of Technology in Haifa. Es wird seit 1996 unter der Leitung von Dr. Ben-Shaul entwickelt. FarGo basiert technisch gesehen, wie Dejay, auf Java und stellt, wie Dejay, eine Erweiterung der Sprache dar. Der Fokus dieses Projektes liegt aber vor allem auf der Migration und deren Koordination und nicht auf der Zusammenfassung von Verteilung, Nebenläufigkeit und Persistenz.

FarGo setzt, anstelle einer externen Middleware-Bibliothek wie Voyager, direkt auf die in Java integrierte RMI-API auf. Bei FarGo sind keine statischen Strukturierungsmechanismen wie der virtuelle Prozessor vorgesehen. Die Migration basiert direkt auf Objekten. Ein Objekt, das von sich aus beweglich sein soll, implementiert das Interface Complet. Es bildet den Anker (anchor) eines (Gesamt-)Complets. Das Complet besteht aus dem Anker sowie der transitiven Hülle aller durch lokale Java-Referenzen vom Anker aus direkt oder transitiv referenzierter Objekte. Wird das Anker-Objekt bewegt, folgt die gesamte Hülle. Innerhalb eines Complets werden Parameter by-Reference übergeben. Bei Aufrufen von Methoden eines anderen Complets wird entweder - bei "norma-

len" Objekten - by-Value übergeben, sprich eine Kopie angefertigt, oder - bei Übergabe eines Ankers - eine Referenz.

Die Ausführungseinheiten der Complets werden als Core bezeichnet, stationäre Java-Objekte, welche sich innerhalb einer JVM befinden, und sich um Complet-Referenzen, Complet-Bewegung, und um die Einhaltung von Gruppierungsvorschriften kümmern.

Im Rahmen dieser Arbeit ist insbesondere der Gruppierungsmechanismus von FarGo von Interesse. Gruppierung ist ein zentrales Anliegen von FarGo. Während vergleichbare Projekte mehr Wert auf Nebenläufigkeit, Verteilungstransparenz oder syntaktische Äquivalenz von lokalen und entfernten Aufrufen legen, liegt in FarGo der Fokus des wissenschaflichen Interesses auf der Gruppierung von Complets und Optimierung von Aufrufen. Das Standardbeispiel (das in allen Veröffentlichungen über FarGo genannt wird) ist ein ToDo-System, in dem Anwender, die sich über verschiedene Zeitzonen der Welt verteilen, auf einen gemeinsamen Terminplaner zugreifen. Je nachdem, auf welcher Seite der Erde gerade Tag ist (und dementsprechend viele Anwender ohne größere Latenzzeiten arbeiten möchten) sucht sich der Optimierer des Terminplaners einen geeigneten Ort, migriert dorthin, und zieht dabei den mit ihm verknüpften Rest des Systems mit sich.

Die Gruppierung erfolgt, indem innerhalb des Programmcodes eines Objektes a festgelegt wird, in welcher Beziehung a zu einem Objekt b steht. Die Verknüpfung nennt sich Relokator. Mit Relokatoren lassen sich durch transitives Verhalten auch größere Verbände bilden. Folgende Relokator-Typen stehen in FarGo zur Verfügung :

1. **Link Relocator:** Im eigentlichen Sinne kein Relokator: wenn b sich bewegt, bleibt die Referenz (Link) von a auf b erhalten, es können weiterhin (trotz Migration) Methodenaufrufe ausgeführt werden

2. **Pull Relocator:** b befindet sich immer am selben Ort wie a. Wenn sich a bewegt, folgt b an den neuen Ort. b kann nicht von sich aus migrieren. b kann zudem nicht einen weiteren Relokator c besitzen, da es - wenn sich a und c an unterschiedlichen Orten aufhalten sollten - nicht beiden folgen kann.

3. **Bidirectional Pull Relocator:** a und b befinden sich am selben Ort. a folgt b, wenn b migriert, und b folgt a.

4. **Stamp-Relocator:** Migriert a zu einem neuen Ort, versucht es dort, ein Objekt zu finden, das äquivalent zu b ist. Äquivalenz ist dabei Definitionssache - je nach Einstellung kann es sich z.B. um ein Objekt der gleichen Klasse handeln, oder um ein Objekt mit dem gleichen registrierten Namen.

5. **Duplication Relocator:** Wenn sich a bewegt, wird von b eine Kopie angefertigt. Diese Kopie migriert mit a zum neuen Ort.

Zusätzlich bietet FarGo bei Relokatoren weitere Tuning-Möglichkeiten an. Als erstes sei der oben erwähnte Optimierer genannt. Bei jeder Form von Kommunikation ist eine Verzögerung (delay) messbar. Wenn ein zentrales Complet mit

vielen anderen, gleichartigen kommuniziert, macht es Sinn, es so zu platzieren, dass die Summe der Verzögerung minimal ist. Hierfür kann ein LocationOptimizer angelegt werden, dem die zu überwachenden Cores, in denen sich die anderen Complets befinden können, genannt werden. Tritt während der Überwachung eine starke Veränderung der Verzögerungswerte auf, berechnet der Optimierer den günstigsten Ort im Netz, und bewegt sich sowie verknüpfte Complets dorthin.

Des weiteren können den einfachen Relokatoren Bedingungen mitgegeben werden. Beispielsweise kann einem Pull-Relokator als Parameter mitgeteilt werden, dass das Nachziehen des verknüpften Objekts erst nach einer Verzögerung von n Millisekunden erfolgen soll, oder erst dann, wenn das "ziehende" Objekt sich so weit vom "gezogenen" entfernt hat, dass das zwischen ihnen liegende Netzwerk einen Delay von mehr als n Millisekunden aufweist.

10.1.1 Vergleich

Der generelle Aufbau von Dejay und FarGo unterscheidet sich in vielen Details. Doch was die Migration angeht, sind sehr viele Parallelen vorhanden; UniPull und BiPull entsprechen sich weitgehend, da hier FarGo für Dejay Pate stand. Dejay besitzt im Gegenzug Eigenschaften, die FarGo fehlen:

- FarGo bietet eine Möglichkeit an, Objekte ortsfest zu machen. Dies erfolgt aber statisch über die Implementation eines Interfaces. In Dejay können virtuelle Prozessoren mittels setFixed(boolean b) dynamisch fixiert und wieder gelöst werden.

- Dejay bietet mit dem Rubberpull-Relocator einen schwachen Kompositionsmechanismus, mit dem sich einige Problemstellungen elegant lösen lassen und eine größere Flexibilität erlaubt.

In FarGo gibt es hingegen zwei Mechanismen, die Dejay fehlen.

- Der Stamp-Relocator ist ein relativ aufwändiger Mechanismus. Er sorgt dafür, dass ein migriertes Objekt bei Ankunft sich ein Objekt sucht, an das es sich anschließen kann - je nach Einstellung ein Objekt mit einem speziellen Namen oder einer Klasse. Dieser Mechanismus ist auf Objektebene sicherlich nützlich: Beispielsweise kann ein bewegliches Dokument sich auf diese Weise immer nach der Ankunft in einem neuen System einen stationären Drucker suchen. Auf Prozessorebene macht ein solcher Mechanismus aber wenig Sinn: Prozessoren sind Behälterobjekte, die keine Applikationsmechanismen beinhalten.

- Der Duplication Relocator bewirkt, dass bei Migration von Objekt B das Objekt A kopiert wird, und mit an den Zielort bewegt wird. Dies ist aber wie der Stamp-Relocator auf Prozessorebene kaum benutzbar. Man müsste hierfür zum Beispiel wissen, dass in dem zu duplizierenden Prozessor zu jedem Zeitpunkt ausschließlich zustandsfreie Objekte vorhanden sind, denn alles andere würde zu Inkonsistenz der Daten führen.

10.1.2 Symbiose

Durch Einführung von an FarGo angelehnte Gruppierungseigenschaften in Dejay ist ein Großteil der Symbiose bereits erfolgt. Weitere Möglichkeiten bietet eventuell der Stamp-Relokator. Wie beschrieben, macht dieser Relokator auf Prozessorebene keinen Sinn. Es wäre aber möglich, ihn auf Ebene der DjObjekte einzuführen. Wenn auch nicht so komfortabel wie in FarGo, bei dem man das Äquivalenz-Protokoll selber definieren kann, so wäre doch denkbar, dass sich Objekte zumindest auf Ebene von Namensgleichheit (im Namensdienst) nach erfolgter Migration einen geeigneten "Partner" suchen.

10.2 Pangaea

Unter der Leitung von Klaus-Peter Löhr sind am Fachbereich Informatik der Freien Universität Berlin mehrere aufeinander aufbauende Systeme zur Verteilten Programmierung entwickelt worden, von denen zwei in den folgenden beiden Abschnitten vorgestellt werden sollen. Zwar unterscheiden sie sich im Ansatz fundamental von einem System wie Dejay, könnten aber in Zukunft wichtige Impulse für Dejay liefern.

Im Projekt Pangaea [Spiegel 1999a] wird versucht, mittels sowohl statischer als auch dynamischer Analyse die Verteilung und Gruppierung von Objekten zu automatisieren. Pangaea basiert auf der Annahme, dass die geeignete Verteilung der Komponenten einerseits zu einem Großteil Routinearbeit darstellt, die dem Programmierer abgenommen werden kann. Anderseits könne vom Programmierer nicht ohne weiteres jeder Aspekt der Verteilung erkannt werden; manche Entscheidungen seien nur maschinell (gar zur Laufzeit) zu treffen, da der Aufwand einer genauen Analyse für einen Menschen unvertretbar sei.

Pangaea wird angewendet auf beliebige, zentralisierte Programme. Transparent und automatisch wird das Programm beim Kompilieren derart umgesetzt, dass es - je nach verwendetem Adapter- unterschiedliche Zielplattformen wie RMI oder CORBA unterstützt. Die Semantik des Programmes wird nicht angetastet. Ein sequentielles Client/Server-Programm wird auch nach der Verteilung sequentiell bleiben, ein threadbasiertes, nebenläufiges Programm wird nach der Verteilung ebenfalls nebenläufig (wenn auch nun parallel, auf mehrere Rechner verteilt) ausgeführt.

10.2.1 Statische Analyse

Unter dem Namen "Object Graph Analysis" [Spiegel 1999b] wird ein Algorithmus zur statischen Analyse eines Java-Programmes und seiner Laufzeitstruktur vorgestellt (im Folgenden als OGA bezeichnet.) Ziel des OGA ist, anhand der statischen Auswertung des Quelltextes das Laufzeitverhalten eines Java-Programms zu approximieren. Hierzu wird in mehreren aufeinander folgenden, in sich jeweils transitiven Schritten ein Objektgraph konstruiert, der Erzeugungs-, Referenz- und Benutzungsinformationen der Objekte untereinander darstellt. Die folgende Beschreibung stellt einen groben Überblick dar, der genaue Algorithmus kann in [Spiegel 1999b] nachgeschlagen werden.

- Im ersten Schritt werden die zu benutzenden Typen ermittelt. Hierbei wird nicht nur nach Java-Klassen sortiert, sondern innerhalb einer Klasse nochmal nach statischen und nicht-statischen Methoden/Variablen unterschieden.

- Aus den Typen wird im zweiten Schritt ein Typgraph erstellt, dieser zeigt die statischen Beziehungen und den Datenfluss der Klassen an.

- Anhand des Typgraphen wird ein Objektgraph erstellt, hierbei wird unterschieden zwischen statischen und dynamischen Objekten. Statische Objekte werden zur Laufzeit eines Programmes genau einmal generiert, dynamische dagegen 0..n Mal. Dynamische Objekte werden nochmals unterteilt in konkrete und indefinite Objekte. Bei konkreten Objekten ist garantiert, dass sie zur Laufzeit erzeugt werden. Beispielsweise dadurch, dass sie in der Methode main() der Start-Klasse, oder im Konstruktor einer Klasse, die von main() aus instantiiert wird, vorkommen. Weiterhin darf die Erzeugung eines Objektes nicht durch eine Kontrollstruktur wie "if" oder "for" in Frage gestellt worden sein. Objekte, bei denen man sich nicht sicher sein kann, ob sie wirklich erzeugt werden können, heißen indefinit. Ein indefinites Objekt kann seinerseits nur indefinite Objekte erzeugen, selbst wenn im Konstruktor des indefiniten Objektes ein weiteres Objekt "ohne wenn und aber" instantiiert wird. Durch das rekursive Parsen des Quelltextes ist nach dem dritten Schritt bekannt, welche Objekte welche erzeugen können.

- Im vierten Schritt wird anhand des Typgraphen ermittelt, wie Objekte durch Methodenaufrufe und Parameterübergabe aufeinander Referenzen erhalten. Diese Information wird in Form von Referenz-Kanten im Objektgraphen eingefügt.

- Der letzte Schritt fügt dem Objektgraphen die "Benutzt"-Beziehung hinzu. Ein Objekt a benutzt b, wenn es Methoden von b aufruft.

Durch die Unterscheidung von konkreten und indefiniten Objekten ist OGA weit mehr als eine reine Typanalyse. Bei einer Typanalyse könnte man zum Schluss kommen, dass Objekte von Typ A Objekte von Typ B aufrufen, und sich deshalb immer gemeinsam in der selben Laufzeitumgebung aufhalten sollten. Nur mit einer Laufzeit-Approximation wie OGA kann erkannt werden, dass ein Objekt a' in einem Kontext regelmäßig (und durch konkrete Objekte auch garantiert) auf Objekte von Typ B zugreift, und deshalb mit b' in der selben Laufzeitumgebung sein sollte. In einem anderen Kontext dagegen könnte es sein, dass a" nur sporadisch auf b" zugreift, und daher eine Gruppierung nicht sinnvoll ist.

Aufgrund von Testreihen sind die Entwickler zum Ergebnis gekommen, dass der Algorithmus, obwohl prinzipiell exponentieller Natur, in der Praxis eine polynomielle Komplexität besitzt. Programme mit mehr als 10.000 Zeilen Code wurden binnen Minuten analysiert.

10.2.2 Dynamische Analyse

Nachdem der Objektgraph ermittelt wurde, wird er auf bestimmte Vorkommnisse untersucht. Beispielsweise werden Objekte, auf die nicht oder nur während der Initialisierung schreibend zugegriffen wird, als immutable objects erfasst, und bei Migration oder Parameterübergabe als Kopie übermittelt. Dies erspart danach teure Fernaufrufe. Andere Objekte kommen in gewissen Kontexten nur als private Variablen vor, und müssen ebenfalls nicht fernaufrufbar sein. Methodenaufrufe, die innerhalb von Schleifen ausgeführt werden, signalisieren, dass das rufende und das gerufene Objekte nicht voneinander getrennt werden sollten, da hier - zumal bei unbekannter Schleifenlänge - entfernte Aufrufe leicht in die Hunderte gehen und das Programm drastisch verlangsamen könnten.

Da eine inhaltliche Bewertung (welche Programmteile gehören zum Server, welche zum Client?) unmöglich ist, wird die Analysephase vom Entwickler angestoßen. Dies geschieht beispielsweise, indem Elemente der GUI als clientseitig und Datenbankzugriffs-Objekte als serverseitig deklariert werden. Basierend auf dieser Initial-Information versucht Pangaea anhand der eben genannten Kriterien, eine Trennlinie zwischen den benutzten Objekten zu ziehen.

Zusätzlich zur statischen Analyse wird auch während der Laufzeit ermittelt, welche Objekte sehr häufig miteinander kommunizieren. Diese Objekte werden - wenn möglich - zusammengefasst. Zwar ist das Laufzeit-Verfahren aufwändiger, und greift erst, nachdem schon Geschwindigkeitseinbußen aufgetreten sind, andererseits lässt die statische Analyse nur Annäherungen an das Laufzeitverhalten zu, keine exakten Vorhersagen. Die Laufzeitanalyse unterstützt daher den Programmierer gerade während der Entwicklungsphase - in dieser Phase sind Geschwindigkeitseinbußen kaum relevant. Bis zum Einsatz des Programmes unter realen Bedingungen sollten alle Geschwindigkeitsoptimierungen vollzogen sein, so dass die Laufzeitanalyse, welche selber auch Rechenzeit erfordert, dann deaktiviert werden kann.

10.2.3 Gruppierung

Gruppierung erfolgt, wie erwähnt, zunächst durch Festlegung durch den Entwickler, dann durch statische Ermittlung der Abhängigkeiten der Programmteile, und als letztes durch Auswertung des Laufzeitverhaltens. Es ist also denkbar, dass die Gruppierung während der Ausführung je nach Kontext umgestoßen und neu formiert wird.

Gruppierung auf Programmebene ist nicht vorgesehen - dem Entwickler sind keine Möglichkeiten gegeben, Objekte zusammenzufassen, wie er es für sinnvoll hält. Dadurch unterscheidet sich Pangaea grundlegend von FarGo und Dejay.

10.2.4 Symbiose

Pangaea kann Dejay in zwei Hinsichten um wichtige Elemente bereichern:

- Pangaea kann dem Programmierer viel Routinearbeit abnehmen, zumal wenn es sich um bereits fertig entwickelte Programme handelt. Es wä-

re eine Adapterklasse denkbar, die aus normalem Programmcode Dejay-Programme generiert.

- Die Analysemethoden von Pangaea wären ideal, um den Entwickler vor Fehleinschätzungen bei der Gruppierung zu warnen.

Ein Problem bei Dejay ist, dass der Programmierer Zusammenhänge falsch einschätzen und es durch ungeschickte Gruppierung zu Performance-Einbrüchen kommen kann. Programmierung ohne umfangreiche Testläufe ist nicht sinnvoll. Dafür kann aber ein fundiertes Hintergrundwissen über die Applikation zu optimalen Ergebnissen führen.

Ein Probleme bei Pangaea ist, dass Moduswechsel nicht bemerkt werden können. Zwar wird der fortgeschritte OGA-Algorithmus angewandt, aber wenn zwei Objekte A und B viele gegenseitige Aufrufe (gar aus Schleifen heraus) aufeinander haben, schließt Pangaea, dass sie zusammengehören. Die Aufrufe könnten jedoch von gewissen (selten erfüllten) Bedingungen abhängen, während andere Aufrufe viel öfter stattfinden. Selbst wenn der Programmierer dieses Hintergrundwissen besitzt, kann er es in Pangaea nicht modellieren. Zudem kann in einem Dejay-Programm schon bevor eine Aufgabe gelöst werden soll eine Umgruppierung der Prozessoren stattfinden - in Pangaea müssten erst Laufzeitanalysen erfolgen und eine schlechte Performance registrieren, bevor eine Umgruppierung erfolgen kann.

Eine ideale Symbiose von Pangaea und OGA mit Dejay wäre, dass der Programmierer während der Entwicklung Gruppen bestimmt, die zusammengehören. Pangaea gibt dabei durch statische Analyse wichtige Hinweise. Zur Laufzeit werden durch Pangaea weitere Probleme und Performanceprobleme aufgedeckt, diese werden durch modusabhängige Verknüpfungen und Trennungen mittels Dejay-Gruppen optimiert, bis sie bei den folgenden Läufen des Programms vom Optimierer nicht mehr bemängelt werden.

10.3 Doorastha

Doorastha [Dahm 2000] ist ein Tag-basiertes System, mit dem vorhandener, zentralisierter (eventuell nebenläufiger) Code ohne großen Aufwand verteilt werden kann. Anders als bei vielen ähnlichen Systemen ist ein treibendes Motiv hinter Doorastha, dass Code ohne Änderung des Quelltextes benutzt werden können soll. Weder muss von gewissen Klassen geerbt werden, um migrationsfähige Objekte zu erhalten, noch muss der Quelltext von benutzten Programmpaketen/Utilities vorliegen. Ein Programm, das nebenläufig - beispielsweise mit Threads - geschrieben wurde, soll allein durch den Einsatz von Tags verteilt werden können. Die Tags werden innerhalb von Kommentaren eingefügt, dadurch bleibt das Programm auch mit gängigen Compilern und Interpretern ausführbar.

In Doorastha werden zwei Arten der Parameterübergabe unterschieden. **"Pass by copy"** steht für eine Parameterübergabe, bei der eine Kopie eines Objektes übergeben wird. Dies schließt von shallow-copy über beliebig viele Zwischenstufen bis deep-copy alles ein. **"Pass by reference-value"** ("by-refvalue") ist eine Präzisierung des gebräuchlichen Terms "by reference": "by reference" sei keine

ausreichende Beschreibung, da die Referenz (anders als z.B. bei einem Pointer in C) selber nicht bearbeitet werden kann, nur das referenzierte Objekt.

Die zur Verfügung stehenden Tags werden in spitzen Klammern und innerhalb von Kommentarzeichen angegeben. Hier eine Zusammenstellung der wichtigsten Tags, die in Doorastha zur Formulierung von Verteilung zur Verfügung stehenden:

- Die wichtigsten Tags auf Klassenebene:
  - <**copyable**> Damit ein Objekt by-copy übergeben werden kann, muss es kopierbar sein. Basistypen wie int und serialisierbare Objekte und sind automatisch kopierbar. Alle anderen können durch das Tag <copyable> in der Klassendefinition zu kopierbaren Objekten gemacht werden.
  - <**globalizable**> Soll ein Objekt dagegen by-refvalue übergeben werden können, z.B. um dieses Objekt auf einem entfernten Rechner bekannt zu machen (durch by-refvalue - Parameter), muss die Klasse des Objekts mit dem Tag <globalizable> versehen werden.
  - <**migratable**> Nur wenn die Klasse dieses Tag vorweist, kann ein Objekt der Klasse migrieren
- Nutzung von Tags auf Methodenebene: Parameter- und Rückgabewerte
  - <**by-value**> Der Parameter oder Rückgabewert wird als Kopie übergeben.
  - <**by-refvalue**> Der Parameter oder Rückgabewert wird als Referenz übergeben.
  - Wenn kein Tag angegeben wird, hängt die Art der Parameterübergabe von der Klasse des formalen Parameters ab: Ist die Klasse <copyable>, geschieht die Übergabe per default <by-copy>, ansonsten <by-refvalue>. Es wird eine Warnung ausgegeben.

Eine Klasse kann sowohl <copyable> als auch <globalizable> sein. Es hängt von den Tags auf Methodenebene ab, wie die Übergabe erfolgt.

- Wichtigste Tags auf Instanzvariablenebene: Alle Instanzvariablen einer Klasse können mit Tags bezeichnet werden, die bestimmen, wie sie sich während eines Kopiervorgangs des Objektes verhalten.
  - <**by-value**> Die Instanzvariable wird kopiert
  - <**by-refvalue**> Die Instanzvariable wird als Referenz übergeben
  - <**null**> Nach einer Übergabe wird die Variable den Wert null haben. z.B: <null>File file;
  - <**rebind**> Nach einer Übergabe wird die Variable neu instanziiert, z.B: "<rebind: expr= new Hashtable()> Hashtable hashtable;"

Nur wenn alle Instanzvariablen einer Klasse entweder mit <by-value> markiert sind, aus Basistypen wie int bestehen, oder serializable sind, wird eine deep-copy erzeugt. Wenn alle Instanzvariablen <by-refvalue> markiert wurden, spricht man von einer shallow-copy. Dazwischen sind alle möglichen Zwischenstufen möglich.

10.3.1 Migration

Die einfachste Methode zur Verteilung ist die entfernte Erzeugung von Objekten, wie bei diesem anonymen Thread:

```
Thread t= new <remotenew :host="nidan"><globalizable>Thread () { [...] };
t.start();
```

Doch entfernte Erzeugung ersetzt keine Migration. Migration erfolgt in Doorastha nicht durch einen spezifischen move()-Befehl, sondern durch die Angabe des Tags <by-move> in einem Methodenaufruf. Während bei <by-copy> eine Kopie des Parameter-Objekts an das gerufene Objekt übergeben wird, führt das Tag <by-move> dazu, dass das Parameter-Objekt komplett übergeben wird, und am Aufenthaltsort des gerufenen Objektes verbleibt.

10.3.2 Beispiel zur Verwendung von Tags:

Eine einfach verkettete Liste, die Datenobjekte enthält, soll die eben genannten Tags erklären

```
class <copyable> List
{
   private Node start, end;
   public void append(DataObject data)
   {
      if (start==null)
         start=end=new Node(data);
      else
      {
         end.next= new Node(data);
         end=end.next;
       }
}
class <globalizable> DataObject
{
    public DataObject(<by-refvalue>Object o) { ... };
    public <by-refvalue> Object getContent() { ... }
}
class <copyable> Node
{
   private <by-refvalue> DataObject dataObject;
   private Node <by-copy> nextNode;
```

```
    public Node (DataObject data) { dataObject=data; }
}
```

Ohne die eingefügten Tags handelt es sich um eine einfach verkettete Liste, die Nodes enthält. Jeder Node wiederum enthält ein DataObject, welches ein beliebiges Objekt beherbergt. Eine Möglichkeit, diese Liste für die Verteilung zu präparieren, wäre das Tag <copyable> in der Definition der Liste einzuführen. Nun könnte eine Klientenklasse auf einem entfernten Rechner sich eine Kopie der Liste anfertigen, die Liste durchlaufen, und Werte lesen/ändern. Doch da es sich um eine Kopie handelt, würden sich Änderungen nicht am Original bemerkbar machen, außerdem würden alle DataObject-Objekte kopiert werden - eventuell ein große Datenmenge.

Um die Liste komplett als Referenz zu übergeben, würde ein Tag <globalizable> genügen: Alle Aufrufe wären entfernter Art, die Daten-Integrität bleibt gewahrt. Doch der Nachteil daran wäre, dass viele entfernte Aufrufe zu bewältigen wären.

Insofern bietet der obige Ansatz eine elegante Lösung. Die Liste selber ist kopierbar. Auch die enthaltenen Nodes sind kopierbar, und werden bei der Übergabe der Liste repliziert. Einzig die Data-Objekte sowie ihre enthaltenen Objekte sind Referenzen. Eine Kopie der Liste kann z.B. bis zum 100. Element durchlaufen werden, ohne einen einzigen Fernaufruf auszuführen. Erst der lesende oder schreibende Zugriff erfordert einen Fernaufruf.

Nach Angaben der Entwickler von Doorastha wurden diverse, nicht-triviale Programme an Doorastha angepasst und erfolgreich ausgeführt. Ein einfacher Raytracer von 30 Klassen und circa 1.000 Programmzeilen wurde mittels nur 6 Tags verteilt.

10.3.3 Symbiose

Leider können nur sehr wenige Ansätze von Doorastha in Dejay übernommen werden. Dejay ist auf Entwickler zugeschnitten, die genau wissen, wie sie einen bestimmten Algorithmus implementieren möchten - je nach Wunsch können unterschiedliche Aufrufmethoden (synchron vs. asynchron), Verteilungsstrategien und Gruppierungsmöglichkeiten benutzt werden. Das Layout von Dejay-Programmen wird von Grund auf neu entwickelt - nur bestehende Datenstrukturen werden übernommen. Die feinkörnige Programmierung steht der - in sich sehr sinnvollen - Nutzung von vorhandenen (eventuell nicht einmal im Sourcecode verfügbaren) Programmteilen diametral entgegen.

Einzig die Aufrufform Call-by-Move wäre mit akzeptablem Aufwand adaptierbar. Doch ob man einen Methodenaufruf als "server.calculate(<by move> a);" oder als "djA.moveTo(djServer); djServer.calculate(djA);" formuliert, macht keinen besonderen Unterschied.

Die meisten anderen Tags haben in Dejay bereits ihre Entsprechung: Beispielsweise sind alle Klassen, die in einer *.dj-Datei implementiert sind, automatisch <globalizable>, und die entfernte Erzeugung von Objekten wie mit "Thread t= new **<remotenew :host="sun">** **<globalizable>** Thread () { [...] };" hat sein Analogon in "DjA djA= new DjA(new DjProcessor("tcp://sun:8000"));".

Dieses Projekt zeigt aber, dass die konkrete Syntax sehr unterschiedlich ausfallen kann, selbst wenn ähnliche Konzepte der, Aspekte der Verteilung umgesetzt werden.

Teil III

Evaluation

Kapitel 11

Evaluation der Aspekte Verteilung, Nebenläufigkeit und Persistenz

In diesem Kapitel soll der Nachweis erbracht werden, dass das vorgestellte und in Dejay umgesetzten Konzept der virtuellen Prozessoren für die Behandlung der Aspekte Nebenläufigkeit, Verteilung und Persistenz geeignet ist. Daher wird zunächst jeder einzelne dieser Aspekte isoliert behandelt und an drei konkreten Beispielen die Umsetzung mit jeweils dem selben Konstrukt demonstriert. Alle Beispiele tragen Charakteristika dieser drei Aspekte in sich, doch in jedem Beispiel wird einer dieser Aspekte besonders hervorgehoben.

Im ersten Beispiel wird ein Szenario aufgezeigt, in dem es um das Verhandeln und Unterzeichnen eines Vertrages geht. Die beteiligten Verhandlungspartner sind räumlich voneinander getrennt, so dass der Gegenstand der Verhandlung, der Vertrag, mal entfernt zugegriffen und mal migriert werden muss. Hier wird also insbesondere der Aspekt der Verteilung und der Migration gezeigt.

Das zweite Beispiel untersucht den Aspekt der Nebenläufigkeit. Dazu wird ein rechenintensiver Algorithmus herangezogen, der sich für eine nebenläufige Berechnung eignet und die Rechenlast auf verschiedene Prozessoren verteilt werden kann. Das konkrete Beispiel ist die Berechnung eines Fraktals, des so genannten Apfelmännchens.

Im dritten Beispiel sollen die Vorteile von Dejay für den Aspekt der Persistenz gezeigt werden. Hierfür wird ein persönlicher Datenassistent verwendet, der Termine, Adressen und eine Aufgabenliste verwalten kann. Auf diesen kann entweder über ein Netz entfernt zugegriffen werden, oder er kann migriert werden. Wenn er gerade nicht benötigt wird, kann er auf sehr einfache Weise abgespeichert (persistiert) und bei Bedarf wieder aktiviert werden, worauf der Schwerpunkt dieses Beispiels liegt.

11.1 Verteilung

Man stelle sich ein Szenario vor, in dem mehrere Beteiligte sich über ein bestimmtes Vorhaben einigen wollen und dafür einen Vertrag erstellen wollen. Die

Beteiligten, die hier als Vertragspartner bezeichnen werden, können oder wollen sich nicht persönlich treffen, weil sie zum Beispiel weit voneinander entfernt wohnen. Oder sie haben dies schon getan und wollen die Details nun von ihrem Büro aus erledigen. Dafür benutzen sie ein Netz, das entweder das Internet, ein Intra- oder ein Extranet sein kann, je nach Anforderungen und gegebenen Bedingungen. Als Koordinator, der den Verhandlungsverlauf steuert und überprüft, bestimmen sie einen Notar. Dieser wird (wie alle Beteiligten) in diesem Beispiel durch ein Programm simuliert, könnte aber auch in der Realität vollautomatisch operieren.

Der Notar soll den Ablauf der Verhandlung und des Vertragsabschlusses steuern, er ist sozusagen der Hauptakteur. In Ermangelung einer realen oder auch nur virtuell vorhandenen Welt muss der Notar zunächst die Vertragspartner ebenso wie den Vertrag neu erzeugen, die jeweils durch ein Objekt repräsentiert werden. Er ordnet die Partner diesem Vertrag zu und legt jedem den Vertrag zur Begutachtung, zur Korrektur und schließlich zur Unterschrift vor. Um Inkonsistenzen zu vermeiden, soll es dabei nur ein Exemplar des Vertrages geben. Damit aber der Zugriff auf das Vertragsobjekt bei der Bearbeitung schnell ist, wird der Vertrag jeweils zu dem Partner bewegt, der es einsehen, korrigieren oder unterschreiben darf. Zu guter Letzt soll der Notar überprüfen, ob alle Vertragspartner unterschrieben haben.

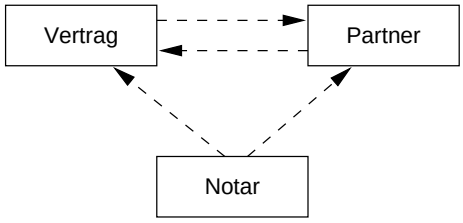
Der Vertragspartner hat einen Namen und eine »Heimatadresse«, dargestellt durch eine URL. Auf weitere Informationen wird der Einfachheit halber verzichtet. Auch das Kontrollieren und Korrigieren eines Vertrages wird deshalb hier nicht demonstriert, sondern das Augenmerk wird vornehmlich auf das Unterschreiben gerichtet. Der Partner hat eine wesentliche Methode, nämlich für das Unterschreiben. Damit kann der Notar ihm den Vertrag zur Unterschrift vorlegen. Der Partner will den Vertrag nur dann unterschreiben, wenn er sich am gleichen Ort (an der gleichen URL) befindet wie er selbst. Als Grund dafür könnte man annehmen, dass beim Unterschreiben ein Passwort oder ein Schlüssel übergeben wird, und der Partner wird berechtigtes Interesse haben, diese vertraulichen Daten nicht über das Netzwerk zu befördern. Wichtig ist, dass der Partner, bevor er den Vertrag unterschreibt, überprüft, ob er am gleichen Ort ist. Wenn das nicht der Fall ist, sagt er dem Notar (durch den Rückgabewert false), dass er den Vertrag nicht unterschreiben will. Der Notar verschiebt dann den Vertrag zum Partner und ruft die Methode erneut auf. Jetzt wird der Partner unterschreiben.

Der Vertrag bildet das zentrale Element des Beispiels, an dem der entfernte Zugriff und die Migration gezeigt werden. Er wird vom Notar erzeugt, der über eine entfernte Referenz jeder Zeit auf den Vertrag zugreifen und ihn verschieben kann. Auch die beteiligten Partner sind vom Vertrag aus über eine entfernte Referenz erreichbar.

Der Vertrag

Nun sollen die einzelnen Klassen im Detail vorgestellt werden. Der Inhalt des Vertrages ist ohne weitere Bedeutung und wird daher einfach durch einen String Bezeichner repräsentiert. Einziges weiteres Feld ist der Vektor partnerListe, in

Abbildung 11.1: Die Klassen des Vertrags-Szenarios



dem die Partner des Vertrages abgelegt werden. Die Partner werden vom Notar mit der Methode neuerPartner() in den Vertrag eingefügt. Die Methode ichUnterschreibe() wird von den Partnern aufgerufen, wenn sie den Vertrag unterschreiben wollen. Diese Methode erzeugt natürlich nicht eine Signatur im Sinne der Kryptographie (das Passwort wird gar nicht benutzt, der Vertragspartner wird anhand seines Namens identifiziert...), aber für Demonstrationszwecke reicht sie hier aus.

Mit der Methode habenAlleUnterschrieben() kann geprüft werden, ob alle Partner dem Vertrag zugestimmt haben. Dazu wird die Aufzählung (Enumeration) aller in der Partnerliste vertretenen Partner durchgegangen und bei jedem das Feld hatUnterschrieben überprüft. Die Schleife bricht ab, wenn ein Partner nicht unterschrieben hat und gibt false als Rückgabewert zurück. Haben jedoch alle Partner unterschrieben, läuft die Schleife ganz durch und es wird true zurückgegeben.

Das wichtige an dieser Methode ist, dass in der Partnerliste ja nicht vollständige Partner abgespeichert sind, sondern eine entfernte Referenz auf diese Partner, DjPartner. Das heißt, dass hier auf die entfernten Partner zugegriffen wird, um den aktuellen Wert von hatUnterschrieben zu ermitteln.

Zu guter Letzt gibt es noch eine Methode meldeDich(), um den Vertrag eine Meldung ausgeben zu lassen, damit man während des Programmlaufs sieht, wo er sich befindet.

```
public class Vertrag {

  public Vector partnerListe;
  public String bezeichner;

  public Vertrag(String bezeichner) {
    this.bezeichner = bezeichner;
    partnerListe = new Vector()
  }

  public void neuerPartner(DjPartner neuerPartner) {
    System.out.println("neuer Partner "+neuerPartner);
    partnerListe.addElement(neuerPartner);
  }

  public void ichUnterschreibe(String unterschreibender, String password) {
```

```
    Partner naechsterPartner;
    for(Enumeration e = partnerListe.elements(); e.hasMoreElements();) {
      naechsterPartner = (DjPartner)(e.nextElement());
      if(naechsterPartner.gibName().equals(unterschreibender)) {
        System.out.println(naechsterPartner.gibName() + " unterschreibt.");
        naechsterPartner.setzeHatUnterschrieben(true);
      }
    }
  }

  public boolean habenAlleUnterschrieben() {
    DjPartner partner;
    System.out.println("Haben alle "+ partnerListe.size() + " unterschrieben?");
    for(Enumeration e = partnerListe.elements(); e.hasMoreElements();) {
      partner = (DjPartner)(e.nextElement());
      System.out.println("Überprüfe "+partner.gibName());
      if(!partner.gibHatUnterschrieben()) {
        System.out.println(partner.gibName() + " hat nicht unterschrieben.");
        return false;
      } else {
        System.out.println(partner.gibName() + " hat unterschrieben.");
      }
    }
    return true;
  }

  public void meldeDich() {
    System.out.println("---------- Der Vertrag "+bezeichner+" ist hier!");
  }
}
```

Der Partner

Der Partner hat nur drei Felder. Das Feld hatUnterschrieben wird standardmäßig auf false gesetzt. Name und heimatAdresse werden vom Konstruktor gesetzt. Zu jedem dieser Felder gibt es eine Methode, um es auszulesen. Dies ist insbesondere für den entfernten Zugriff wichtig, denn entfernt kann nur über Methoden zugegriffen werden, nicht auf die Felder selbst. Die wichtige Methode dieser Klasse ist unterschreibe(). Hier soll ein als Parameter übergebener Vertrag unterschrieben werden. Der Vertrag wird als entfernte Referenz übergeben, so dass der Vertrag zwar zugreifbar, aber nicht unbedingt lokal ist. Daher wird die Adresse des Vertrages mittels vertrag.getRemoteAddress() herausgefunden. Nur wenn diese Adresse mit der Heimatadresse übereinstimmt, wird der Vertrag unterschrieben. Anhand des Rückgabewerts kann der Notar erkennen, ob der Vertrag unterschrieben worden ist oder nicht und gegebenenfalls den Vertrag zum Partner verschieben, um den Partner nochmals aufzufordern, den Vertrag zu unterschreiben. Wie beim Vertrag gibt es eine Methode meldeDich(), um ei-

ne Nachricht auf den Bildschirm auszugeben, damit man sehen kann, wo der Partner ist.

```
public class Partner {

  public String name;
  public String heimatAdresse;
  public boolean hatUnterschrieben;

  public Partner(String name, String heimatAdresse) {
    this.name = name;
    this.heimatAdresse = heimatAdresse;
    this.hatUnterschrieben = false;
    System.out.println("Ich bin " + name);
  }

  public String gibHeimatAdresse() {
    return heimatAdresse;
  }

  public String gibName() {
    return name;
  }

  public void setzeHatUnterschrieben(boolean hatUnterschrieben) {
    this.hatUnterschrieben = hatUnterschrieben;
  }

  public boolean gibHatUnterschrieben () {
    return hatUnterschrieben;
  }

  public boolean unterschreibe(DjVertrag vertrag) {
    String vertragsAdresse = vertrag.getRemoteAddress();

    if(vertragsAdresse.equals(heimatAdresse)) {
      // hier kommt der Abschnitt, wo "vertrauliche" Daten übergeben werden
      vertrag.ichUnterschreibe(name, "meinPaßwort");
      System.out.println("Ich unterschreibe den Vertrag "+vertrag);
      return true;
    } else {
      System.out.println("Vertrag ist nicht hier, ich unterschreibe ihn nicht!");
      return false;
    }
  }

  public void meldeDich() {
```

```
    System.out.println("---------- Hallo, ich bin "+name+", ich bin hier!");
  }
}
```

Der Notar

Der Notar ist die zentrale Schaltstelle dieses Beispiels. Er ist nötig, um die Funktionalität der einzelnen Klassen und Methoden zu testen. Da es in diesem Beispiel hauptsächlich um die entfernte Erzeugung und die Mobilität von Objekten geht, in diesem Fall um die Mobilität des Vertrags, sei das Hauptaugenmerk darauf gerichtet und der übrige Code nur beiläufig kommentiert.

Zunächst werden drei DjProcessoren erzeugt. Auf einem wird der Vertrag gestartet, auf den anderen beiden werden Partner erzeugt, die dann dem Vertrag hinzugefügt werden. Jetzt wollen wir, dass die Partner den Vertrag unterschreiben. Die Methode unterschreibe(vertrag) wird für jeden Partner zweimal aufgerufen, wobei der erste Aufruf fehlschlägt, da der Vertrag nicht auf dem gleichen Rechner ist. Dann wird der Vertrag verschoben (oder eigentlich: der DjProcessor, in dem der Vertrag sich befindet) und die Methode erneut aufgerufen. Jetzt wird der Partner ihn unterschreiben. Der Vertrag soll sich nach dem Verschieben einmal melden, damit deutlich wird, wo er sich befindet (siehe Programmausgaben). Schließlich kann überprüft werden, ob alle Vertragspartner unterschrieben haben.

```
public class Notar {

  public static void main(String args[]) {
    DjVertrag vertrag;
    DjPartner partner1;
    DjPartner partner2;

    String adresse0 = "tcp://sun:8000";
    String adresse1 = "tcp://lin:8000";
    String adresse2 = "tcp://win:8000";

    DjProcessor proc0 = null;
    DjProcessor proc1 = null;
    DjProcessor proc2 = null;

    try {
      proc0 = new DjProcessor(adresse0);
      proc1 = new DjProcessor(adresse1);
      proc2 = new DjProcessor(adresse2);
    } catch (ConstructProcessorFailedException e) {
      System.out.println("kein Prozessor!"+e);
    }
    // den Vertrag erstellen
    vertrag = new DjVertrag("MS-Lizenzvertrag", proc0);
```

```
    // die Vertragspartner erstellen
    partner1 = new DjPartner("Toby",adresse1,proc1);
    partner2 = new DjPartner("Jan",adresse2,proc2);

    System.out.println("Partner werden hinzugefügt.");
    vertrag.neuerPartner(partner1);
    vertrag.neuerPartner(partner2);
    if (!(partner1.unterschreibe(vertrag))) {
      vertrag.moveTo(partner1);
      vertrag.meldeDich();
      partner1.unterschreibe(vertrag);
    }

    //überprüfen, ob unterschrieben wurde
    System.out.println(partner1.gibHatUnterschrieben());

    //Dasselbe für partner2
    if (!(partner2.unterschreibe(vertrag))) {
      vertrag.moveTo(partner2);
      vertrag.meldeDich();
      partner2.unterschreibe(vertrag);
    }

    System.out.println(partner2.gibHatUnterschrieben());

    vertrag.moveTo(adresse0);

    if (vertrag.habenAlleUnterschrieben()) {
      System.out.println("OK!!!");
    } else {
      System.out.println("Nicht OK!!!");
    }
    System.exit(0);
  }
}
```

Programmausgaben

Dieses Programm kann nun mit dem Dejay-Compiler dejayc übersetzt werden. Der erzeugte Code muss letztendlich jedem beteiligten Rechner zur Verfügung stehen oder über einen Code-Server geladen werden können, wie dies im letzten Kapitel beschrieben wurde.

```
sun> dejayc Notar.dj Vertrag.dj Partner.dj
```

Um das Programm ablaufen lassen zu können, muss zunächst die zugrundeliegende Infrastruktur, die *virtual backplane*, aufgebaut werden, wozu auf jedem

beteiligten Rechner jeweils ein Voyager-Daemon gestartet wird. Die verwendeten Rechnern sind hier sun, lin und win.

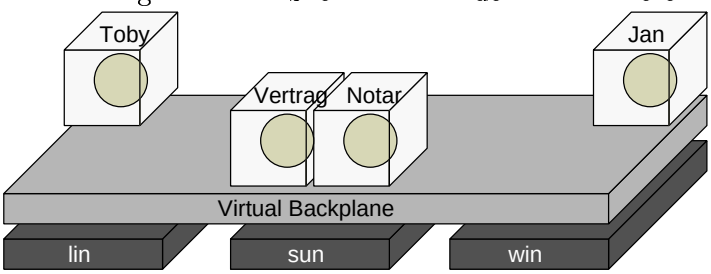
```
lin> voyager 8000

win> voyager 8000

nt > voyager 8000
```

Beim Notar wird der Voyager-Daemon automatisch gestartet, da der Notar das Hauptprogramm enthält und der Dejay-Compiler Hauptprogramme auf diese Weise vorbereitet. Der Notar wird auf dem Rechner sun gestartet. Die nachfolgenden Ausgaben sind zeitlich sortiert, aber man achte darauf, dass sie auf unterschiedlichen Rechnern erscheinen.

```
sun> java Notar
sun| Address - main : tcp://sun.informatik.uni-hamburg.de:7042

lin| Ich bin Toby

win| Ich bin Jan

sun| Partner werden hinzugefügt.
sun| neuer Partner Partner@80d3460
sun| neuer Partner Partner@80d25f3
```

Auf dem Rechner sun sind nun ein Notar und ein Vertrag, jeweils in einem eigenen virtuellen Prozessor, auf dem Rechner lin und win jeweils ein Partner in einem virtuellen Prozessor erzeugt worden, Abbildung 11.2 zeigt dieses Szenario. Der Vertrag enthält entfernte Referenzen auf die beiden Partner (in der Abbildung nicht dargestellt).

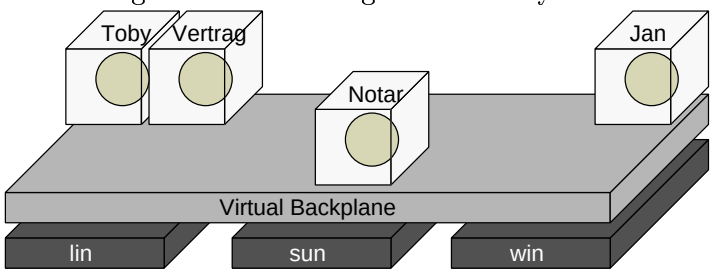
Abbildung 11.2: Das Szenario nach dem Initialisieren



Toby wird aufgefordert, den Vertrag zu unterschreiben, doch da sich der Vertrag nicht lokal bei ihm befindet, lehnt Toby dies ab. Der Notar verschiebt daraufhin den Vertrag. Die Ausgabe hierzu erscheint an der Konsole des Notars. Dann ist der Vertrag bei Toby und er unterschreibt. Schließlich überprüft der Notar, ob Toby auch wirklich unterschrieben hat, was zu einer Ausgabe beim Notar führt. Die jetzige Situation ist in Abbildung 11.3 festgehalten.

```
lin| Vertrag ist nicht hier, ich unterschreibe ihn nicht!

sun| Bisherige Adresse ist tcp://sun.informatik.uni-hamburg.de:8080
sun| Neue Adresse ist tcp://lin.informatik.uni-hamburg.de:8000

lin| ---------- Der Vertrag MS-Lizenzvertrag ist hier!
lin| Toby unterschreibt.
lin| Ich unterschreibe den Vertrag Vertrag@80d2264

sun| true
```

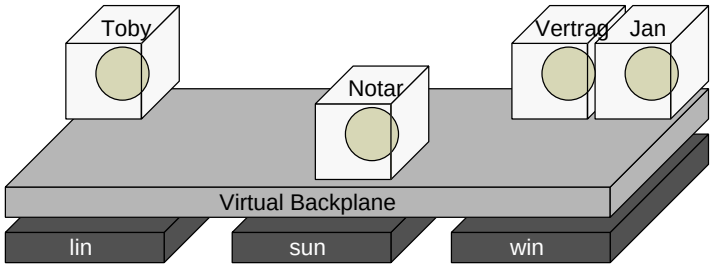
Abbildung 11.3: Der Vertrag wird zu Toby verschoben



Dasselbe passiert noch einmal mit Jan.

```
win| Vertrag ist nicht hier, ich unterschreibe ihn nicht!

sun| Bisherige Adresse ist tcp://lin.informatik.uni-hamburg.de:8000
sun| Neue Adresse ist tcp://win.informatik.uni-hamburg.de:9000

win| ---------- Der Vertrag MS-Lizenzvertrag ist hier!
win| Jan unterschreibt.
win| Ich unterschreibe den Vertrag Vertrag@80d250d

sun| true
```

Abbildung 11.4: Der Vertrag wird zu Jan verschoben



Anschließend holt der Notar den Vertrag wieder zu sich und überprüft, ob alle Partner unterschrieben haben, und stellt fest, dass alles OK ist. Diese Überprüfung verursacht einen entfernten Zugriff auf die jeweiligen Partner.

```
sun| Bisherige Adresse ist tcp://win.informatik.uni-hamburg.de:9000
sun| Neue Adresse ist tcp://sun.informatik.uni-hamburg.de:8080
sun| Haben alle 2 unterschrieben?
sun| Überprüfe Toby
sun| Toby hat unterschrieben.
sun| Überprüfe Jan
sun| Jan hat unterschrieben.
sun| OK!!!
```

Dieses Beispiel verdeutlicht, dass die Migration von Objekten in Dejay sehr einfach ist und entfernte Referenzen dabei erhalten bleiben. Der Vertrag hat sowohl eigene, lokale Objekte (wie die partnerListe als Vektor oder den Bezeichner als String-Objekt) als auch Referenzen auf entfernte Objekte, nämlich die Partner. Jedes Objekt liegt in einem virtuellen Prozessor. Wird einem Objekt gesagt, es möge sich verschieben, so wird immer der gesamte virtuelle Prozessor, in dem es liegt, verschoben, also auch alle Objekte, auf die lokale Referenzen vorliegen können, sowie alle Objekte, die auf das zu verschiebende Objekt lokale Referenzen haben können. Das Problem der Dateninkonsistenz tritt nicht mehr auf. In diesem Beispiel bedeutet dies, dass der Vertrag, wenn er verschoben wird, immer auch seinen Bezeichner-String und den Vektor partnerListe mitnimmt. Die entfernten Referenzen auf die Partner bleiben erhalten, auch wenn der Vertrag migriert wird.

11.2 Nebenläufigkeit

Im nächsten Beispiel soll besonders der Aspekt der Nebenläufigkeit in Dejay gezeigt werden. Ein wichtiges Anwendungsgebiet für die Nebenläufigkeit sind Probleme, die zwar einen großen Rechenaufwand erfordern, sich aber gut in Teilprobleme zerlegen lassen. Dann kann man die Teilprobleme parallel auf verschiedenen Rechnern oder Prozessoren gleichzeitig berechnen und anschließend wieder kombinieren. Beispiele hierfür sind Wettervorhersagen, physikalische Simulationen (z.B. Simulation von Atombombentests) oder grafische Algorithmen (z.B. Animation von Trickfilmen). Ein Beispiel, das diese Eigenschaft besitzt, ist die Berechnung eines fraktalen Bildes. Ein besonders interessanter Vertreter solcher Fraktale ist das so genannte Apfelmännchen. Der Vorteil dieses Problems ist einerseits, dass der Algorithmus sehr kompakt und einfach zu Programmieren ist, aber dennoch einen hohen Rechenaufwand erzeugt, und andererseits, dass das Ergebnis bei geeigneter Interpretation ein faszinierendes Bild erzeugt, eben das Apfelmännchen. Der Algorithmus beruht auf einer rekursiven Formel, die zwei Parameter als Startwerte bekommt. Für jeden dieser Berechnung kann der Ergebniswert entweder unendlich werden (divergieren) oder sich einem bestimmten Wert annähern (konvergieren). Um entscheiden zu können, ob er divergiert oder konvergiert, muss man unterschiedlich viele Rekursionen durchlaufen. Bei der grafischen Ausgabe werden die Startwerte (die zwei Bestandteile einer komplexen Zahl) als Koordinaten verwendet und die Rekursionstiefe in einen Farbwert umgewandelt, so dass sich ein Bild ergibt. An den Rändern der sich bildenden Figur kann man die Berechnung beliebig verfeinern, das heißt,

man kann beliebig tief in das Bild »hineinzoomen«, wobei immer weitere Verschnörkelungen erkennbar werden, die sich erstaunlicherweise ähneln. Weitere Informationen zum faszinierenden Thema der fraktalen Bilder und ihrer Berechnung findet man in [Peitgen et al. 1992]. Die Berechnung des Bildes lässt sich in viele Einzelprobleme zerlegen, die einzeln und verteilt berechnet werden können, bis am Ende das ganze Bild wieder zusammengesetzt werden kann. Diese Berechnungen sollen mehrere Server übernehmen, die ihre Teilaufgaben von einem Client zugeordnet bekommen. Es ist außerdem möglich gleichzeitig mehrere Clients zu starten.

Der Programmcode für dieses Beispiel, vor allem für die grafische Darstellung, ist zu umfangreich, als dass er hier vollständig abgedruckt werden könnte. Doch alle wesentlichen Teile, insbesondere jene, die für die Verteilung und Nebenläufigkeit wichtig sind, werden hier gezeigt. Es werden im wesentlichen die folgenden Klassen benötigt:

- einen Client (ApfelMannClient), der die Berechnungen in Auftrag gibt und wieder zusammenfügt
- eine dazugehörige Benutzeroberfläche (ApfelMannGUI)
- und einen Berechnungsserver (Server), der den eigentlichen Algorithmus ausführt
- eine Teilaufgabenbeschreibung (Aufgabe)
- ein Berechnungsergebnis (Ergebnis)

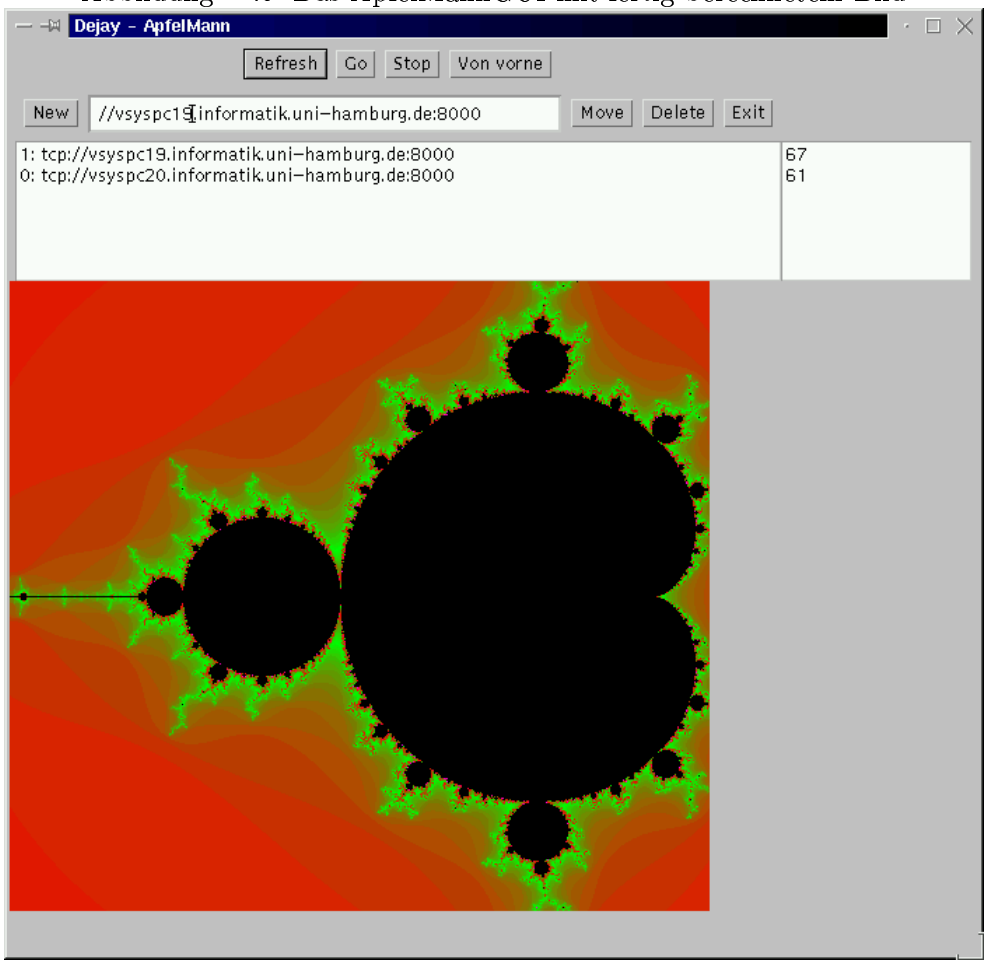
Das User-Interface

In Abbildung 11.5 ist das GUI nach der vollständigen Berechnung gezeigt. Seine wichtigste Aufgabe ist natürlich die Darstellung des Apfelmännchens. Aber es dient auch zur Steuerung der gesamten Anwendung. Von hier aus wird die Berechnung gestartet und der Rechenverlauf dargestellt sowie die Server verwaltet. Neue Berechnungsserver können von hier aus hinzugefügt und gelöscht und – da mit Dejay die entfernte Erzeugung von Objekten möglich ist, was später noch diskutieren wird – auch neu erzeugt werden. Man kann Server auch verschieben, sogar während des Betriebes, was ebenfalls erst später betrachtet wird. Die beteiligten Server werden angezeigt, zusammen mit der Information, wie viele Teile des ganzen Bildes sie schon berechnet haben. Im gezeigten Bild gibt es zwei Server, die jeweils etwa die Hälfte der 128 Teilbilder berechnet haben. Der Programmcode des GUI wird hier nicht gezeigt. Er hat mit der Nebenläufigkeit oder der Verteilung nichts zu tun.

Der Client

Die wichtigste Klasse in diesem Beispiel ist der ApfelMannClient. Diese Klasse wird aufgerufen, um das Programm zu starten, von hier aus werden dann das GUI ebenso wie die Server erzeugt und hier findet die Steuerung der Nebenläufigkeit und die entfernte Kommunikation statt. Der ApfelMannClient ist einerseits dafür zuständig, dass die Benutzerinteraktion umgesetzt wird, andererseits

Abbildung 11.5: Das ApfelMannGUI mit fertig berechnetem Bild



verwaltet er die Server und bearbeitet die Ergebnisse. Dazu hat der ApfelMann-Client zwei Hashtables. Im ersten, meineServer, sind alle Server abgelegt, wobei der key eine fortlaufende Nummer ist. Im zweiten, meineErgebnisse, werden die Ergebnisse abgelegt, wozu derselbe key verwendet wird.

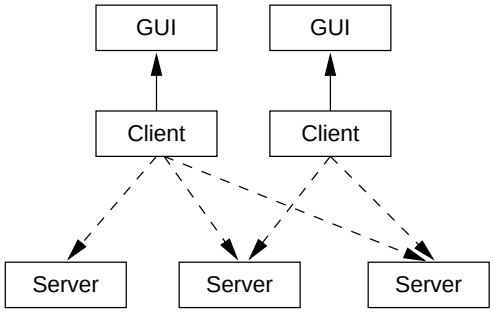
Zur Verwaltung der Server hat der ApfelMannClient die folgenden wesentlichen Methoden:

- neuerServer()
- bewegeServer()
- gibNeueAufgabe()

Damit kann der ApfelMannClient neue Server erzeugen, sie verschieben und ihnen Teilaufgaben zuweisen.

Für das (wohlgemerkt entfernte) Erzeugen eines Servers wird die Methode neuerServer() vom ApfelMannGUI aufgerufen und bekommt eine Adresse übergeben, an der ein DjProcessor und darin dann ein Server erzeugt wird. Falls gerade

Abbildung 11.6: Der Client ist die zentrale Klasse



eine Berechnung läuft und noch Rechtecke zu berechnen sind, wird dem neuen Server gleich eine Aufgabe gegeben.

Obwohl dieser Abschnitt hauptsächlich den Aspekt Nebenläufigkeit behandelt, wird hier auch die Migration gezeigt, wofür die Methode bewegeServer() dient. Dieser Methode werden lediglich zwei Strings übergeben, einer zur Identifikation der Servers, der bewegt werden soll, der zweite als Zieladresse, wohin er migrieren soll. Diese Methode kann im laufenden Betrieb aufgerufen werden, und der Server bewegt sich zwischen der Berechnung zweier Aufgaben von einem auf einen anderen Rechner.

Nun zur eigentlichen Berechnung. Die Methode gibNeueAufgabe() weist einem Server eine Teilaufgabe zu. Damit die Berechnung parallel erfolgt, darf der Aufruf des Servers nicht wie ein normaler Methodenaufruf blockieren und auf das Ergebnis warten. Daher wird der entscheidende Aufruf (server.berechne()) asynchron ausgeführt, was durch die Angabe des zusätzlichen Parameters Dejay.ASYNC erreicht wird. Danach hat man die Möglichkeit, mit isAvailable() herauszufinden, ob das Ergebnis des Aufrufs schon vorhanden ist, oder mit waitForResult() explizit darauf zu warten. Am einfachsten ist allerdings, wenn man das Ergebnis einfach dann benutzt, wenn man es benötigt (wait-by-necessity). In diesem Beispiel wird eine Referenz auf das Ergebnis einfach in eine Hashtabelle gestellt und erst später darauf zugegriffen.

Der Client ist selbst als Thread implementiert, damit er nebenläufig zum GUI läuft. Die run()-Methode des ApfelMannClients enthält mehrere Schleifen, wovon die erste einmal durch die Liste der Server läuft und jedem etwas zu tun gibt, so lange noch Rechtecke und damit zu berechnende Teilaufgaben vorhanden sind. Da die Methode gibNeueAufgabe() einen asynchronen Aufruf an den Server schickt, kann die Schleife durchlaufen werden, ohne an dieser Stelle warten zu müssen. Die folgende zweite Schleife läuft, so lange noch Ergebnisse erwartet werden, durch die Liste der Ergebnisse und überprüft mit isAvailable(), ob sie schon fertig berechnet und beim ApfelMannClient angekommen sind. Der Aufruf ergebnis.isAvailable() liefert genau dann false zurück, wenn ein getätigter asynchroner Aufruf (in gibNeueAufgabe()) noch nicht beantwortet wurde. Ist das Ergebnis da, wird es in eine Grafik konvertiert und der Eintrag im Hashtable gelöscht. Wenn noch Rechtecke übrig sind, wird dem Server gleich eine neue Aufgabe übergeben. Immer, wenn eine Aufgabe vergeben wird, dann wird die Variable Anzahl inkrementiert, und wenn ein Ergebnis bearbeitet wurde, dann

wird sie dekrementiert. Sobald **anzahl** gleich null ist, sind alle Teilaufgaben erledigt und das Bild ist fertig. Hier nun der Code in Auszügen:

```
import java.util.*;
import dejay.base.*;

public class ApfelMannClient extends Thread {

   ApfelMannGUI gui;

   Hashtable meineServer = new Hashtable();
   Hashtable meineErgebnisse = new Hashtable();
   Hashtable meineResults = new Hashtable();


  public void neuerServer(String wo) {
    DjProcessor processor = null;
    DjServer server = null;
    String key = "";
    if(wo.length() > 0) {
      try {
        String adresse = "tcp://"+wo;
         processor = new DjProcessor(adresse);
         server = new DjServer(processor);
        key = meineServer.size() + "";
        meineServer.put(key, server);
        if (meinThread.isAlive()) {
          if (Rechtecke.size() > 0)
            gibNeueAufgabe(key);
        }
      } catch (Exception e) {System.out.println(e);}
    }
  }


  public void bewegeServer(String welchen, String wohin) {
    int position = welchen.indexOf(":");
    String nummer = welchen.substring(0,position);
    DjServer server = (DjServer)meineServer.get(nummer);
     server.moveTo("tcp://"+wohin);
    System.out.println("Nr. "+nummer+" nach " + wohin + " verschoben...");
  }

  public void gibNeueAufgabe(Object key) {
    Ergebnis ergebnis;
    Rectangle rect = (Rectangle)(Rechtecke.firstElement());
    Rechtecke.removeElementAt(0);
```

```
    DjServer server = (DjServer)meineServer.get(key);
    Aufgabe aufgabe = new Aufgabe(rect, meinZoom, xs, ys, meineIterationsTiefe );
    ergebnis = server.berechne(aufgabe, Dejay.ASYNC);
    meineErgebnisse.put(key, ergebnis);
  }

  public void run() {
    int anzahl = 0; //rausgegebene Aufträge
    int rechtecke= Rechtecke.size();;

    // erster Durchlauf (durch die Server)
    for( Enumeration e = meineServer.keys(); e.hasMoreElements();  ) {
      if (Rechtecke.size() > 0) {
        Object key =  e.nextElement();
        gibNeueAufgabe(key);
        anzahl++;
      }
    }

    // alle weiteren Durchläufe (durch die Ergebnisse)
    do {
      for( Enumeration e = meineResults.keys(); e.hasMoreElements();  ) {
        Object key = e.nextElement();
        DjErgebnis ergebnis = (DjErgebnis) meineErgebnisse.get(key);
        if(ergebnis.isAvailable()) {
          konvertiere(ergebnis.getCopy());
          if(Rechtecke.size() > 0) {
            gibNeueAufgabe(key);
            anzahl++;
          }
        gui.zeichne();
        anzahl--;
      }
    }
  }
}
```

Der Server

Der Server ist ein recht einfaches Objekt. Wesentlich ist einzig die Methode berechne(), die vom Client aufgerufen wird. Sie bekommt als Parameter ein Aufgaben-Objekt und gibt ein Objekt vom Typ Ergebnis zurück. In diesem Ergebnis befindet sich dann ein SerialArray, das zu jedem Pixel des berechneten Teilbildes einen Farbwert enthält. Der Code wird hier nicht gezeigt, da er keine Besonderheiten von Dejay aufweist, sondern sich so schreiben lässt, als ob er lokal benutzt würde.

Implementationsvergleiche

Dieses Beispiel eignet sich sehr gut, um verschiedene Implementierungstechniken zu vergleichen. Es ist ausreichend einfach und erfordert dennoch einen gewissen Aufwand an Kommunikation. Es wurde daher in mehreren Varianten programmiert und getestet. Im folgenden soll zum einen die Ausführungsgeschwindigkeit und zum anderen die Komplexität für den Programmierer in den jeweiligen Varianten (zumindest kurz) untersucht werden. Die Ausführungsgeschwindigkeit lässt sich recht leicht messen und objektiv vergleichen. Schwieriger wird es bei der Komplexität. Als objektives Maß soll hier die Länge des benötigten Codes herangezogen werden. Auch wenn solche Zahlen nur ein unvollkommenes Bild wiedergeben, vermitteln sie doch einen Eindruck. Dabei wurde versucht eine Objektivität möglichst zu erhalten und nicht etwa durch abweichende Implementierungen, Weglassen von Kommentaren oder Leerzeilen zu manipulieren. Konkret wurden die folgenden Kommunikationsmechanismen betrachtet:

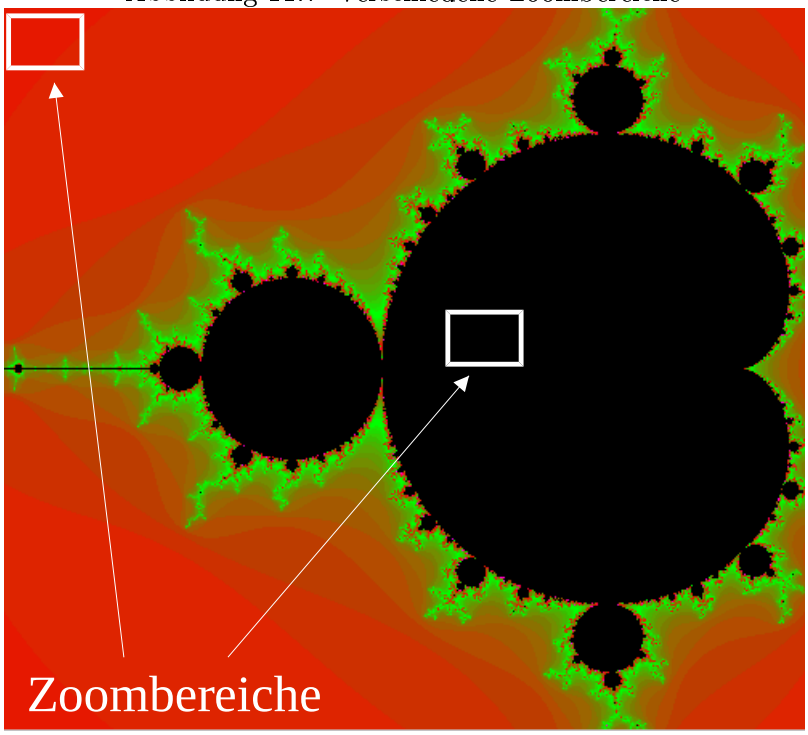
- Sockets
- RMI
- Voyager (in zwei Varianten)
- Dejay

Um einen möglichst gerechten Vergleich zu erreichen, wurde der Dejay-Code möglichst direkt in die Socket-, RMI- und Voyager-Implementierung übernommen. Der Kommunikationsmechanimus wurde dabei möglichst angemessen verwendet. Auf der Clientseite musste in den meisten Varianten ein Thread, der die asynchrone Kommunikation zu jeweils einem Server übernimmt, hinzugefügt werden. Bei Dejay wird diese Funktion bereits durch den asynchronen Aufruf und das »wait-by-necessity« bereitgestellt. Auch Voyager bietet einen asynchronen Aufruf durch das Future-Konzept. Daher wurde die Implementierung von Voyager einmal mit Futures und einmal ähnlich zu der Socket- und RMI-Implementierung programmiert.

Zunächst einige Bemerkungen zur Ausführungsgeschwindigkeit. Um verschiedene Szenarien zu betrachten, bietet ein fraktales Bild gute Möglichkeiten. Die Berechnung der Pixel eines Fraktals dauert unterschiedlich lange, je nachdem wie oft die Rekursion durchgeführt werden muss, um einen Grenzwert zu erreichen. Man kann also durch »Hineinzoomen« in unterschiedliche Bereiche des Bildes unterschiedlich komplexe Berechnungen erreichen.

Zoomt man zum Beispiel an den Rand des Bildes, so erhält man ein einfarbig rotes Bild, das jedoch sehr schnell berechnet wird (circa 10 Millisekunden), weil bei jedem Pixel nur eine Iteration durchgeführt werden muss. Dies ist also ein sehr kommunikationsintensives Szenario, es findet fast alle 10 Millisekunden ein entfernter Aufruf statt. Zoomt man hingegen in einen Bereich in der Mitte des Bildes, so ist das komplette Bild schwarz. Bei jedem Pixel muss die Rekursion bis zur maximalen Iterationstiefe (in diesem Beispiel auf 401 festgelegt) durchlaufen werden. Hierbei haben die Kommunikationsmechanismen längere Pausen von

Abbildung 11.7: Verschiedene Zoombereiche



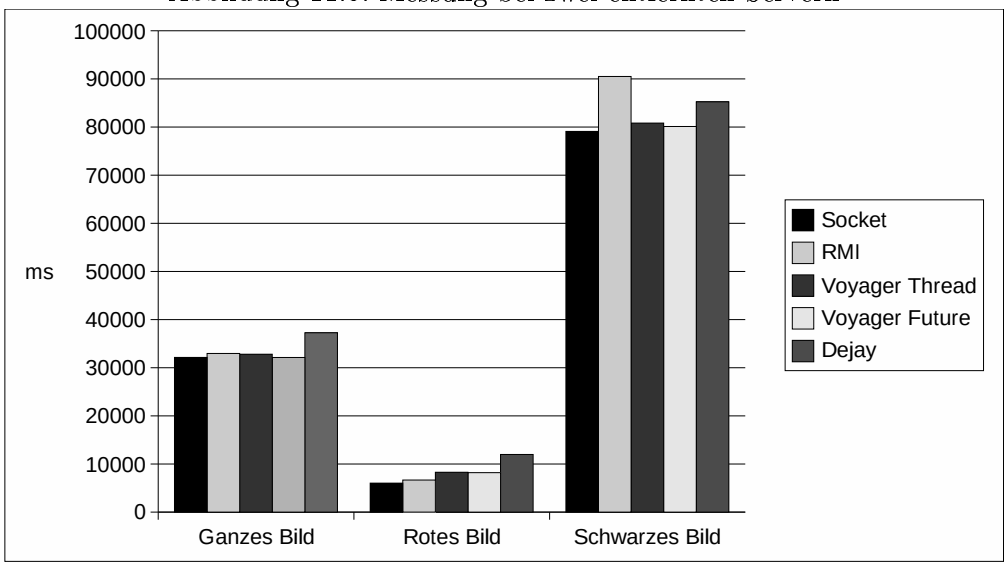
jeweils etwa anderthalb Sekunden. Es werden jedoch wie beim »roten« Bild genau 128 Ergebnisse erzeugt und übertragen.

Als Rechner standen drei PCs mit Intel Pentium II-Prozessor, 333 MHz zur Verfügung, die über ein 10 MBit Ethernet mit einem Switch verbunden waren und auf denen jeweils der Client bzw. ein Server lief. Das Ergebnis zeigt Abbildung 11.8. Angegeben ist die geringste Zeit in Millisekunden, die das Programm mit dem jeweiligen Mechanismus benötigte, um ein ganzes, ein rotes und ein schwarzes Bild zu berechnen.

Die Ergebnisse zeigen, dass Dejay zwar langsamer ist, als andere Alternativen, aber dennoch annehmbare Ergebnisse erzielt. Dejay basiert auf Voyager und kann daher nicht schneller sein, als eine entsprechende Implementierung in Voyager. Und Voyager ebenso wie RMI verwenden ihrerseits Sockets zur Übertragung, so dass zu erwarten ist, dass die Socketimplementierung am schnellsten ist. Für kommunikationsintensive Szenarien (wie beim roten Bild) zahlt sich daher eine Implementation direkt mit Sockets aus. Doch für Szenarien, bei denen auf den Servern auch eine gewisse Rechenleistung erbracht wird, wie es in praktischen Anwendungen meist der Fall ist, bleibt der zusätzliche Zeitaufwand durch Dejay gering (unter 15% für das ganzes Bild bzw. 9% für das schwarze Bild).

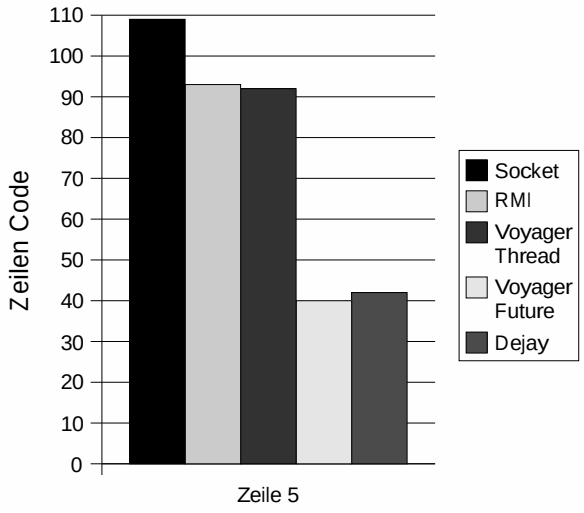
Bemerkenswert ist, dass der Overhead von Dejay im Vergleich zu Voyager für alle Szenarien etwa konstant ist. Dies liegt an der zusätzlichen Erzeugung von Objekten, die sehr zeitintensiv ist und noch nicht optimiert ist. Hier sind noch Verbesserungen von Dejay möglich, so dass es insgesamt noch näher an die Resultate von Voyager rücken kann.

Abbildung 11.8: Messung bei zwei entfernten Servern



Doch das Ziel war vor allem die Programmierung verteilter Systeme zu vereinfachen. Daher soll nun der Aufwand für den Programmierer, gemessen in Codezeilen, betrachtet werden. Die Client- und die Serverseite werden getrennt untersucht.

Abbildung 11.9: Codelänge für den Client



Beim Client ist ein zusätzlicher Aufwand für die Simulation asynchroner Kommunikation durch einen zusätzlichen Thread bei drei Alternativen nötig, nämlich bei den Sockets, RMI und der ersten Voyager-Variante. Dadurch ist hier ein sehr großer Unterschied zu Dejay zu erkennen, der zwischen 120% und 160% mehr Code in den relevanten Methoden liegt. Die Möglichkeit der asynchronen Kommunikation, die in Voyager (Future-Variante) und Dejay zur Verfügung steht, stellt also eine erhebliche Vereinfachung dar. Diese Voyager-Variante und

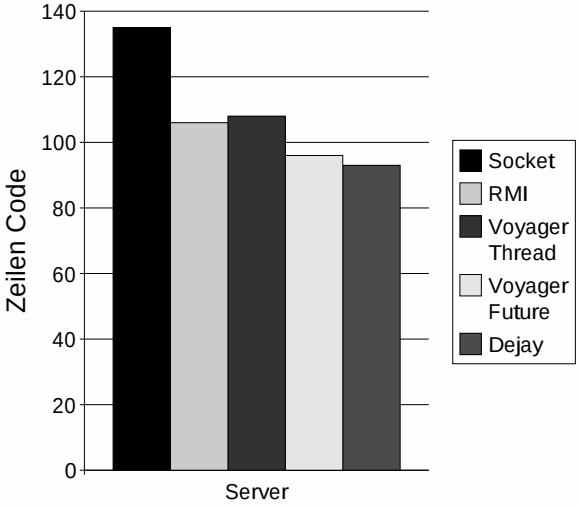
das Dejay-Programm sind etwa gleichlang. Doch Dejay ist viel einfacher in der Benutzung. Zum Beispiel ist der asynchrone Aufruf bei Dejay deutlich einfacher als bei Voyager. Bei Voyager wird eine statische Methode der Klasse Future aufgerufen, der eine Referenz auf das aufzurufende Objekt, der Methodenname als String und die Parameter als Object-Array übergeben wird.

```
//Voyager
Result result = Future.invoke(server,"berechne", new Object[]{aufgabe});
```

Bei Dejay dagegen wird lediglich ein zusätzlicher Parameter beim Aufruf eines entfernten Objektes angegeben, der asynchrone Aufruf entspricht aber ansonsten der normalen Syntax eines Methodenaufrufs.

```
// Dejay
DjErgebnis ergebnis = server.berechne(aufgabe, Dejay.ASYNC);
```

Abbildung 11.10: Codelänge des Servers (kommunikationsrelevante Teile)



Das vielleicht objektivere Bild erhält man auf der Serverseite. Die Codelängen der jeweils für die Kommunikation relevanten Teile der unterschiedlichen Varianten sind in Abbildung 11.10 aufgeführt. Für die Socket-Variante muss ein erheblicher Aufwand für die Aufbereitung der Daten zu einem Bytestrom getrieben werden, so dass diese Variante deutlich am meisten Code erfordert. Für die Varianten Socket und RMI muss auf der Serverseite eine Instanz erzeugt werden, zu der sich der Client anschließend verbinden kann, da in diesen Varianten keine entfernte Erzeugung von Objekten möglich ist. Außerdem muss dieses Objekt von einem Administrator erzeugt werden und ein Namensdienst verwendet werden, was die Komplexität zusätzlich erhöht. Die erste Voyager-Variante wurde synonym zum RMI-Beispiel gehalten, während in der zweiten die entfernte Erzeugung genutzt wird. Dann sind das Dejay- und das Voyager-Programm vergleichbar lang. Allerdings muss für den Voyager-Server ein zusätzliches Interface (IServer) erstellt werden, was in diese Zahlen nicht aufgenommen ist. Bei

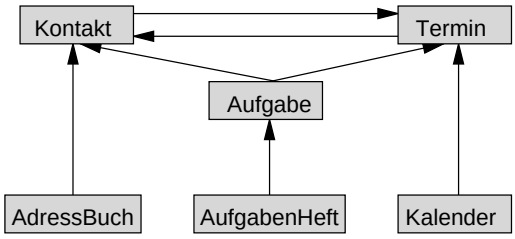
Dejay entfällt auch dieser Schritt, der Server muss nur mit dem Dejay-Compiler übersetzt werden.

11.3 Persistenz

Als Beispiel für die Persistenz dient im Folgenden ein persönlicher Datenassistenten. Dieser soll Adressen, Termine und eine Aufgabenliste verwalten und von einer beliebigen Stelle in einem Netzwerk benutzt werden können und, wenn er nicht benötigt wird, persistent sein. Bei diesem Beispiel kommt es nicht darauf an, eine besonders schöne Anwendung zu entwickeln, die alles kann, sondern auch hier geht es vor allem darum, die Vorteile von Dejay, insbesondere unter dem Aspekt der Persistenz, zu zeigen.

Die Anwendung besteht eigentlich aus drei verschiedenen Anwendungen: einem Adressbuch, einem Terminkalender und einem Aufgabenheft. Sie soll auch so entworfen werden, dass sich diese Teile auch unabhängig voneinander nutzen lassen. Doch die Daten, auf denen sie arbeiten, hängen auf natürliche Weise zusammen: Ein Kontakt im Adressbuch kann neben Name, Straße usw. auch einen Termin, wie etwa den Geburtstag, enthalten; eine Aufgabe des Aufgabenhefts kann neben einer Beschreibung einen Termin (z.B. den Abgabetermin) und einen Kontakt enthalten; und ein Termin schließlich soll neben normalen Einträgen auch Kontakte oder Aufgaben enthalten können.

Abbildung 11.11: Klassen des Datenassistenten



Es werden also, neben einer grafischen Benutzeroberfläche (GUI), die folgenden Klassen benötigt:

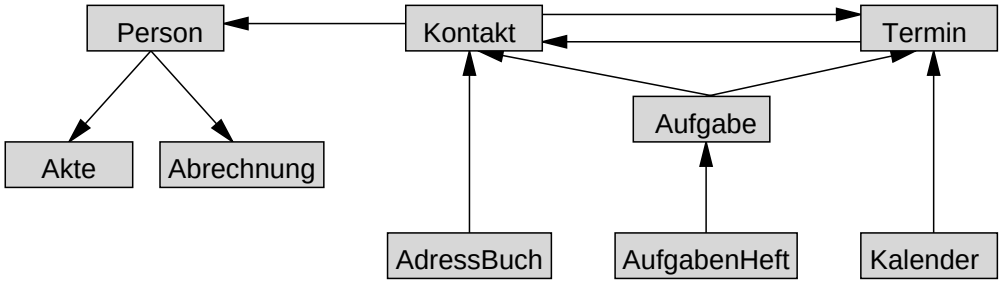
- AdressBuch, enthält und verwaltet alle Kontakte
- TerminKalender, enthält und verwaltet alle Termine
- AufgabenHeft, enthält und verwaltet alle Aufgaben
- Aufgabe
- Kontakt
- Termin

Diese Problemstellung lässt sich sicherlich mit JDBC erledigen, doch dafür muss ein erheblicher Aufwand betrieben werden, um den *impedance mismatch* zwischen relationalen Datenbanken und objektorientierter Programmierung zu

überwinden. Viel leichter wäre es, dies mit einer objektorientierten Datenbank zu lösen, in die man einfach die oben angegebenen Objekte ablegen und bei Bedarf wieder darauf zugreifen. In OO-Datenbanken definiert man dafür ein Wurzelobjekt, und alle von diesem Objekt erreichbaren Objekte werden gemeinsam persistent gemacht. Das Wurzelobjekt muss also sowohl auf AdressBuch als auch auf AufgabenHeft und TerminKalender zeigen, um alle Objekte zu erreichen.

Angenommen die Personalverwaltung möchte dieses System nutzen, wünscht sich aber eine Erweiterung. Sie möchten aus dem Adressbuch von einem Kontakt – zum Beispiel einem Angestellten – direkt zu dessen Personalakte zugreifen können. Es muss also eine zusätzliche Referenz zu einem Objekt Person in die Klasse Kontakt eingebaut werden, an der wiederum die Objekte wie PersonalAkte und PersonalAbrechnung hängen, wie in Abbildung 11.12 dargestellt.

Abbildung 11.12: Erweiterung für die Verwaltung



Aber möglicherweise sollen die Personendaten, die eine beträchtliche Größe erreichen können, in einer anderen Datenbank liegen als die des Datenassistenten. Dies ist mit Java und OO-Datenbanken nicht mehr so einfach hinzubekommen. Dafür müssten zum Beispiel die Referenzen von Kontakt zu Person vor dem Speichern gelöscht und nach dem Laden wieder eingesetzt werden.

In Dejay dagegen werden die Objekte des Datenassistenten und die Personaldaten einfach in zwei unterschiedliche virtuelle Prozessoren platziert, wie in Abbildung 11.13 gezeigt. Referenzen vom Kontakt zu Personendaten sind dann entfernte Referenzen. Bei Bedarf kann auf die Personendaten zugegriffen werden, die entweder im Hauptspeicher (möglicherweise auf einem entfernten Rechner) oder in einer Datenbank liegen und beim Aufruf automatisch wieder aktiviert werden.

Eine mögliche grafische Oberfläche zu dieser Anwendung wird in 11.14 dargestellt. Sie ermöglicht es, neue Kontakte, neue Termine und neue Aufgaben zu erstellen oder sie zu löschen. Um den Zustand des Datenassistenten zu speichern, muss die Methode persist() auf den umgebenden Prozessor aufgerufen werden, was in der Beispielimplementation immer beim Beenden des Programms oder bei Betätigung des Button »Speichern« geschieht. Die Anwendung wird gestartet von einer weiteren Klasse, die Assistent heißt und im Folgenden aufgeführt ist. Darin wird entweder versucht einen Datenassistenten aus einer Datenbank zu laden oder, wenn dies nicht gelingt, wird ein neuer Datenassistent angelegt und zunächst einmal ohne weitere Objekte darin persistent gemacht. Dann wird das GUI gestartet, das die eventuell vorhandenen Daten anzeigt und die Möglichkeit bietet, neue zu erstellen.

Abbildung 11.13: Aufteilung in verschiedene virtuelle Prozessoren in Dejay

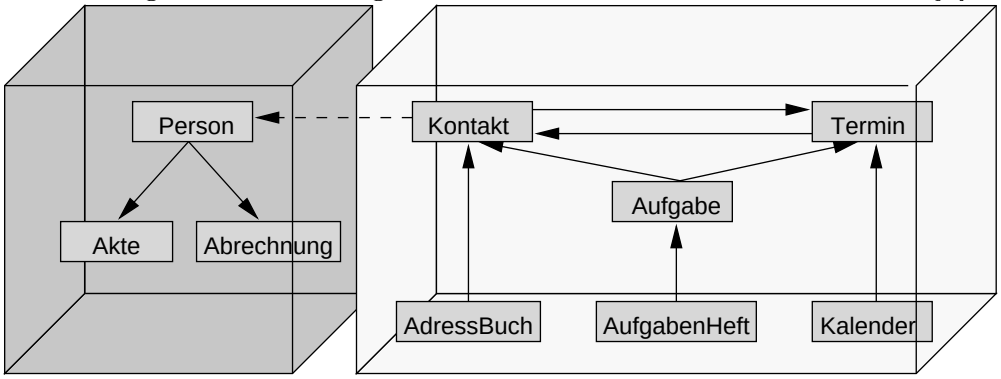


Abbildung 11.14: Das GUI zum Datenassistenten


```
public class Assistent implements java.io.Serializable {

  public DjAdressBuch adressBuch;
  public DjTerminKalender terminKalender;
  public DjAufgabenHeft aufgabenHeft;
  public DjProcessor processor;

  GUI gui = new GUI(this);

  public static void main(String[] args) {
    // Datenassistenten laden oder neu erzeugen
    processor = Dejay.getProcessorFromDB("Prozessor.db", "DatenAssistent", adresse);
    if ( processor!=null ) {
      // die drei Objekte aus dem Prozessor holen
      try {
       terminKalender = (DjTerminKalender) processor.getObjectByName("kalender");
       adressBuch = (DjAdressBuch) processor.getObjectByName("adressbuch");
       aufgabenHeft = (DjAufgabenHeft) processor.getObjectByName("aufgabenheft");
      } catch (Exception e) {}
    } else {
```

```
        System.out.println("Nothing found in Database. Creating a new Data-assistent.");

        processor = new DjProcessor("Prozessor.db", "DatenAssistent", adresse);
        terminKalender = new DjTerminKalender( processor );
        terminKalender.registerByName( "kalender" );
        adressBuch = new DjAdressBuch( processor );
        adressBuch.registerByName( "adressbuch" );
        aufgabenHeft = new DjAufgabenHeft( processor );
        aufgabenHeft.registerByName( "aufgabenheft" );

        //So erstmal persistent machen
        processor.persist();

        gui = new GUI( this );
        gui.pack();
        gui.show();
    }
  }
}
```

Kapitel 12

Anwendung im Rahmen einer Technologiestudie im industriellen Umfeld

Die in den vorangegangenen Kapiteln vorgestellten und in Dejay implementierten Konzepte sollen in diesem Kapitel in Hinblick auf Relevanz und Einsetzbarkeit in der Praxis untersucht werden.

Der Nachweis soll anhand eines praxisrelevanten Beispiels erbracht werden, auf das die entwickelten Konzepte angewandt werden. An dieses werden eine Reihe von Forderungen gestellt, die es idealerweise erfüllen sollte. Es kann einen gewissen Rahmen an Größe und Komplexität nicht überschreiten, um innerhalb dieser Arbeit realisierbar und darstellbar zu bleiben. Andererseits muss es eine gewisse Größe erreichen, um aussagefähige Ergebnisse zu liefern. Es sollte ein aus der Praxis entnommenes Problem darstellen und den Bezug zur Realität sicherstellen, so dass der Rahmen der rein akademischen Konzeption verlassen werden kann.

Die meisten praktischen Anwendungen verteilter Systeme weisen Aspekte von Verteilung, Nebenläufigkeit und Persistenz auf, diese sind aber nicht immer deutlich erkennbar oder mit Konzepten niedriger Abstraktion einfach lösbar. Um die in dieser Arbeit vorgestellten Konzepte, insbesondere die Vereinigung von Verteilung, Nebenläufigkeit und Persistenz zeigen zu können, sollte die Anwendung an alle drei Aspekte hohe und offensichtliche Anforderungen haben. Die Problemstellung sollte einen hohen Grad an Verteilung, möglichst sogar eine physikalisch sichtbare Verteilung der Anwender oder der Anwendungsteile aufweisen. Eine Notwendigkeit für die nebenläufige Verarbeitung sollte durch entsprechend hohe Anforderungen an die Antwortzeiten offensichtlich sein. Die für die Anwendung relevanten Daten sollten eine hohe langfristige Bedeutung für die Anwendung haben, so dass Persistenz notwendigerweise behandelt werden muss, und eine komplexe Struktur aufweisen, die eine Behandlung der Persistenz mit objektorientierten Mitteln nahe legt. Wünschenswert wäre weiterhin, dass die Anwendung schon mit herkömmlichen Technologien, wie sie etwa im ersten Teil der Arbeit vorgestellt wurden, umgesetzt wurde, um einen Vergleich zu anderen Ansätzen zu ermöglichen.

Als ein solches Beispiel wurde das Logistiksystem des Hamburger Containerhafens Altenwerder gewählt. In Zusammenarbeit mit der Firma Software Design und Management (sd&m) wurde in einer Technologiestudie ein bei dieser Firma in der Entwicklung befindliches System herangezogen und konkurrierend mit Dejay implementiert. Eine ausführliche Darstellung der Ergebnisse dieses Projektes findet sich in [Braubach und Pokahr 1999].

Da sich die Umsetzung der in dieser Arbeit entwickelten Konzepte in praktische Werkzeuge noch in einem Stadium der Forschung und Entwicklung befinden und die vorhandenen Tools zwar in vollem Umfang funktionsfähig aber dennoch einen prototypischen Charakter haben und noch kein fertiges Produkt darstellen, kann hier sicherlich kein wirklicher Einsatz der in dieser Studie entwickelten Software in einem unternehmenskritischen Bereich erwartet werden. Doch ein Nachweis der praktischen Einsetzbarkeit und der Sinnhaftigkeit der vorgestellten Konzepte kann geführt werden.

12.1 Systembeschreibung der Fallstudie

Der Kontext der dieser Evaluation zugrunde liegenden Anwendung ist die Steuerungssoftware für den Hamburger Containerhafen. Der Containerhafen wird derzeit von der Stadt Hamburg und der Hamburger Hafen und Lager AG (HHLA) am Standort des ehemaligen Fischerdorfes Altenwerder erweitert. Hier entstehen vier neue Liegeplätze, die ein neuartiges Konzept für das Löschen, Lagern und Abtransportieren der Container, ebenso wie in umgekehrter Richtung ermöglichen, das eine sehr hohen Grad an Automatisierung erlaubt. Die hierfür notwendige Datenverarbeitungs- und Steuerungsanlage stellt ein verteiltes System dar und wird derzeit mit Techniken, wie sie im ersten Teil der Arbeit vorgestellt wurden, von Grund auf neu implementiert.

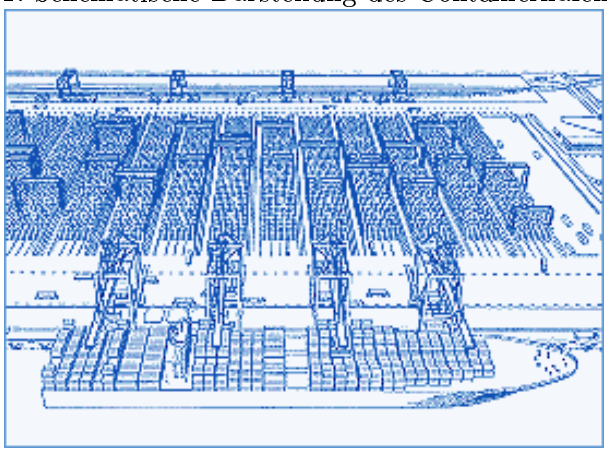
Der neue Hafen Altenwerder besitzt insgesamt 14 Containerbrücken, die zum Be- und Entladen der Schiffe an den vier Liegeplätzen dienen. Die Brücken erhalten ihre Aufträge in Form von elektronischen Lösch- und Ladelisten, die sie der Reihe nach abarbeiten. Die im neuen Hafen verwendeten Containerbrücken bestehen im Gegensatz zu den Vorgängermodellen aus zwei so genannten Laufkatzen und einer auf mittlerer Höhe gelegenen Plattform. Eine der Laufkatzen ist von einem Führer besetzt, rollt über den Ausleger und holt je einen Container aus dem Schiff, um ihn auf der Plattform abzusetzen. Dort werden durch manuelle Arbeit die so genannten Klacken entfernt, Befestigungselemente, die sich während des Schiffstransports an den Eckpunkten zu anderen Containern befinden und ein Verrutschen der Ladung verhindern. Damit die Arbeiter, die diese Arbeit erledigen, nicht durch Fahrzeuge gefährdet sind, ist die Plattform nicht ebenerdig, sondern erhöht ausgelegt. Die zweite Katze ist halbautomatisch und wird von den Arbeitern per Knopfdruck in Gang gesetzt, danach erfolgt der Transport der Container vollautomatisch, bis er an einen LKW ausgeliefert ist. Die Katze transportiert den Container auf ein wartendes Fahrzeug mit dem er in das Blocklager weiter transportiert wird. Diese Fahrzeuge sind führerlos und finden ihren Weg durch Software gesteuert, sie werden daher Autonomous Guided Vehicle oder AGV genannt.

Der Containerhafen Altenwerder wird mit insgesamt 22 Blocklagern ausgelegt, in denen die Container zwischengelagert werden und Raum für insgesamt 30.000 Container bieten. Jeder von ihnen wird von vollautomatischen hochbeinigen Portalkränen, im Fachjargon RMG genannt, bedient. Um einen möglichst schnellen Containertransfer zu erzielen, besitzt jeder Block zwei solche RMGs, die verschieden groß sind und einander passieren können. Gleichzeitiges Ein- und Auslagern ist somit möglich. Außerdem sind die zwei RMGs mit identischer Funktionalität ausgestattet, so dass bei Ausfall von einem Kran, der andere die Arbeit des ausgefallenen mitübernehmen kann. So wird die Ausfallwahrscheinlichkeit des gesamten Blocks verringert.

Die AGVs transportieren die Container zu logistisch günstig gelegenen Blocks, wo sie von den RMGs entladen und die Container wiederum innerhalb des Blocks nach logistischen Gesichtspunkten optimal gelagert werden. Kriterien hierfür sind unter anderem die Fahrwege und -zeiten für AGVs, die Auslastung der RMGs und Blocklager und die Beachtung der Auslieferungsreihenfolge. Aber auch Gefahrgutrichtlinien und die Wartbarkeit von Kühlcontainern muss beachtet werden.

Die Auslieferung findet meist an LKW statt. Eintreffende LKW müssen zunächst eine Eingangskontrolle passieren, in der Daten überprüft und ergänzt werden. Die LKW, ihre Ladung und die auszuliefernden Container werden erfasst. Das logistische System erstellt eine Liste von Anlaufpunkten, die der LKW-Fahrer in festgelegter Reihenfolge anzufahren hat. Sobald der Fahrer alle gewünschten Container geladen oder abgeliefert hat, kann er das System nach einer Ausfahrtskontrolle in Richtung seines Fahrzieles wieder verlassen. Eine schematische Darstellung des Hafengeländes und der beteiligten Einheiten ist in Abbildung 12.1zu sehen.

Abbildung 12.1: Schematische Darstellung des Containerhafens Altenwerder



Die bisherige Ablaufbeschreibung umfasste lediglich den Umschlag in einer Richtung, vom Schiff auf LKW. Natürlich findet auch ein Umschlag in umgekehrter Richtung statt. Darüber hinaus werden Container nicht nur von LKW, sondern auch von Schienenverkehr geliefert oder abtransportiert. Für die nachfolgenden Betrachtungen werden diese Aspekte vernachlässigt, da sie nach prin-

zipiell gleichen Verfahren zu behandeln sind, die Modellierung aber unnötig kompliziert machen, ohne einen Zugewinn für die Aussage zu erbringen. Außerdem steht für verschiedene Teile der Anlage bereits hoch spezialisierte Software zur Verfügung, die nicht mit modelliert werden muss. So gibt es z.B. hochspezialisierte Logistikprogramme, die das Beladen der Schiffe mit Containern plant, damit Gewichte genau ausgeglichen und Vorschriften auch in Hinsicht auf z.B. Gefahrgut und Kühlung eingehalten werden. Ein weiteres Beispiel betrifft die Stauung der Container in den verschiedenen Lagern an optimalen Positionen. Diese Softwaresysteme können für das reale System hinzugekauft werden und müssen in der Modellierung nur als anzusprechende Komponenten berücksichtigt werden.

12.2 Rahmenbedingungen

Die Realisierung dieses Projektes wird von der HHLA in Zusammenarbeit mit einem Konsortium von Firmen durchgeführt. Eine dieser Firmen ist sd&m, die maßgeblich an der Gestaltung des Gesamtsystems mitgewirkt hat sowie Teilsysteme vollverantwortlich implementieren wird. Die HHLA hat als Betreiber des gesamten Containerhafens Hamburg langjährige Erfahrung mit entsprechenden Softwaresystemen. Die Steuerung des alten Systems ist in einer proprietären Sprache mit dem Namen »M« implementiert. Diese bietet für diesen Anwendungsfall einige Vorteile, hat allerdings die Nachteile schwer wartbar und zu anderen technologischen Entwicklungen inkompatibel zu sein. Eine technologische Vorgabe der HHLA für die Implementation des neuen Steuerungssystems war die Verwendung von Java. Hierin drückt sich die Erwartung aus, mit Java auf eine zukunftssichere und moderne Sprache zu setzen. Dies bedeutet eine vollständige Ablösung des Altsystems durch ein neues. Damit sind die Einschränkungen durch Altlasten ungewöhnlich gering. Dies bedeutet aber auch, dass man sich mit diesem Projekt auf noch relativ neue Technologien einließ, für das tiefgreifende Erfahrungen noch fehlen, so dass viele Optionen erst evaluiert werden müssen.

Durch diese Rahmenbedingungen war eine Offenheit gegenüber neuartigen innovativen Entwicklungen gegeben und eine Zusammenarbeit zwischen der Universität und der Firma sd&m konnte erzielt werden. Bei Projektbeginn existierten prototypische Implementierungen, die zur Findung einer geeigneten Architektur und zur Evaluation geeigneter Implementationstechniken gedacht waren. Dabei wurden zwei grob voneinander getrennte Bereiche identifiziert und mit jeweils unterschiedlichen Technologien prototypisch entwickelt. Einerseits wurde ein dialog-orientierter Teil unter dem Namen CBS (Container-Basissystem) und andererseits ein prozessorientierter Teil unter dem Namen TLS (Terminallogistik und -steuerung) erarbeitet.

Inhalt des Container-Basissystems stellen vor allem Dialoge und Eingabemöglichkeiten dar, die zur Kommunikation an verschiedenen Stellen des Systems eingesetzt werden. Somit stellt das CBS eine administrative dialogorientierte Komponente dar, die inhaltlich relevante Elemente darstellen kann und den Benutzern so erlaubt bestimmte Aktionen einzuleiten, zu steuern oder zu

beobachten. Aufgrund der nötigen Auslegung dieses Systemteils als Mehrbenutzersystem war ein wichtiges Kriterium für die Implementierungstechnik eine Transaktionsfähigkeit, weshalb man sich für Enterprise Java Beans entschieden hat.

Aufgabe des Terminallogistik und -steuerungssystems ist die Verwaltung und Abwicklung der Umschlagprozesse. Dies umfasst die internen Steuerungsmechanismen, die für die Ablaufkontrolle der Vorgänge zuständig sind, sowie deren Optimierung nach logistischen Gesichtspunkten. Die von der TLS zu steuernden Anlagen sind physikalisch auf dem Gelände des Hafens verteilt, zum Teil sogar mobil, wie die AGVs. Inwieweit die TLS selbst zentral oder dezentral ausgelegt werden sollte, war eine noch nicht entschiedene Designfrage. Eine prototypische Implementierung auf Basis von CORBA lag zu dem Zeitpunkt vor. Als Persistenzmechanismus wurde eine objektorientierte Datenbank gewählt, da die Abbildung der Daten auf Objektstrukturen natürlicher erschien als auf Tabellen. Eine Entscheidung, ob die Daten in einer zentralen oder in verschiedenen dezentralen Datenbanken abgelegt werden sollen, war noch nicht entschieden.

Diese beiden Teilsysteme sind, obwohl in dieser Phase unterschiedlich implementiert, stark ineinander verzahnt. Beide Systeme basieren auf den gleichen Daten, bzw. stellen wichtige Daten für den anderen Teil zur Verfügung. Durch die unterschiedliche Implementierung dieser beiden Teilsysteme mit unterschiedlichen Mitteln entstand ein technologischer Graben, dessen Überwindung mit den bis dahin gewählten Techniken einen erhöhten Aufwand erforderte.

Die in diesem Projekt offenen Fragestellungen waren also zum einen, in wieweit eine verteilte Auslegung des TLS möglich oder sinnvoll ist, und zum anderen, wie die Teilsysteme TLS und CBS besser miteinander integriert werden können. Die in dieser Arbeit vorgestellten Konzepte erschienen für die Untersuchung dieser Fragestellungen sehr geeignet. Ebenso stellt das vorliegende Projekt einen ausgezeichneten Rahmen zur Evaluation von Dejay dar. Es handelt sich um ein verteiltes System, in dem viele Vorgänge von der Natur der Sache her nebenläufig geschehen oder bearbeitet werden müssen und die auftretenden Daten zu Verwaltung und Steuerung des Containerumschlags weisen eine komplexe Struktur auf, die eine objektorientierte Behandlung nahe legen. Darüber hinaus ist man bei der Entwicklung des Prototypen mit den herkömmlichen Techniken auf Probleme gestoßen, die aus der Komplexität verteilter Anwendungen im Allgemeinen und dem Zusammenspiel von Verteilung, Nebenläufigkeit und Persistenz im Besonderen resultiert.

Obwohl prototypische Implementierungen vorhanden waren, konnten diese aus firmenpolitischen Gründen nicht für diese Studie zugänglich gemacht werden. Dies erschwerte den direkten Vergleich der Implementationen miteinander. Eine wünschenswerte quantitative Untersuchung konnte daher nicht durchgeführt werden. Dies ermöglicht aber andererseits eine objektivere Vergleichbarkeit der Entwicklungsprozesse, da für die Entwicklung im Rahmen der Studie nicht mehr Information zur Verfügung stand, als den Entwicklern der firmeninternen Prototypen. Eine tiefgehende eigene Analyse war daher nötig. Es ließ sich sehr gut der Entwicklungsprozess und insbesondere die Übertragung der Analyseergebnisse auf die Implementation mit Dejay untersuchen. Es folgt daher zunächst eine Beschreibung der Analyse- und Designphase.

12.3 Analyse und Design

Um die Komplexität der Vorgänge der logistischen Steuerung und Verwaltung des Containerhafensystems beherrschbar zu machen, war ein vornehmliches Ziel dieser Studie eine geeignete Architektur für das Gesamtsystem unter Verwendung der in dieser Arbeit entwickelten Konzepte zu erarbeiten. Die Identifikation einer geeigneten Struktur auf Ebene von Systemkomponenten stand im Vordergrund. Der Detaillierungsgrad innerhalb der Komponenten soll dabei unabhängig von den übrigen Komponenten erhöht werden können, ohne die Gesamtstruktur ändern zu müssen. Daher kommt der Modellierung des Systems eine wichtige Rolle zu.

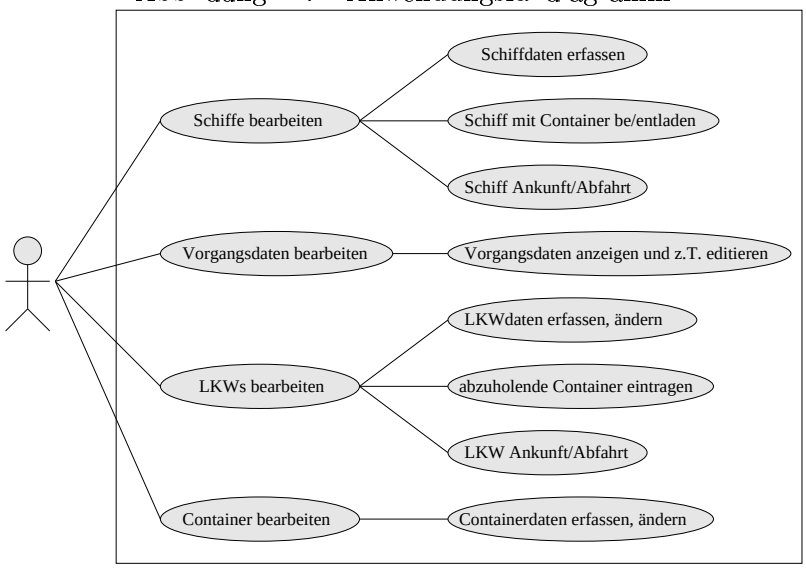
12.3.1 Analyse

Zunächst wurde eine Betrachtung aus Sicht der Anwender des Systems mit Hilfe von Use Cases [Jacobson et al. 1999] durchgeführt. Als Grundlage hierfür dienten ausführliche Gespräche mit den Mitarbeitern von sd&m, die durch ihre Erfahrung aus der eigenen Entwicklung eines Prototypen über gute Domänenkenntnisse verfügten und uns gegenüber die Rolle der zukünftigen Anwender übernahmen. Die Einsicht in die bei sd&m erstellten Dokumente war nur in sehr eingeschränktem Maße möglich, so dass der Prozess der Modellierung als eigenständig und realitätsnah bezeichnet werden kann.

Als Kernabläufe des Systems aus Sicht des Anwenders wurden die folgenden Punkte identifiziert:

- Schiffsinformation bearbeiten.
  - Schiffsdaten neu erfassen / ändern.
  - Containerladelisten eines Schiffes erstellen/ bearbeiten.
  - Ereignis Schiffsankunft/abfahrt melden.
- LKW-Information bearbeiten.
  - LKW-Daten neu erfassen / ändern.
  - Containerladeliste für LKW erstellen/bearbeiten.
  - Ereignis LKW-Ankunft melden.
- Containerinformation bearbeiten.
  - Containerdaten neu erfassen / ändern.
- Vorgangsdaten bearbeiten.
  - Vorgangsdaten anzeigen und z.T. editieren.

Abbildung 12.2: Anwendungsfalldiagramm



Eine grafische Repräsentation dieser Anwendungsfälle als UML Anwendungsfalldiagramm ist in Abbildung 12.2 dargestellt.

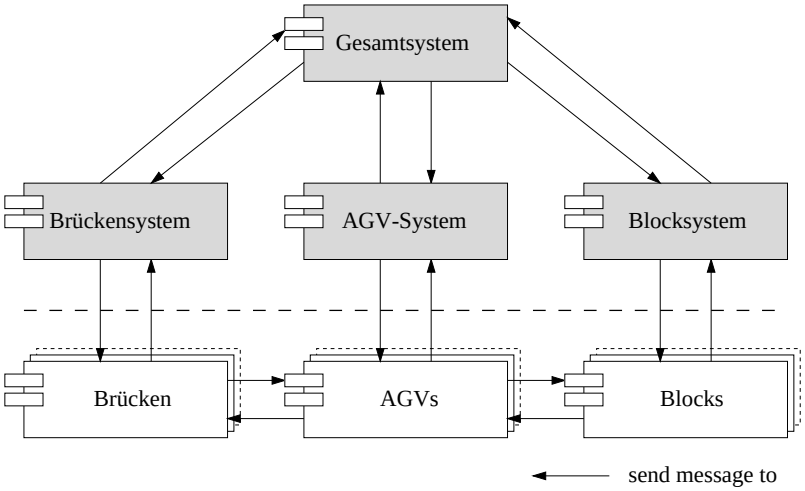
Ein wichtiger Anwendungsfall ist die Ankunft eines Schiffes, das den Prozess des Ent- und Beladens auslöst. In der folgenden Betrachtung wird die Analyse ohne Einschränkung der Gemeingültigkeit auf den Prozess des Entladens beschränkt. Physikalisch sind daran die Brücken, die AGVs und die RMGs (die zu jeweils einem Block gehören) beteiligt, die vom Logistiksystem gesteuert werden müssen. Sie finden eine direkte Entsprechung in Klassen. Zur Gliederung dieser Komponenten zu Teilsystemen werden die Klassen Brückensystem, AGV-System und Blocksystem sowie eine Klasse Gesamtsystem eingeführt. Die Aufteilung des gesamten Systems in einzelne Teilsysteme, die im realen System zwar koordiniert aber natürlich parallel arbeiten, findet damit eine geeignete Abbildung im Softwaresystem. Diese Struktur ist in Abbildung 12.3 dargestellt.

Um den Vorgang des Transports von einem einzelnen Container vom Schiff über die einzelnen Stationen bis hin zu einem LKW zu beschreiben, wird eine Klasse Umschlag eingeführt, in der die Informationen über den Verladevorgang des Containers sowohl für die Phase der Planung als auch für die Protokollierung gehalten werden.

Damit ergibt sich ein Vokabular und eine Menge von Klassen, die in der Abbildung 12.4 aufgeführt sind. Damit ist eine Grundlage für die weitere Modellierung gelegt; Eine Erhöhung des Detaillierungsgrades ist durch die Ableitung von spezialisierten Klassen, zum Beispiel für Gefahrgutcontainer, LKW mit Anhänger etc. erreichbar.

Nun sollen die Beziehungen unter diesen Klassen betrachtet werden. Das schon näher betrachtete Gesamtsystem besteht aus AGV-System, Brückensystem und Blocksystem, daher besteht zwischen dem Gesamtsystem und dem jeweiligen Teilsystem eine Kompositionsbeziehung, die ausdrückt, dass die Teile nur mit dem Ganzen existieren. Im Gegensatz dazu verwalten die Teilsysteme ihre

Abbildung 12.3: Kommunikation der TLS als Komponentendiagramm
Komponenten der Terminallogistik und -steuerung (TLS)



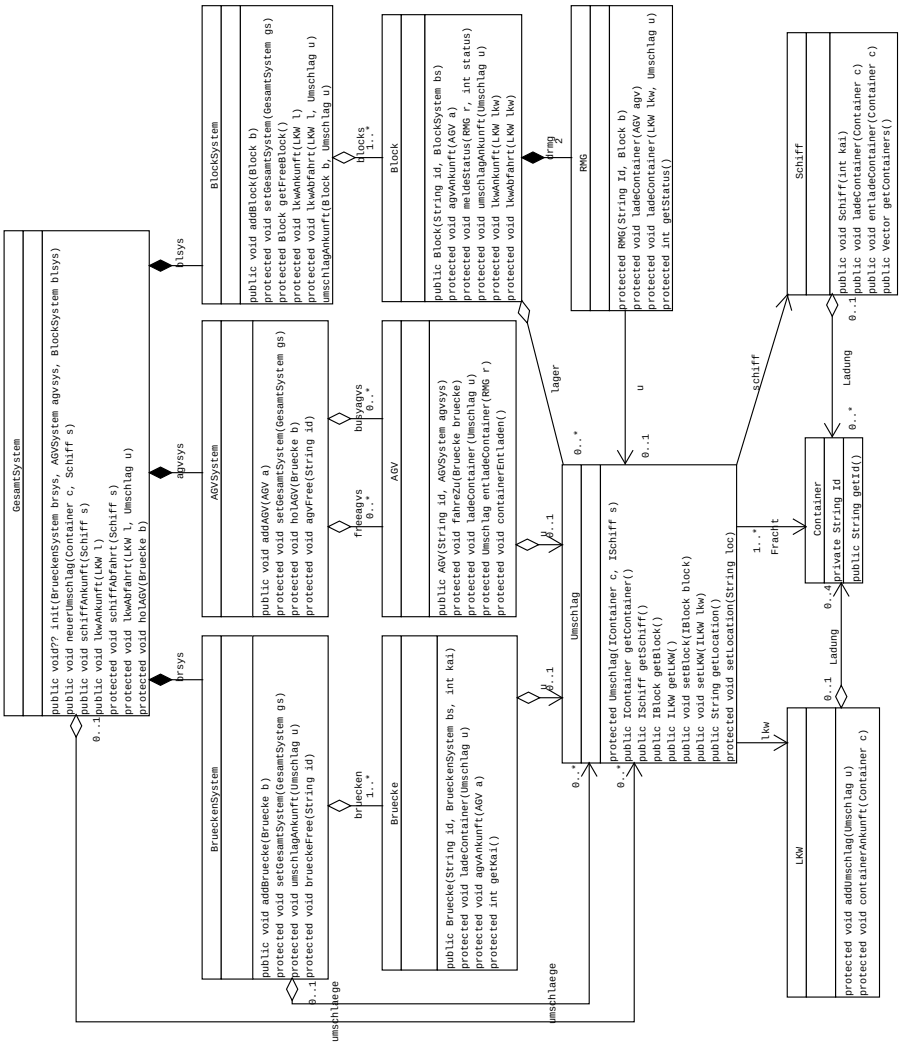
Objekte und diese können natürlich unabhängig von der Steuerung existieren. Trotzdem sind die Steuerungssysteme ohne Objekte sinnlos, was eine Aggregationsbeziehung rechtfertigt. Das Brückensystem verwaltet die 14 Brücken, das AGV-System verwaltet freie und beschäftigte AGVs und das Blocksystem hat die Kontrolle über die Blocklager und die dazugehörigen zwei RMGs.

Eine zentral wichtige Position in dem Diagramm nimmt der Umschlag ein. Er besitzt Relationen zu allen anderen Objekten, mit Ausnahme des AGV- und des Blocksystems. Das Gesamt- und Brückensystem sowie der Block kennen beliebig viele Umschläge, da die beiden Systeme für Auftragsvergaben und der Block für die Umschlagverwaltung zuständig sind. Eine Brücke, ein AGV und ein RMG dagegen sind höchstens mit einem Umschlag-Objekt zur Zeit vertraut, da sie jeweils maximal einen Umschlag zur Zeit bearbeiten können. Der Umschlag selber kapselt viele Informationen, die zur Auftragsbewältigung notwendig sind. Dazu gehört die Kenntnis zu welchem Container er gehört, auf welchem Schiff sich der Container befindet, in welchen Block er eingelagert werden soll und wer ihn abholen wird. Es kann vorkommen, dass ein Container mehrmals durch das System geschleust wird, so erklärt sich, dass es für einen Container mehrere Umschläge geben kann. Ein Container wird maximal einem Schiff und einem LKW zugeordnet. Andersherum können Schiffe und LKWs Relationen zu mehreren Containern besitzen, da sie deren Ladung darstellen.

12.3.2 Design

Nachdem als Ergebnis der Analyse die beteiligten Komponenten und deren Zusammenspiel herausgearbeitet wurde, soll im Folgenden näher auf das interne Verhalten und die modellierten Zustände einzelner Komponenten sowie deren genaue Interaktion eingegangen werden. Zur Modellierung des Verhaltens eignen sich die Zustandsdiagramme aus UML, die im Folgenden beispielhaft für drei Komponenten, die Brücke, das AGV sowie für das RMG entwickelt werden

Abbildung 12.4: Klassendiagramm

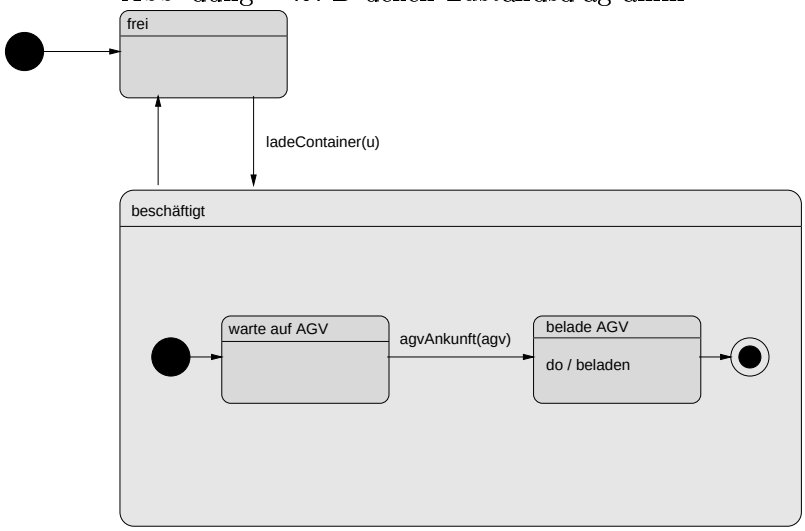


sollen. Zur Illustration der einzelnen Zustände dieser Objekte soll der Weg eines Containers und dessen Einfluss auf die beteiligten Objekte nachvollzogen werden, vom Entladen des Containers durch die Brücke bis zum Beladen des LKW durch das RMG.

Zunächst befindet sich der Container auf einem Schiff, dass an einem bestimmten Kai festliegt und entladen werden soll. Das Gesamtsystem ermittelt welche Brücken für die Entladung zur Verfügung stehen und erstellt einen Entladeplan. Dem entsprechend erhält eine freie Brücke den Auftrag mittels ladeContainer() einen Container zu entladen. Der Zustand der Brücke ändert sich also von *frei* in *beschäftigt*. Dieser Zustand ist unterteilt in Unterzustände und beginnt im Zustand *warte auf AGV* (Abb. 12.5). Dieser Zustand wechselt in *belade AGV*, wenn ein AGV and der Brücke eintrifft und dies mittels agvAnkunft() meldet. Sobald die Ladetätigkeit der Brücke abgeschlossen ist, kehrt sie in den *freien* Zustand zurück und ist somit bereit für neue Aufträge.

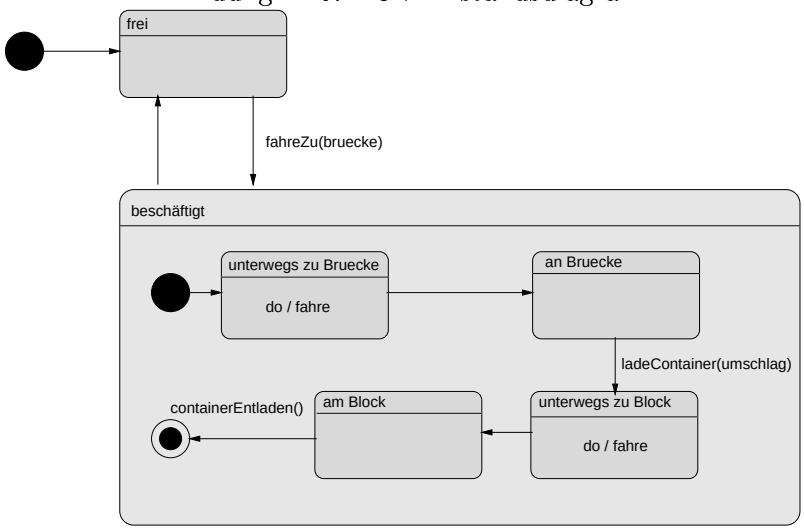
Um den Container von der Brücke abzuholen wird also ein *freies* AGV ver-

Abbildung 12.5: Brücken-Zustandsdiagramm



anlasst, den Container bei einer bestimmten Brücke mittels fahreZu(brücke) abzuholen (Abb. 12.6). Der Zustand ändert sich in *beschäftigt*, genauer in den Subzustand *unterwegs zu Brücke*. Selbständig wird er in *an Brücke* gewechselt, sobald die Fahrt beendet ist. Die Brücke lädt nun den Container auf das AGV und sendet diesem das Signal ladeContainer(umschlag), wenn sie die Aufgabe erledigt hat. Das AGV nimmt nun den Zustand *fahre zu Block* ein, der wiederum selbständig in *an Block* bei Ankunft verlassen wird. Der Block veranlasst nun ein RMG den Container vom AGV zu übernehmen. Das AGV wird benachrichtigt, wenn dieser Vorgang abgeschlossen ist und steht für neue Fahrten im Zustand *frei* wieder zur Verfügung.

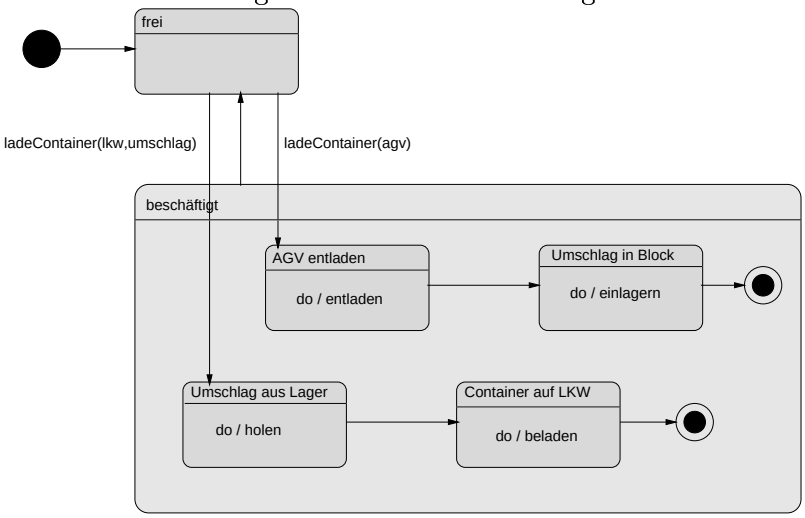
Abbildung 12.6: AGV-Zustandsdiagramm



Das RMG besitzt ebenfalls die zwei Grundzustände *frei* und *beschäftigt*, anders als die bisher betrachteten Objekte kann es jedoch zwei verschiedene

Arten von Aufgaben bewältigen, nämlich ein- und auslagern (Abb. 12.7). In dieser Situation veranlasst der Block das Einlagern des vom AGV angelieferten Containers. Daraufhin ist das RMG *beschäftigt* und tritt in den Subzustand *entlade AGV* ein. Nachdem es diese Tätigkeit abgeschlossen hat, beginnt der Subzustand *Umschlag in Block*, in dem es während des Einlagerns verweilt. Ist auch diese Aktivität beendet, kehrt es in den Zustand *frei* zurück.

Abbildung 12.7: RMG-Zustandsdiagramm



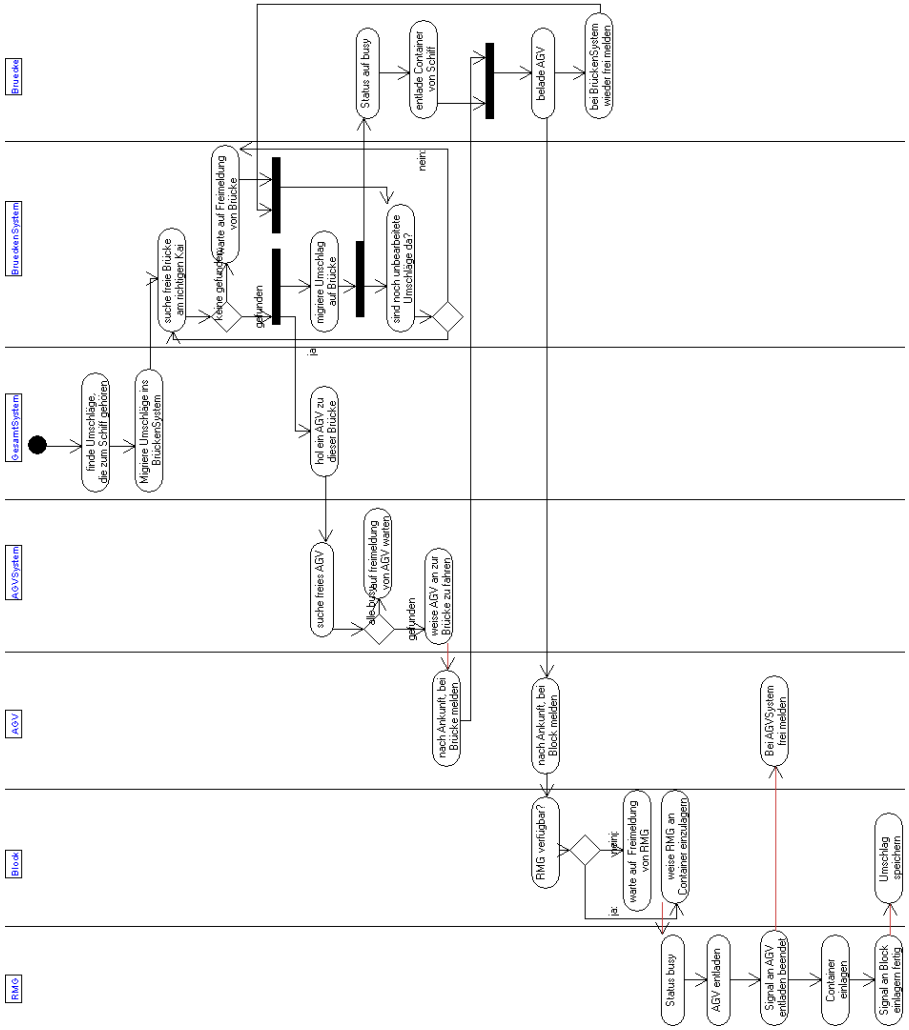
Trifft ein LKW ein, der einen Container abholen möchte, wird nach einer Reihe von Aktionen schließlich ein RMG angewiesen, den Container auszulagern. Das RMG ist wieder *beschäftigt* und befindet sich im Subzustand *Umschlag aus Lager*, den es in *Container auf LKW* nach Beendigung der zugeordneten Aktivität wechselt. *Frei* ist das RMG dann wieder, sobald die Beladetätigkeit mit dem LKW abgeschlossen ist. Der Container befindet sich nun auf dem LKW, der nach einer Endkontrolle das System verlässt.

Damit sind die Zustände der einzelnen Objekte und ihr internes Verhalten beschrieben. Um die Interaktion der Objekte genauer zu beschreiben, können in UML Aktivitätsdiagramme verwendet werden. Die an einer Aktivität beteiligten Objekte können im Aktivitätsdiagramm durch so genannte Swimlanes getrennt werden. Im Falle von verteilten Systemen können diese Swimlanes anzeigen, wann eine entfernte Kommunikation stattfindet, nämlich dann, wenn die Interaktion zweier Aktivitäten eine Swimlane kreuzen. Dies soll beispielhaft an einem Anwendungsfall dargestellt werden.

Der Anwendungsfall SchiffAnkunft() initiiert einen komplexen Ablauf im System der in Abb. 12.8 in einem Aktivitätsdiagramm dargestellt ist. Man erkennt, dass eine starke Interaktion über das gesamte verteilte System hinweg nötig ist, um diesen Anwendungsfall vollständig abzuarbeiten.

Das Gesamtsystem besitzt zunächst die Aufgabe, dafür zu sorgen, dass die Umschläge, die zu den Containern des Schiffes gehören, gefunden und an das Brückensystem weitergeleitet werden. Das Brückensystem sucht eine freie Brücke am richtigen Kai. Der weitere Verlauf verzweigt nun, abhängig davon,

Abbildung 12.8: Aktivitätsdiagramm Schiff-Ankunft



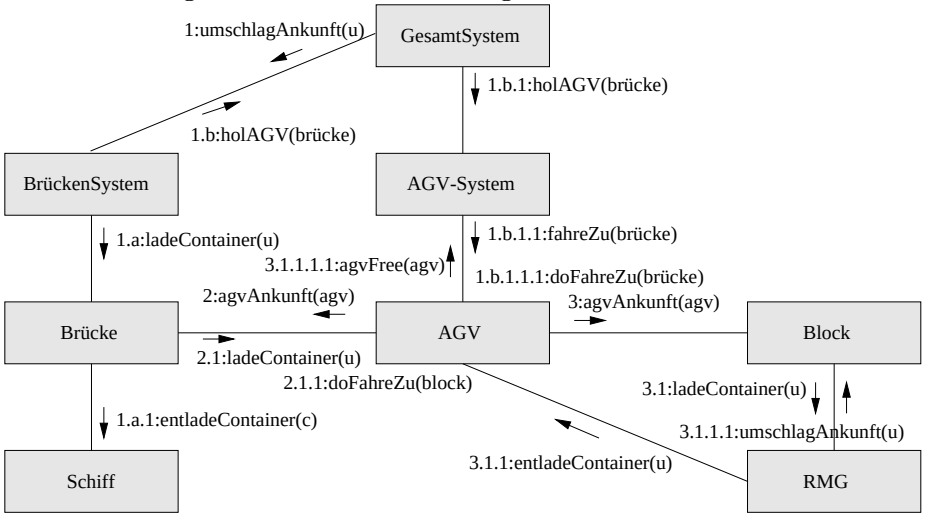
ob eine freie Brücke gefunden wurde, oder nicht. Ist eine vorhanden, dann wird einerseits eine AGV-Anforderung an das Gesamtsystem abgesetzt, andererseits die Brücke mit einem Beladeauftrag versehen. Je nachdem, ob sich noch weitere Umschläge im Brückensystem befinden, beginnt der Ablauf innerhalb des Brückensystems evtl. wieder von vorn. Die Brücke entlädt nun den Container vom Schiff und sobald ein AGV eingetroffen ist, belädt sie dieses. Das Gesamtsystem leitet die AGV-Anforderung an das AGV-System weiter, und diese weist ein AGV an, zu der Brücke zu fahren. Falls gerade alle AGVs beschäftigt sind, wartet das AGV-System, bis sich ein AGV wieder freimeldet. Das AGV fährt zur Brücke und meldet dort seine Ankunft, woraufhin es von der Brücke beladen wird. Ist der Beladevorgang beendet, macht es sich auf den Weg zum Block und teilt ihm seine Ankunft mit. Der Block sucht ein freies RMG und weist es an, den Container des AGVs einzulagern. Dazu entlädt das RMG das AGV und teilt diesem die Beendigung des Vorgangs mit. Das AGV meldet sich dann beim Steuerungssystem wieder als frei. Das RMG lagert den Container im Block ein

und teilt dem Gesamtsystem das Ende der Aktion mit.

Um die Interaktion verschiedener Objekte im Zusammenhang eines Anwendungsfalls oder einer anderen Aufgabe darzustellen eignen sich Kollaborations- bzw. Sequenzdiagramme. Im Folgenden wird das Kollaborationsdiagramm für das Entladen eines Containers näher betrachtet.

Das Gesamtsystem ruft 1.:umschlagAnkunft(umschlag) synchron beim Brückensystem auf, woraufhin dieses zwei nebenläufige Aktionen ausführt. Erstens wird die Brücke angewiesen, einen durch den Umschlag spezifizierten Container aus dem Schiff zu entladen 1.a ladeContainer(umschlag), und zweitens wird das Gesamtsystem angewiesen, ein AGV für diese Brücke zur Verfügung zu stellen 1.b:holAGV(bruecke). Das Gesamtsystem leitet diesen Aufruf an das AGV-System mittels 1.b.1:holAGV(bruecke) weiter und dieses weist ein freies Fahrzeug durch 1.b.1.1:fahreZu(brücke) an, sich auf den Weg zur Brücke zu machen. Um den Aufrufer nicht zu blockieren, ruft das AGV sich selbst asynchron mit der Nachricht 1.b.1.1.1:doFahreZu(brücke) auf. An dieser Stelle kehren alle Aufrufe mit Syntax 1.x.y.z... zurück. Zu beachten ist, dass der Aufruf 1.b.1.1:fahreZu(bruecke) an das AGV zwar synchron ist, trotzdem aber nur ein bestätigter Event ist und sofort zurückkehrt. Die Semantik dieses Aufrufs gleicht einer Aufforderung an das AGV sich auf den Weg zu machen. Sofern das AGV die Brücke erreicht, meldet es sich dort mit 2.:agvAnkunft(agv). Die Brücke ruft 2.1:ladeContainer(umschlag) des AGVs auf, nachdem es den Container durch den Aufruf 1.a.1:entladeContainer(c) aus dem Schiff geholt hat. Das AGV verwendet den oben beschriebenen Mechanismus erneut, indem es 2.2.1:doFahreZu(block) an sich selbst asynchron aufruft.

Abbildung 12.9: Kollaborationsdiagramm “Container entladen”



Es meldet sich mit 3:agvAnkunft(agv) beim Block an, der durch 3.1:ladeContainer(u) ein RMG beauftragt, den Container des AGVs zu übernehmen. Nach Ende des Stauvorgangs ruft das RMG 3.1.1.1:umschlagAnkunft(u) am Block auf, wodurch dieser von der Ankunft eines neuen Containers informiert wird. Um neue Aufträge entgegen nehmen zu können, meldet sich das AGV nun mit

3.1.1.1:agvFree(agv) frei.

Das bis hierher beschriebene Design ist unabhängig von der zu verwendenden Sprache, Plattform oder Middleware beschrieben. Auch die Entscheidung über die Verteilung (Deployment) der einzelnen Komponenten ist noch nicht entschieden. Im Folgenden wird auf die Übertragung vom Design zur Implementation mittels Dejay eingegangen.

12.4 Implementierung in Dejay

Bei der Implementierung des eben beschriebenen Modells ergeben sich eine Reihe von Freiheitsgraden, über die entschieden werden muss. Diese können in die Aspekte Verteilung, Nebenläufigkeit und Persistenz untergliedert werden. In Bezug auf die Verteilung ist die räumliche Verteilung der physikalischen Geräte vorgegeben, doch die Distribution der Objekte, die diese Geräte steuern und modellieren, und deren entfernte Kommunikation muss entschieden werden. Zwar besteht die Möglichkeit alle Objekte auf einem zentralen Rechner zu platzieren, so dass eine entfernte Kommunikation zwischen diesen nicht mehr nötig wäre, doch dann muss die Kommunikation zwischen den Objekten und den Geräten, die sie steuern, entfernt ausgelegt werden. Der Umgang mit entfernter Kommunikation ist also eine unumgängliche Notwendigkeit, zu dessen Lösung (u.a.) die im ersten Teil beschriebenen technischen Möglichkeiten zur Verfügung stehen. Auch die Nebenläufigkeit ist eine vorgegebene Notwendigkeit, da die physikalischen Geräte nebenläufig arbeiten. Persistenz schließlich ist eine Anforderung, die aus der Fülle der Daten, deren Langlebigkeit und deren langfristige Bedeutung resultiert. Es muss also auf jeden dieser Aspekte eingegangen werden.

In der Umsetzung der HHLA musste für jeden dieser Aspekte einzeln eine Entscheidung getroffen werden, wie dieser Aspekt in diesem konkreten Projekt umgesetzt werden sollte. Die in der Einleitung geäußerte Vermutung, das diese Aspekte weitgehend orthogonal zueinander sind, wurde dabei bestätigt. Im Folgenden wird jeder dieser Aspekte betrachtet und auf den Einsatz von Dejay hingewiesen.

12.4.1 Verteilung

Die hinsichtlich der Verteilung zu treffenden Entscheidungen betreffen die Aufteilung der aus dem Design entwickelten Objekte in zusammenhängende Komponenten und deren Verteilung auf physikalische Plattformen, sowie der Realisierung der entfernten Kommunikation zwischen den entfernt zueinander liegenden Teilen.

Implementierung durch Komponenten

Die Identifikation von Komponenten ist eine schwierige Aufgabe, deren gute oder schlechte Lösung tiefgreifende Auswirkungen auf das System haben kann. Die Ausführungsgeschwindigkeit kann stark darunter leiden, wenn Objekte, die viel miteinander kommunizieren in räumlich getrennte Komponenten aufgeteilt werden. Die Wiederverwendbarkeit von Software basiert in zunehmendem Maße

auf deren Unterteilung in Komponenten. Doch die Erfahrung, welche Objekte wie viel miteinander kommunizieren und wie sie gemeinsam wiederverwendet werden können zeigt meistens erst der Einsatz in der Praxis und ist im Design nur schwer korrekt vorwegzunehmen. Daher ist eine flexible Handhabung dieser Aufteilung von großem Vorteil.

Durch die in Dejay entwickelten Konzepte ist es möglich, die Zuweisung von Objekten zu Komponenten verhältnismäßig einfach und schnell zu ändern. Komponenten entstehen in Dejay durch die gemeinsame Erzeugung von Objekten im gleichen virtuellen Prozessor. Die Zugehörigkeit zu einer Komponente kann also durch das Austauschen des Parameters, auf welchem virtuellen Prozessor es erzeugt werden soll, geändert werden.

In der Umsetzung des Containerhafenbeispiels wurde jede der identifizierten Teilsysteme auf eine Komponente abgebildet. In der prototypischen Implementierung kann dies zum Teil bedeuten, dass innerhalb einer Komponente nur ein Objekt vorhanden ist, doch in einer realen Implementierung kann davon ausgegangen werden, dass diese durch komplexere Einheiten ersetzt werden und deren Darstellung auf Komponenten gerechtfertigt scheint.

Durch die Aufteilung ist eine logische Untergliederung erfolgt, die die feinstmögliche Granularität für die Verteilung, aber noch nicht deren konkrete Verteilung vorwegnimmt. Die Komponenten können unabhängig von dieser Aufteilung gemeinsam auf mehreren oder sogar auf einem einzigen zentralen Rechner ausgelegt werden. Sie können aber auch jeder auf einem getrennten Rechner installiert werden. Dabei ist die Kommunikation zwischen ko-lokalen Komponenten von den Kommunikationskosten her mit lokalen Aufrufen vergleichbar. Dadurch ergibt sich eine hohe Flexibilität, wie die Komponenten in Praxisbetrieb verteilt werden.

Entfernte Erzeugung

Die Entscheidung, welche Komponenten auf welchen Knoten installiert werden, muss in herkömmlichen Systemen, wie etwa beim RMI oder bei CORBA, weit vor dem Start des Systems getroffen werden. Die Software muss typischerweise auf den einzelnen Rechnern installiert, die einzelnen Teilsysteme gestartet und über einen Namensdienst verfügbar gemacht werden. Dies ist in der gleichen Weise mit Dejay ebenfalls möglich, durch die Möglichkeit der entfernten Erzeugung aber unnötig. Das gesamte System kann von einem zentralen Rechner aus gestartet werden, die Software kann von einem Web-Server geladen werden und die Notwendigkeit eines Namensdienstes entfällt, da nach der Erzeugung Referenzen auf die entsprechenden Objekte vorhanden sind.

Die Erzeugung und Verteilung lässt sich zum Beispiel durch eine Setup-Klasse realisieren, deren main-Methode für die einmalige Erzeugung der einzelnen Komponenten auf ihren Rechnerknoten sorgt. Dazu werden zuerst auf den wahlweise entfernten Rechnerknoten, oder zum einfachen lokalen Testen auf einem einzigen Voyager-Daemon, virtuelle Prozessoren erzeugt. In diesen Prozessoren werden die System-Objekte und die zugehörigen Einheiten gestartet und gegenseitig bekannt gemacht. Das folgende Listing zeigt, wie die Systeme und jeweils eine Einheit erzeugt werden, und dann dem Gesamtsystem die Referenzen

auf das Untersystem übergeben werden. p[1] bis p[7] sind dabei virtuelle Prozessoren, in denen die Objekte erzeugt werden, die auf beliebigen Knoten laufen können. Deren physikalischer Ort kann zum Beispiel in einer Konfigurationsdatei abgelegt und beim Start eingelesen werden. Diese Informationen können aber auch interaktiv von einem Benutzer beim Start abgefragt und dann verwendet werden. Im Beispielsystem wurden diese einfach festen Variablen in der Setup-Klasse zugewiesen.

```
 1: DjBrueckenSystem brsys  = new DjBrueckenSystem("BrueckenSystem",p[1]);
 2: DjAGVSystem      agvsys = new DjAGVSystem("AGVSystem",p[2]);
 3: DjBlockSystem    blsys  = new DjBlockSystem("BlockSystem",p[3]);
 4: DjGesamtSystem   gs     = new DjGesamtSystem("GesamtSystem",p[4]);
 5:
 6: brsys.addBruecke(new DjBruecke("bruecke_1",brsys,1,p[5]));
 7: agvsys.addAGV(new DjAGV("agv_1",agvsys,p[6]));
 8: blsys.addBlock(new DjBlock("block_1",blsys,p[7]));
 9:
10: gs.init(brsys,agvsys,blsys);
```

Entfernte Kommunikation

Gewöhnliche Middleware für die verteilte Kommunikation, wie RMI oder CORBA, müssen für einen entfernten Aufruf den gesamten Protokollstack durchlaufen, selbst wenn die Kommunikation zu Komponenten auf dem selben Rechner erfolgen. Daher ist man bemüht, in herkömmlichen Systemen die logische Aufteilung mit der physikalischen Verteilung in Übereinstimmung zu bringen.

Bei Dejay ist Kommunikation zwischen Objekten unterschiedlicher Komponenten zunächst logisch entfernt. Nur dann, wenn diese Komponenten auch physikalisch entfernt sind, wird die Kommunikation als entfernter Aufruf realisiert. Dadurch entstehen Kosten (in Form von Zeitaufwand) für die entfernte Kommunikation nur dann, wenn sie tatsächlich notwendig ist, unabhängig von der Aufteilung in logische Komponenten. Dadurch kann die logische Aufteilung unabhängig von der physikalischen Verteilung vorgenommen werden.

Daher kann die Kommunikation hinsichtlich der logischen Verteilung abgebildet werden, so wie sie im Design entwickelt wurde. Abbildung 12.3 in Abschnitt 12.3.1 stellt die Kommunikationswege in der TLS dar. Das Gesamtsystem kommuniziert mit den Untersystemen, die Untersysteme mit ihren Einheiten, und die Einheiten zu Kooperationszwecken wiederum untereinander. Die Kommunikation der Systeme beinhaltet hauptsächlich die Vergabe von Aufträgen. Die Ereignisse, die von außen in die TLS gemeldet werden (schiffAnkunft(schiff) bzw. lkwAnkunft(lkw)) werden vom Gesamtsystem bearbeitet. Das Gesamtsystem übergibt dann die Einträge der Löschliste an das Brückensystem, bzw. beauftragt das Blocksystem (lkwAnkunft(lkw)) für die Auslagerung der vom LKW verlangten Container zu sorgen. Außerdem leitet das Gesamtsystem vom Brückensystem stammende AGV-Aufträge an das AGVSystem weiter (holAGV(bruecke)). Beispielhaft soll hier die Methode schiffAnkunft() des Gesamtsystems betrachtet werden.

Das Gesamtsystem erhält bei diesem Aufruf eine Referenz auf ein Schiffsobjekt, das für das Gesamtsystem logisch entfernt ist. Desweiteren hält es entfernte Referenzen auf Umschlagsobjekte und das Brückensystem. Das Gesamtsystem muss seine Umschlagobjekte nach denen durchsuchen, deren Container sich auf dem angekommenen Schiff befinden. Der Umgang mit diesen Objekten erfolgt ganz so wie mit gewöhnlichen Objekten, lediglich die Deklaration der Variablen als entfernte Referenzen und der Aufruf umschlag.moveTo(..) (Zeile 11), der die Migration auslöst und den Umschlag auf den Rechner des Brückensystems verschiebt, unterscheiden den folgenden Codeabschnitt von gewöhnlichem Java. Die Migration erfolgt zur Optimierung der Kommunikation, damit der Umschlag bei der Brücke lokal vorliegt und nicht entfernt zugegriffen werden muss.

```
1: public void schiffAnkunft(ISchiff schiff)
2: {
3:   synchronized(umschlaege)
4:   {
5:     for(int i=0; i<umschlaege.size();)
6:     {
7:       DjUmschlag umschlag = (DjUmschlag)umschlaege.elementAt(i);
8:       if(umschlag.getSchiff().getId().equals(schiff.getId()))
9:       {
10:        umschlaege.removeElement(umschlag);
11:        umschlag.moveTo((DjBrueckenSystem)brueckensystem);
12:        brueckensystem.umschlagAnkunft(umschlag);
13:      }
14:      else
15:      {
16:        i++;
17:      }
18:    }
19:  }
20: }
```

Als weiteres Beispiel wird die Verwaltung der Resourcen aufgeführt, wie sie im AGV- oder im Brückensystem auftritt. Das AGV-System und das Brückensystem sind von der Funktionalität praktisch identisch. Sie verwalten zwei Listen mit freien bzw. beschäftigten Einheiten, und bekommen vom Gesamtsystem Aufträge erteilt, die sie an geeignete Einheiten weiterleiten. Im Unterschied zum AGV-System muss das Brückensystem aber zusätzlich überprüfen, ob eine freie Brücke zur Ausführung eines bestimmten Auftrages fähig ist, da eine Brücke einen festen Platz hat, und deshalb natürlich nur die Container auslädt, die sie erreichen kann. Ist keine Einheit zur Ausführung eines Auftrages frei, trägt das AGV- bzw. Brückensystem den Auftrag in eine Warteliste ein. Wenn sich eine Einheit freimeldet wird die Warteliste nach geeigneten Aufträgen durchsucht und die Einheit evtl. gleich wieder beschäftigt.

Die Methoden zur Auftragsvergabe und Freimeldung der Einheiten sind nachfolgend am Beispiel des AGVSystems wiedergegeben. Wenn bei der Auftragsvergabe (holAGV()) ein AGV frei ist (freeagvs.size()>0, Zeile 5) dann wird

diesem eine Nachricht gesendet, dass es zu dieser Brücke fahren soll (Zeile 9). Ist kein AGV frei wird die Brücke in die Warteschlange eingetragen (Zeile 13). Meldet sich ein AGV nach beendeter Arbeit wieder frei, wobei es sich mit seiner Id identifiziert, die in einer Hashtable abgelegt ist (Listing 2, Zeile 5), wird es wieder in die Liste der freien AGVs eingetragen (Zeile 6). Sind noch Brücken in der Warteschlage (bruecken.size()>0, Zeile 7) wird die erste Brücke ausgetragen, und der Auftrag erneut vergeben (holAGV(b), Zeile 11).

```
 1: public void holAGV(Bruecke b)
 2: {
 3:   synchronized(freeagvs)
 4:   {
 5:     if(freeagvs.size()>0)
 6:     {
 7:       DjAGV agv = (DjAGV)freeagvs.elementAt(0);
 8:       freeagvs.removeElement(agv);
 9:       agv.fahreZu(b);
10:     }
11:     else
12:     {
13:       bruecken.addElement(b);
14:     }
15:   }
16: }
```

```
 1: public void agvFree(java.lang.String agvid)
 2: {
 3:   synchronized(freeagvs)
 4:   {
 5:     DjAGV agv = (DjAGV)agvs.get(agvid);
 6:     freeagvs.addElement(agv);
 7:     if(bruecken.size()>0)
 8:     {
 9:       DjBruecke b = (DjBruecke)bruecken.elementAt(0);
10:       bruecken.removeElement(b);
11:       holAGV(b);
12:     }
13:   }
14: }
```

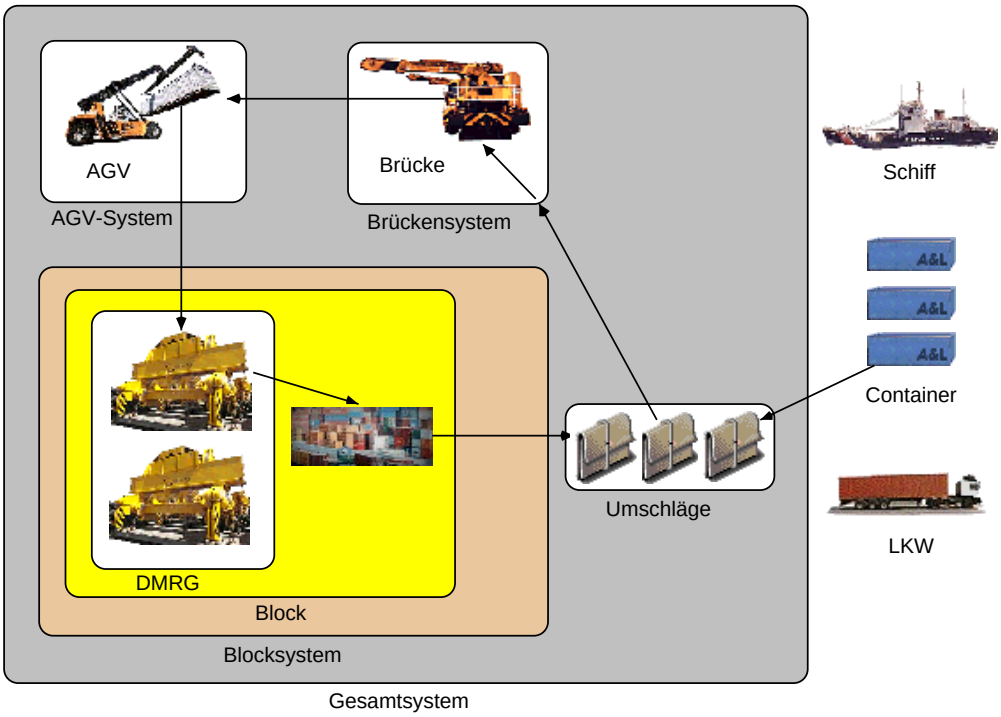
Migration

Eine wichtige Rolle bei der Steuerung des Containersystems spielen die Umschläge, die den Weg und die Bearbeitungsschritte jeden einzelnen Containers verwalten und steuern. Diese Objekte werden von nahezu allen Teilsystemen zu unterschiedlichen Zeitpunkten benötigt. Die Kommunikation zwischen Umschlag und Teilsystem tritt genau dann auf, wenn das Teilsystem den entspre-

chenden Container behandeln muss. Zu diesen Zeitpunkten ist die Kommunikation mit dem Umschlag intensiv und gleichzeitig zeitkritisch. Es besteht einerseits die Möglichkeit die Umschlagobjekte zentral zu halten und durch entfernte Kommunikation auf diese zuzugreifen, was allerdings zu einem hohen Kommunikationsaufkommen führen würde und die Infrastruktur unnötig belastet und um einige Größenordnungen langsamer ist, als lokale Kommunikation. Eine weitere Möglichkeit ist, diese Objekte zu replizieren und Kopien auf den jeweiligen Teilsystemen zu erstellen. Dadurch kann die zeitkritische Kommunikation lokal durchgeführt werden und das Übertragungsnetz wird entlastet. Dafür ist erhöhter Aufwand nötig, um die Konsistenz der einzelnen Kopien zu gewährleisten, auf die nicht nur lesend, sondern auch schreibend zugegriffen wird.

In Dejay besteht hierzu eine weitere Möglichkeit. Da ein starker Nachrichtenaustausch zwischen Teil-System und Umschlag genau am jeweiligen Aufenthaltsort des Containers auftritt, soll das Umschlagsobjekt seinen Ort dynamisch ändern. Es migriert parallel mit dem Container durch das System. Dazu wird der Umschlag in einem eigenen virtuellen Prozessor angelegt, der die Fähigkeit zur Migration zur Verfügung stellt. Die Abbildung 12.10 zeigt den Weg, den das Umschlagobjekt auf diese Weise durch das System nimmt. Es wird zunächst im Gesamtsystem erzeugt, migriert bei Ankunft eines Schiffes zu der Brücke, die den zugehörigen Container entladen soll, begleitet den Container mit dem AGV zum Blocksystem, wo es zur Steuerung des RMGs benötigt wird, und endet nach Auslieferung an einen LKW wieder im Gesamtsystem, in dem es für die langfristige Datenhaltung persistent gemacht werden kann.

Abbildung 12.10: Migrationsübersicht



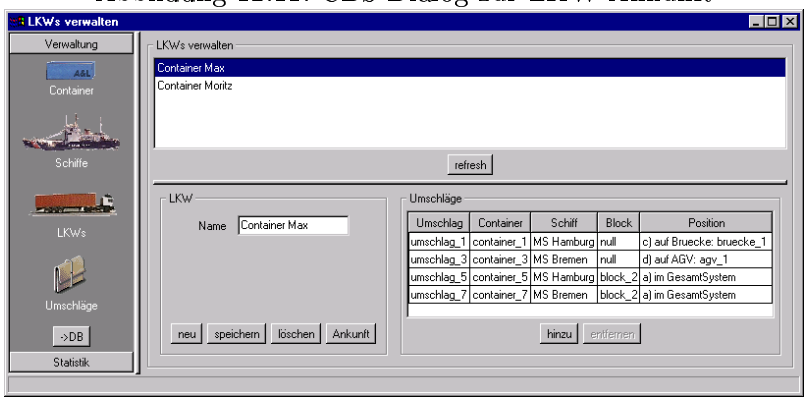
Events

Das Container Basis System (CBS) stellt die Schnittstelle für die Steuerung des Systems dar. Hier sollen aktuelle Daten angezeigt, neue Daten eingegeben und das System überwacht und gelenkt werden. Entscheident dabei ist die Aktualität der Daten. Sie sollen möglichst den tatsächlichen Zustand des Systems widerspiegeln. Auch zur Realisierung dieser Komponente standen mehrere Techniken zur Diskussion. Die klassische Kopplung solcher Systeme erfolgt über eine zentrale relationale Datenbank. Da das gesamte System aber sehr objektorientiert ausgelegt ist, bot sich eine objektorientierte Datenbank für diese Anbindung an. Das CBS greift dabei auf zentral in einer Datenbank gehaltene Objekte zu. Diese Objekte müssen mit den Objekten aus dem verteilten System abgeglichen werden und bieten daher nur eine zeitverzögerte Darstellung der tatsächlichen Situation. Das CBS greift dann auf die Objekte in der Datenbank transaktional zu.

In nicht-verteilten Systemen dagegen werden für solche Systeme häufig Events eingesetzt. Dafür registrieren sich Observer-Objekte als Interessenten für Ereignisse ein und werden vom System benachrichtigt, sobald solch ein Ereignis eintritt. Das beobachtete Objekt muss dafür den Observer nicht kennen.

In Dejay stehen diese Events ebenso wie in lokalen Systemen zur Verfügung und können völlig transparent eingesetzt werden. Der für dieses Projekt entwickelte Prototyp des CBS basiert daher auf diesem Ansatz. Ein Dialogobjekt trägt sich bei den dargestellten Objekten als Observer (s. [Gamma et al. 1995, p.293ff]) ein, und wird über Änderungen am Objekt benachrichtigt. Dadurch ist gewährleistet, dass der Benutzer nicht auf alten Daten operiert, und dass nur die Daten versendet werden, die für die Bedienung unbedingt notwendig sind. Das hierfür entworfene GUI ist in Abbildung 12.11 gezeigt.

Abbildung 12.11: CBS Dialog zur LKW-Ankunft



Das GUI kann so auf entfernten Dejay-Objekten arbeiten, wodurch einerseits die Synchronisation sichergestellt ist, und andererseits das GUI relativ unabhängig von Dejay-Konzepten entwickelt werden kann. Die Anbindung des GUI an das Dejay-Programm erfolgt dann über den Naming/Lookup-Service von Dejay.

Die Architektur des Systems ist so gewählt, dass die grafische Oberfläche sowohl mit der Datenbank als auch mit dem Gesamtsystem verbunden ist. Initiiert

der Benutzer Anwendungsfälle im GUI, werden diese Aktionen an das Gesamtsystem geleitet und von dort weiter an die verantwortlichen Teil-Systeme und Objekte delegiert und abgearbeitet. Das Gesamtsystem realisiert somit die Aktionsschnittstelle des Systems. Benötigt das GUI Daten für die Darstellung, wird die DB Schnittstelle für Anfragen benutzt. Möchte der Nutzer Daten ändern, so werden diese direkt vom GUI in den Modellobjekten modifiziert, die es von der DB erhalten hat.

Das Interface des Gesamtsystems zeigt noch einmal detailliert, welche Aktionen der Benutzer auslösen kann. Neben der Erzeugung von Schiff-, LKW-, Container- und Umschlagobjekten gibt es Methoden, um Schiffe und LKWs im System ankommen und wieder abfahren zu lassen.

```
 1: public interface IGesamtSystem
 2: {
 3:   public ISchiff    neuesSchiff(String id);
 4:   public ILKW       neuerLKW(String id);
 5:   public IContainer neuerContainer(String id);
 6:   public void       neuerUmschlag(IContainer container, ISchiff schiff);
 7:   public void       schiffAnkunft(ISchiff schiff);
 8:   public void       schiffAbfahrt(ISchiff schiff);
 9:   public void       lkwAnkunft(ILKW lkw);
10:   public void       lkwAbfahrt(ILKW lkw);
11:   public Datenbank  getDatenbank();
12: }
```

Der Namensdienst

Das GUI ist als eigenständige Applikation konzipiert und kann dadurch auf beliebigen Rechnern unabhängig vom Rest des Systems gestartet werden. Um eine Verknüpfung des GUIs mit dem System zu ermöglichen wurde der in Dejay integrierte Namensdienst verwendet. Die Verwendung des Namensdienstes soll am Beispiel der Methode GUI.main(), über die das GUI auf einem eigenen Rechner gestartet werden kann, gezeigt werden.

```
 1: public static void main(String[] args)
 2: {
 3:   try
 4:   {
 5:     Dejay.setDefaultProcessor(Dejay.getLocalProcessor());
 6:     IGesamtSystem gs;
 7:     if(args.length>0)
 8:     {
 9:       gs = (IGesamtSystem)Namespace.lookup(args[0]);
10:     }
11:     else
12:     {
13:       gs = (IGesamtSystem)Namespace.lookup ("//localhost:7042/GesamtSystem");
14:     }
```

```
15:     MainFrame mf = new MainFrame();
16:     mf.setGesamtSystem(gs);
17:     mf.show();
18:   }
19:   catch(Exception e)
20:   {
21:     System.out.println("Kein GesamtSystem gefunden!");
22:     e.printStackTrace();
23:   }
24: }
```

Beim Starten des GUIs kann die Adresse der TLS als Parameter übergeben werden und wird unter dieser Adresse (Zeile 9) gesucht. Wird kein Parameter angegeben, wird unter einer voreingestellten Adresse (Zeile 13) nach dem Gesamtsystem, dass die Schnittstelle zur TLS darstellt, gesucht. Danach wird das GUI-Hauptfenster (MainFrame) erzeugt, initialisiert und angezeigt. Das Gesamtsystem registriert sich während der Initialisierung mit einem Proxy beim Lookup-Service, wie im folgenden Listing dargestellt.

```
 1: public void init(...)
 2: {
 3:   ...
 4:
 5:   try
 6:   {
 7:     Namespace.bind("/GesamtSystem", Dejay.proxyFor(this));
 8:   }
 9:     catch(Exception e)
10:   {
11:     System.out.println("Konnte GesamtSystem nicht registrieren!");
12:     e.printStackTrace(); System.exit(0);
13:   }
14: }
```

12.4.2 Nebenläufigkeit

Nebenläufigkeit ist in einem System wie dem Vorliegenden unabdingbar. Die in der Realität auftretende Nebenläufigkeit konnte durch die Abbildung auf Komponenten, die durch das Konzept der virtuellen Prozessoren jeweils zueinander nebenläufig sind, gut in das Softwaresystem übertragen werden und konnte damit leichter als mit herkömmlichen Mitteln erreicht werden. Die eigentliche Herausforderung besteht aber darin, die Kommunikation so auszulegen, dass diese Nebenläufigkeit auch genutzt werden kann. In objektorientierten Systemen erfolgt die Kommunikation nämlich meist synchron, so dass bei einem Aufruf der Kontrollfluss erst wieder zur Verfügung steht, wenn der Aufruf mit einem Ergebnis zurückkehrt. Um auf Basis dieser synchronen Aufrufe die eigentlich nötigen asynchronen Aufrufe zu realisieren, muss in herkömmlichen Systemen wie CORBA oder RMI ein beträchtlicher Aufwand getrieben werden. Dazu müssen auf

Seite des Klienten oder des Servers zusätzliche Kontrollflüsse abgespalten werden, die einen synchronen Aufruf durchführen, auf das Ergebnis warten und sich mit dem Hauptkontrollfluss wieder synchronisieren, um diesem gegenüber eine Asynchronität zu simulieren. In Dejay dagegen ist ein einfacher Mechanismus für die Versendung asynchroner Nachrichten und die Behandlung der Ergebnisse integriert.

Als Beispiel ist hier eine Methode des Brückensystems aufgeführt, deren Aufgabe es ist, aus einer Ladeliste (der Liste der Umschläge) das Entladen der Container mit Hilfe einer Brücke und je eines AGV pro Container zu steuern. Die Methode überprüft, ob die Brücke am richtigen Kai steht und ordert dann für jeden Container einen AGV und beauftragt die Brücke den Container zu entladen. Dabei ist wichtig, dass diese Methode nicht blockiert und wartet bis einzelne Schritte abgearbeitet sind, sondern nebenläufig zu AGV und Brücke weiterarbeiten kann. Daher ist an dieser Stelle asynchrone Kommunikation von Nöten. In Dejay wird dies einfach durch die Angabe der Konstanten Dejay.ASYNC als zusätzlichen Parameter bei einem entfernten Aufruf realisiert.

```
1: public void beschaeftigeBruecke(Bruecke bruecke) {
2:   synchronized(umschlaege) {
3:     int kai = bruecke.getKai();
4:     DjUmschlag umschlag;
5:     for(int i=0; i<umschlaege.size(); i++) {
6:       umschlag = (DjUmschlag)umschlaege.elementAt(i);
7:       if(umschlag.getSchiff().getKai()==kai) {
8:         gs.holAGV(bruecke, Dejay.ASYNC);
9:         umschlag.moveTo((DjBruecke)bruecke);
10:        bruecke.ladeContainer(umschlag, Dejay.ASYNC);
11:        umschlaege.removeElement(umschlag);
12:      }
13:    }
14:  }
15:}
```

12.4.3 Persistenz

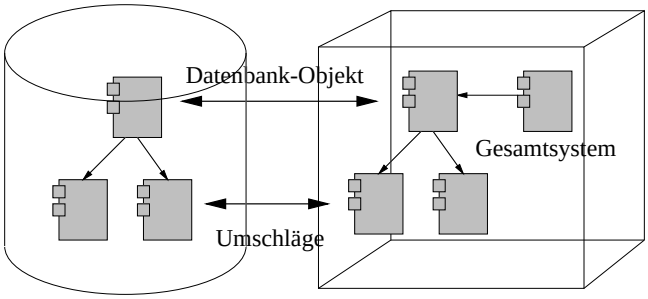
In Dejay wird die Persistenz als spezielle Art der Migration betrachtet, nämlich als Migration in einen persistenten Raum. Dabei werden, wie bei der Migration auch, virtuelle Prozessoren als Ganzes persistent gemacht. Referenzen auf die persistierten Objekte bleiben gültig, und persistente Objekte werden bei Bedarf (samt ihrem virtuellen Prozessor) reaktiviert. Allerdings muss die Granularität für Verteilung, Nebenläufigkeit und Persistenz identisch sein, was einen Einfluss auf die Modellierung in der Designphase haben kann. Andererseits ist die Entscheidung nach einer Aufteilung in Komponenten verhältnismäßig einfach, welche Komponenten persistent gemacht werden müssen.

In dem vorliegenden Beispiel sind auf der einen Seite Komponenten zu identifizieren, die lediglich aktionsausführenden Einheiten bilden und auf der anderen Seite passive Objekte, die nur zur Datenhaltung vorgesehen sind. Damit fällt

die Identifikation der zu persistierenden Objekte leicht und umfasst die Objekte Container, LKW, Schiff und Umschlag.

Referenzen auf diese Objekte werden (auch zu Abfragezwecken) in einem Datenbank-Objekt gesammelt. Dieses Datenbank-Objekt hat einen eigenen virtuellen Prozessor, und Container-, LKW- und Schiffsobjekte werden ebenfalls auf diesem "Datenbankprozessor" erzeugt (Abb. 12.12). Da die Umschläge einzeln durch das System migrieren (s.o.), haben sie jeweils einen eigenen virtuellen Prozessor, zum Persistieren des gesamten Datenbestandes ist es also notwendig sowohl den Datenbankprozessor mit den darin enthaltenen Container-, LKW-, und Schiffsobjekten zu persistieren, als auch die Prozessoren aller Umschläge, die vorher noch auf den Rechnerknoten des Datenbankprozessors migriert werden müssen.

Abbildung 12.12: Persistenz-Architektur



Dazu wird zuerst der (lokale) Datenbankprozessor persistiert (Zeile 3 im folgenden Listing), wobei id der Name der Datenbank ist (z.Zt. eine einfache Datei mit den serialisierten Objekten), und DB den Prozessor benennt, damit er später wieder geladen werden kann (s.u.). Danach werden der Reihe nach alle Umschläge migriert (Zeile 9) und persistiert (Zeile 10). Dabei ist zu beachten, dass eventuell GUI-Clients als Observer bei den Umschlägen registriert sind, die GUI-Clients aber nicht in der Datenbank gespeichert werden sollen. Deshalb werden diese Registrierungen gelöscht (Zeile 8) und nach dem Speichern wieder hergestellt (Zeilen 7 und 11).

```
 1: public void save(java.lang.String id)
 2: {
 3:   Dejay.getLocalProcessor().persist(id,"DB");
 4:   for(int i=0; i<umschlaege.size(); i++)
 5:   {
 6:     DjUmschlag u = (DjUmschlag)umschlaege.elementAt(i);
 7:     Vector v = u.getListeners();
 8:     u.clearListeners();
 9:     u.getProcessor().moveTo(Dejay.getLocalProcessor().getAddress());
10:     u.getProcessor().persist(id,u.getId());
11:     u.setListeners(v);
12:   }
13: }
```

Beim Programmstart wird die Datenbank durch das Gesamtsystem geladen. Dazu wird die nachfolgend aufgeführte statische Methode Datenbank.load-or-create() aufgerufen. Das Laden des Prozessors erfolgt in Zeile 5, wobei wieder Datenbank- und Prozessorname angegeben werden (id,"DB"). Danach wird das im Prozessor enthaltene Datenbankobjekt zugewiesen (Zeile 6). Treten bei diesen Aktionen Ausnahmen auf, so konnte die Datenbank nicht geladen werden. In dem Fall wird ein neuer Prozessor (Zeile 11), und auf diesem ein neues Datenbankobjekt erzeugt (Zeile 12). Dabei wird dem Datenbankobjekt der Name »Datenbank« zugewiesen, damit es später beim Laden wieder extrahiert werden kann (s. Zeile 6).

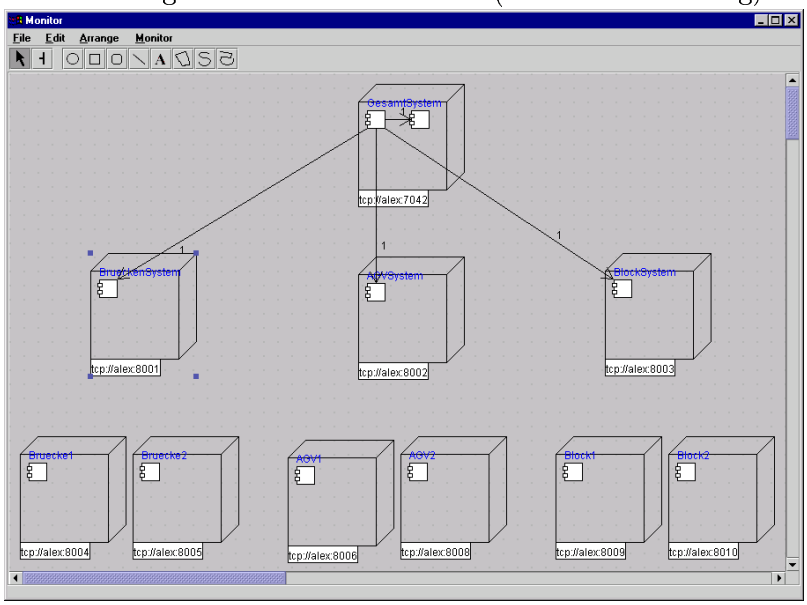
```
 1: public static Datenbank load-or-create(java.lang.String id)
 2: {
 3:   try
 4:   {
 5:     IProcessor dbproc = new DejayDb().retrieve(id,"DB");
 6:     db = (DjDatenbank)dbproc.getObjectByName("Datenbank");
 7:   }
 8:   catch(Exception e)
 9:   {
10:     DjProcessor p;
11:     p = new DjProcessor(Dejay.getLocalProcessor().getAddress());
12:     db = new DjDatenbank(id, p, "Datenbank");
13:   }
14:   return db;
15: }
```

12.5 Monitoring des Laufzeitverhaltens

Das laufende System konnte ohne weitere Änderungen durch den in Kapitel 8 beschriebenen Monitor dargestellt und dessen Verhalten beobachtet werden. Der Dejay-Monitor kann jederzeit zu jedem laufenden Dejay-Programm hinzugeschaltet werden, und zeigt für alle bekannten Systemknoten (Rechner) die virtuellen Prozessoren an. Der Inhalt der Komponenten kann durch Doppelklick abgefragt werden, wobei lediglich die Objekte angezeigt werden, die auch von außen entfernt referenziert werden. Darüber hinaus ist es möglich Komponenten zu migrieren, indem man sie per drag-n-drop auf einen anderen Knoten verschiebt.

Auch das dynamische Verhalten lässt sich überwachen. Abbildung 12.13 zeigt den Systemzustand zu Beginn der Anwendung, die Abbildung 12.14 stellt den Monitor während eines späteren Zeitpunktes der Ausführung der Containerhafen-Applikation dar. Wie auch in Abbildung 12.13 sind die Knoten der Übersichtlichkeit halber ähnlich wie im Design spezifizierten Komponentendiagramm (Abschnitt 12.3.2 Abb. 12.3) angeordnet. Alle entfernten Zugriffe werden durch Pfeile und Anzahl der Aufrufe gekennzeichnet. Natürlich werden auch neu erzeugte Komponenten und migrierende Komponenten gemäß ihrem neuen Aufenthaltsort automatisch dargestellt.

Abbildung 12.13: Laufzeit-Monitor (vor der Ausführung)



Zusätzlich zur dynamischen Darstellung des Verhaltens speichert der Monitor die Programmabläufe. Ablaufprotokolle können in Form von Sequenzdiagrammen angezeigt werden. In Abbildung 12.15 ist das generierte Sequenzdiagramm zu sehen, das dem Vorgang "Container-Entladen" entspricht, in dem ein Container aus einem Schiff entladen und über eine Brücke und ein AGV in einem Block transportiert wird.

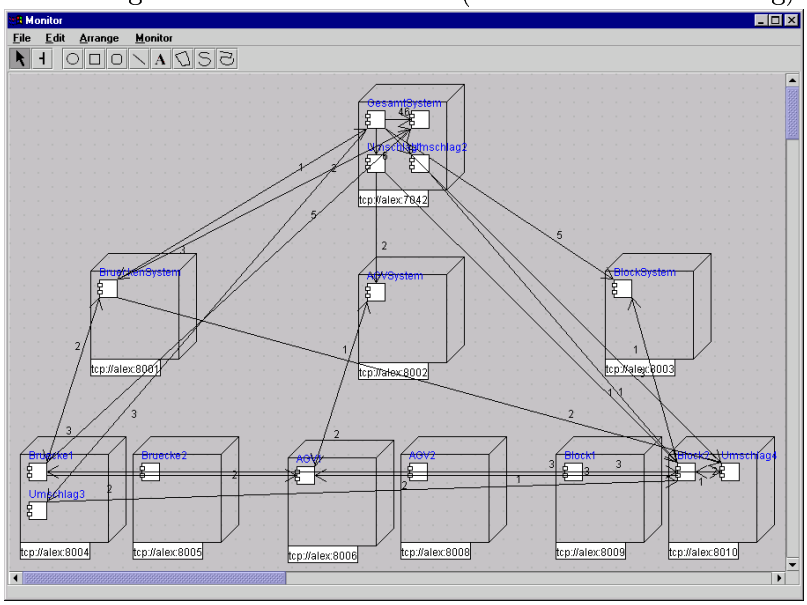
12.6 Ergebnis und Bewertung

Ziel dieser Studie war anhand eines größeren realitätsbezogenen Projektes festzustellen, inwieweit die in dieser Arbeit entwickelten Konzepte für die Realisierung geeignet sind.

Die Übertragung des Designs zur Implementation in Dejay war nach einer Modellierung vom Ansatz her einfach, wurde jedoch durch den noch relativ geringen Reifegrad Dejays und des zugehörigen Compilers hinsichtlich bestimmter Aspekte manchmal erschwert. Besonders störend bei der häufigen Anwendung wirkten sich neben den langen Compilierungszeiten die wenig aussagekräftigen Fehlermeldungen des Compilers aus, so dass viel Zeit in das Debugging investiert werden musste. Doch diese Mängel wurden zum großen Teil begleitend zu diesem Projekt behoben oder eine Verbesserung konzeptionell entwickelt. Insgesamt erhält der Programmierer ein Werkzeug, das ihm hinsichtlich Verteilung, Nebenläufigkeit und Persistenz eine sehr günstige Abstraktionsebene anbietet und die Umsetzung seiner Ideen vereinfacht.

Das durch Analyse und Design entwickelte Modell konnte durch die der Modellierung sehr naheliegenden Konzepte von Dejay als eins-zu-eins Bauplan für die Implementierung angewendet werden. Daher gab es bei der Umsetzung keine konzeptuellen Probleme, sondern lediglich solche, die auf den Reifegrad Dejays

Abbildung 12.14: Laufzeit-Monitor (während der Ausführung)

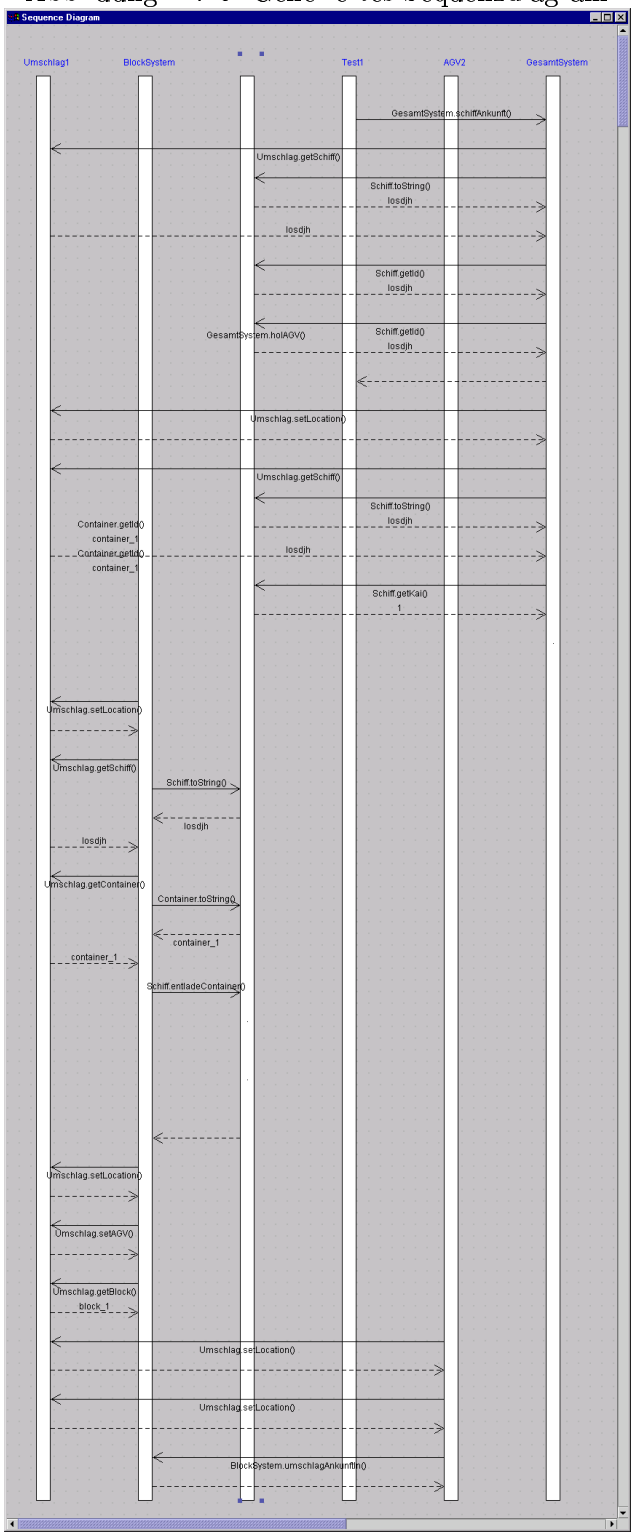


zurückzuführen waren. Durch die flexible Verteilung der Komponenten, je nach Ort der Installation, konnte der Aufwand für die entfernte Kommunikation stark begrenzt werden. Durch die Möglichkeit der Migration können die Kommunikationspfade sogar zur Laufzeit noch optimiert werden. Die Notwendigkeit der Nebenläufigkeit konnte gut durch die virtuellen Prozessoren abgebildet werden und die Ausnutzung der Nebenläufigkeit durch asynchrone Kommunikation ist sehr einfach möglich. Die Persistenz konnte durch die Möglichkeit der modularen Abspeicherung der virtuellen Prozessoren als Einheit leicht bewerkstelligt werden.

Der in den Vorarbeiten von sd&m festgestellte Bruch zwischen CBS und TLS, verursacht durch den Einsatz verschiedener Technologien (EJB vs. CORBA), konnte durch den Einsatz von Dejay beseitigt werden. Durch Dejay ist es möglich alle Objekte unabhängig von ihrer Position und Mobilität im System, einheitlich durch die Benutzungsschnittstelle des CBS anzusprechen und zu modifizieren, wobei durch das Konzept des virtuellen Prozessors die Konsistenz aller Daten auch bei gleichzeitigem Zugriff durch mehrere Benutzer des CBS und Logistikkomponenten der TLS immer sichergestellt bleibt.

Insgesamt lässt sich sagen, dass die in Dejay gesetzten Erwartungen in diesem Projekt erfüllt wurden und die Studie für alle Beteiligten ein Erfolg war. Insbesondere hat sich das Konzept als sehr erfolgreich erwiesen, auch wenn sich die konkrete Umsetzung noch verbessern lässt. Insbesondere kann die Integration von Verteilung, Nebenläufigkeit und Persistenz in dem Konzept des virtuellen Prozessors als erfolgreich bezeichnet werden. Alle drei Aspekte konnten in diesem Projekt mit diesem Konzept eingesetzt werden und stellten eine deutliche Vereinfachung im Vergleich zu herkömmlichen Techniken dar.

Abbildung 12.15: Generiertes Sequenzdiagramm



Kapitel 13

Zusammenfassung und Ausblick

13.1 Zusammenfassung

Ziel dieser Arbeit war, die Entwicklung verteilter Softwaresysteme zu vereinfachen. Ansatzpunkt hierfür ist, wichtige Aspekte der Programmierung verteilter Systeme, die heutzutage getrennt behandelt werden müssen, durch ein geeignetes Konzept zusammenzufassen und durch eine geeignete Abstraktion gemeinsam zu beschreiben. Die betrachteten Aspekte sind Verteilung, Nebenläufigkeit und Persistenz.

Zunächst wurde herausgestellt, dass diese Aspekte heutzutage mit weitgehend orthogonalen Mechanismen behandelt werden. Dafür wurden für jeden dieser Aspekte die wichtigsten, heute zur Verfügung stehenden Mechanismen klassifiziert und beschrieben. Eine Fokussierung auf die Sprache Java und die um sie entstandenen Techniken und Implementierungen spiegelt deren Bedeutung in der Praxis sowie in Wissenschaft und Forschung wider. Wegen ihrer besonderen Eignung für verteilte Systeme durch ihre Eigenschaften Plattformunabhängigkeit, dynamische Ladbarkeit und ihre für solche Umgebungen geeigneten Sicherheitsmechanismen, basieren heutzutage nahezu alle wichtigen neuen Impulse auf diesem Gebiet auf dieser Sprache oder sind eng mit ihr verknüpft.

Für den Aspekt der Verteilung wurden in Kapitel 2 drei zu unterscheidende Kommunikationsprinzipien, die unstrukturierte Datenkommunikation, die Kommunikation über Stellvertreter und die unmittelbare Objektkommunikation herausgearbeitet und an konkreten Mechanismen untersucht.

Der Aspekt der Nebenläufigkeit (Kapitel 3) konnte in die drei Bereiche Nebenläufigkeit durch Hardware oder Prozesse, nebenläufige Kontrollflüsse und objekt- bzw. komponentenbasierte Nebenläufigkeit unterteilt und diese Unterteilung anhand von konkreten Mechanismen untermauert werden.

Für die Persistenz wurden in Kapitel 4 die unstrukturierte Persistenz, die datenstrukturbasierte Persistenz und die objektorientierte Persistenz identifiziert und ebenfalls durch konkrete Mechanismen vorgestellt.

Wichtiges Ergebnis dieser Betrachtung ist, dass in typischen verteilten Systemen alle drei Aspekte berücksichtigt werden müssen. Dabei kann für jeden der drei Aspekte ein konkreter Mechanismus ausgewählt werden, unabhängig von der Entscheidung, welche Technik für die jeweils anderen Aspekte gewählt

wird. Daraus wird eine weitreichend orthogonale Behandlung dieser Aspekte mit heute zur Verfügung stehender Techniken abgeleitet, die den Programmierer zwingt, alle drei Aspekte zu beherrschen und zu behandeln und so zu einer enormen Komplexität bei der Entwicklung verteilter Systeme führt.

Im zweiten Teil der Arbeit wurde hinterfragt, ob diese Orthogonalität zwingend ist, oder ob sich diese Aspekte nicht durch eine geeignete Abstraktion zusammenfassen und dadurch vereinfachen lassen. Die Beobachtung auf anderen Gebieten der Softwareentwicklung legen die Vermutung nahe, dass bei einer erfolgreichen Zusammenfassung eine beträchtliche Vereinfachung zu erreichen ist. Als Beispiel wurde der Paradigmenwechsel von der prozeduralen zur objektorientierten Programmierung angeführt, der sich durch die Zusammenfassung der Aspekte Datenhaltung, Funktionsbeschreibung und Struktur, die in der prozeduralen Programmierung orthogonal behandelt werden müssen, zum Konzept der Klasse charakterisieren lässt (siehe Abschnitt 1.2). Die Vorteile dieser Zusammenfassung waren ausreichend, so dass man behaupten kann, der Paradigmenwechsel habe sich, trotz der Bindung an Altsysteme, weitgehend vollständig vollzogen.

Basierend auf Erkenntnissen anderer Forschungsvorhaben, insbesondere der Entwicklung von Verteilungsmechanismen mit unmittelbarer Objektkommunikation (konkretisiert am Beispiel Emerald) und der komponentenbasierten Nebenläufigkeit (entwickelt als konzeptionelle Erweiterung der Sprache Eiffel) sowie aus der Diskussion aus der Entwicklung des RMI (der in Java standardmäßig eingesetzten entfernten Kommunikation durch Stellvertreter), dargestellt in Kapitel 5, wurde ein eigener Ansatz entwickelt. Grundlegender Gedanke dabei ist eine neu eingeführte Unterscheidung zwischen lokalen und entfernten Referenzen und eine Unterteilung von Objekten zu in sich geschlossenen Gruppen, die von jeweils von einer Ausführungseinheit verwaltet werden. Diese Ausführungseinheit, virtueller Prozessor genannt, sorgt für die Konsistenz der lokalen ebenso wie der entfernten Referenzen auf Objekte der verwalteten Gruppe. Diese Konsistenz kann auch unter Nebenläufigkeit, Migration und Persistenz gewährleistet werden. Dieses Konzept wurde in Kapitel 6 entwickelt und seine Realisierbarkeit wurde in der Umsetzung in einer auf Java basierenden verteilten Programmiersprache namens Dejay gezeigt, die in Kapitel 7 vorgestellt wurde. Einige Aspekte der Implementierung wurden in Kapitel 9 dargelegt. Eine Abgrenzung zu naheliegenden aktuellen Forschungsarbeiten fand in Kapitel 10 Ausdruck.

Im dritten Teil schließlich wurde versucht, den Nachweis zu führen, dass ein geeignetes Konzept gefunden wurde. Dafür wurden in Kapitel 11 konkrete Beispiele für jeden der Aspekte isoliert vorgestellt, bei denen jeweils das Konzept des virtuellen Prozessors zur Implementation erfolgreich verwendet wurde. Anschließend wurde in Kapitel 12 ein größeres Beispiel aus dem Bereich der industriellen Praxis vorgestellt, bei dem das erarbeitete Konzept der virtuellen Prozessoren für alle drei Aspekte gleichzeitig herangezogen werden konnte.

Zusätzlich zu den Vorteilen der Zusammenfassung der drei Aspekte Verteilung, Nebenläufigkeit und Persistenz zu einem Konzept konnten einige weitere Vorteile herausgearbeitet werden. Bei einer getrennten Behandlung dieser Aspekte ist eine Migration nur kaum möglich und praktische Umsetzungen wie durch Voyager weisen deutliche Nachteile auf, die in Abschnitt 2.2.3 aufgezeigt

wurden. Durch die Zusammenfassung von in sich konsistenten Objektgruppen und die Sicherung des konsistenten Zugriffs zwischen diesen war es möglich, einen einfachen Mechanismus für die Migration zu entwickeln, dargestellt in Abschnitt 7.4. Die Modellierung verteilter Systeme mit Modellierungssprachen wie UML, in denen die Zusammenfassung zu Gruppen in Form von Komponenten vorgesehen ist, konnte durch die Entwicklung einer direkt darauf abbildbaren Entsprechung im Code verbessert werden. Durch diese Entsprechung war es auch möglich, laufende Systeme wieder auf die Modelle abzubilden und diese als interaktive Monitore auf das verteilte System zu verwenden, was in Kapitel 8 beschrieben wurde.

Durch die Offenlegung des Sourcecodes [Dejay 2000], die Einrichtung eines Webservers zu diesem Projekt und durch die Veröffentlichung in einem Buch [Boger 1999] sind die entwickelten Konzepte allgemein zugänglich gemacht worden, so dass sie als Grundlage für Weiterentwicklungen dienen können.

13.2 Ausblick

Die in dieser Arbeit präsentierten Konzepte bieten noch Potential für weitere Forschungs- und Entwicklungsarbeiten. Im folgenden sollen einige Ideen und Anregungen für anschließende Arbeiten gegeben werden. Einige sind bereits konkret in Planung und werden zur Zeit oder in naher Zukunft in der Arbeitsgruppe Verteilte Systeme (VSYS) umgesetzt.

Die für die Implementierung des Compilers verwendete Technologie war für den Nachweis der Machbarkeit sehr geeignet, erweist sich allerdings für einen praktischen Einsatz als zu langsam. Die in der Umsetzung verwendete Metasprache OpenJava erwies sich zwar als eine gute Grundlage, einen Prototyp zu entwickeln, hat aber den großen Nachteil ein schlechtes Laufzeitverhalten an den Tag zu legen, insbesondere gilt dies für größere Dateien oder Projekte, so dass das Compilieren eines Projektes wie dem in Kapitel 12 beschriebenen im Bereich von zweistelligen Minuten liegt. Es handelt sich hierbei aber nicht um ein konzeptionelles Problem und lässt sich durch die Verwendung von geeigneten Parsern und Compilern stark beschleunigen. Gleichzeitig lassen sich dabei die Fehlermeldungen verbessern und durch Angabe der Zeilennummer aus dem Originalcode deren Auffindung und Behebung deutlich verbessern.

Die Abbildung durch den Compiler auf die Middleware Voyager erwies sich als ausgesprochen erfolgreich. Derzeit ist, wie auch im ersten Teil dargestellt, keine andere Middleware verfügbar, die Voyager für diese Zwecke in Leistungsumfang, Geschwindigkeit und Stabilität übertreffen könnte. Doch aus Kompatibilitätsgründen wäre sicherlich eine Abbildung unmittelbar auf RMI oder CORBA interessant. Die auf RMI basierenden Umsetzungen aus den Projekten FarGo und Doorastha weisen auf die Umsetzbarkeit auf Basis von RMI hin. Allerdings bietet Voyager auch die Möglichkeit der Integration verschiedener Middleware wie CORBA, RMI oder sogar DCOM. Dadurch müsste sich ein transparenter Zugriff auf Dejay-Komponenten auch über andere Middleware realisieren lassen.

Weiterhin ist zu dem frei verfügbaren Voyager-ORB eine kommerzielle Transaktionsunterstützung erhältlich, auf die aber aus finanziellen Gründen bisher

nicht zugegriffen werden konnte. Eine Integration dieser Technologie mit Dejay könnte allerdings einen Einsatz in unternehmenskritischen Bereichen möglich machen, so dass eine Untersuchung dieser Möglichkeit als vielversprechend eingeschätzt werden kann.

Der Monitormechanismus bietet Grundlage für die Weiterentwicklung zu einem Konfigurationsmanager oder Analysesystem. Die Informationen könnten durch Metriken ausgewertet werden, auf deren Grundlage Verbesserungsvorschläge für die Verteilung der Komponenten angeboten oder automatisch umgesetzt werden könnten. Ansatzpunkte hierfür lassen sich im Projekt Pangaea finden.

Eine interessante Erweiterung des Migrationsmechanismus könnte in der Entwicklung eines Replikationsmechanismus liegen. Wichtige Mechanismen wie das Kopieren ganzer Komponenten, das vollständige Mitprotokollieren aller in eine Komponente eingehenden Aufrufe, wie durch den Monitor gezeigt, und die Reaktion auf das Nichtvorhandensein eines Objektes durch entsprechende Ausnahmen wie bei der Migration, sind bereits vorhanden. Durch ihre Kombination ließe sich ein Schattenobjekt einrichten, das als Kopie einer Komponente angelegt und anschließend als Monitor einer einzelnen Komponente die identischen Methodenaufrufe ausführen könnte. Wenn das Original durch das Versagen eines Knotens unerreichbar wird und das System eine Ausnahme zurückgibt, könnten die Proxies automatisch auf die Benutzung des Schattenobjektes umschalten. Dies entspricht in etwa dem Mechanismus der Weiterleitung von Aufrufen auf migrierte Komponenten, allerdings wird dabei der neue Standort über die Ausnahmenachricht mitgeteilt. Für den Fall der Replikation müsste die Referenz auf das Schattenobjekt bereits im Proxy bekannt sein, was technisch leicht möglich ist, wie der Aktivierungsmechanismus der Persistenz zeigt. Insgesamt ließe sich damit eine Hochverfügbarkeitsmöglichkeit für Dejay-Komponenten erreichen, die, bis auf eine Initialisierung, vollkommen transparent gestaltet werden könnte.

Bei der Implementierung herkömmlicher Multitier-Architekturen ist die Aufteilung der Client- und Serverkomponenten auf die verschiedenen Rechner statisch. Soll das System neu konfiguriert werden, muss es erst angehalten und neu gestartet werden. Eine effizientere Variante wäre, die Komponenten einer verteilten Anwendung dynamisch zur Laufzeit, je nach Auslastung der beteiligten Rechner, zu verteilen. Diese Lastbalancierung könnte, aufbauend auf den in Dejay vorhandenen Mechanismen, automatisch erfolgen.

In [Rousselle 1998] wird der Prototyp eines solchen Lastbalancierung-Systems mit Namen *Sojourner* beschrieben. Es benutzt, ebenso wie Dejay, Voyager als Infrastruktur für entfernte Aufrufe und die Migration von Objekten. Im Sojourner-System existieren zwei Grenzwerte, die bestimmen, ob ein Rechner stark oder schwach ausgelastet ist. Der "nuisance threshold" ist der obere Grenzwert. Eine längere Auslastung über diesem Wert kennzeichnet einen stark ausgelasteten Rechner. Der untere Grenwert ist der "adaquate utilization threshold". Liegt die Auslastung unter diesem Wert, wird der Rechner nur wenig beansprucht und kann weitere Prozesse bearbeiten.

In einer Sojourner-Umgebung gibt es einen Master-Knoten, der alle neu erzeugten Server-Objekte auf die beteiligten Rechner verteilt. Ist die Last nied-

rig, wird der Rechner als potentieller Host für Gast-Objekte beim Master-Knoten registriert. Ist die Last dagegen hoch, werden die vorhandenen Gast-Objekte reihum an die wenig ausgelasteten Rechner gesandt. Ist kein Rechner mit niedriger Auslastung beim Master-Knoten registriert, werden die Objekte zum Master-Knoten selbst gesandt, der diese wieder verteilt, sobald Rechner mit niedriger Last zur Verfügung stehen.

In Ergänzung zu der Messung der Last bietet der Dejay-Monitor die Möglichkeit, die Kommunikationsaktivität zwischen zwei virtuellen Prozessoren zu messen. Zusätzliche Komponenten könnten, wie im Sojourner-System, die Last der beteiligten Rechner messen und dem Monitor melden. Der Dejay-Monitor könnte schließlich virtuelle Prozessoren geeignet verschieben. Dies ist jetzt schon manuell über das GUI des Monitors oder mittels der API möglich. Eine automatische Verteilung könnte neben der Last der einzelnen Rechner auch die Höhe der Kommunikation zwischen den Prozessoren berücksichtigen und somit versuchen die Anzahl der entfernten Aufrufe zu minimieren.

Als besonders vielversprechend kann die Codegenerierung aus UML eingeschätzt werden. Wenn es gelingt, die Abbildung von UML auf Dejay für die Aspekte Verteilung und Nebenläufigkeit zu automatisieren, könnte somit eine automatische Generierung verteilter Systeme aus ihrer Spezifikation in UML erreicht werden. In bisherigen UML-Entwicklungsumgebungen ist meist lediglich die Erzeugung von Codeskeletten möglich, die die Aspekte Nebenläufigkeit oder Verteilung in keiner Weise auszudrücken vermögen. Dass dieser Automatisierungsschritt möglich ist, legen die Erfahrungen aus dem HHLA-Projekt und neueren Experimenten zur Umsetzung von UML in ausführbare Darstellungen, die in [Heyden 2000] und [Kanzlers 2000] beschrieben sind, nahe und wird in bereits geplanten Anschlussprojekten genauer untersucht werden.

Doch bereits in seiner bisherigen Form hat sich das Konzept der virtuellen Prozessoren und die Sprache Dejay als sehr wertvoll erwiesen und kann in weiteren praktischen Anwendungen eingesetzt werden. Eine mögliche solche Anwendung ist die Erweiterung der UML-Entwicklungsumgebung ArgoUML [ArgoUML 2000] um die Möglichkeit der kooperativen gleichzeitigen Nutzung in einer Multi-User-Version. ArgoUML weist eine Model-View-Controller-Architektur auf. Insbesondere die Modell-Schicht ist sauber und klar von den anderen trennbar. Für eine Multi-User-Anwendung müsste genau diese Schicht einmal vorhanden sein, aber von mehreren anderen Knoten aus von getrennten Benutzeroberflächen, die die View- und Controller-Schichten enthalten, erreichbar sein. Dafür müssten die lokalen Referenzen vom Controller zum Modell identifiziert und durch entfernte Referenzen ersetzt werden – ein Schritt, der relativ mechanisch durchführbar ist. Dadurch könnten die Oberfläche und das Modell auf unterschiedlichen Knoten eingerichtet werden. Da Änderungen im Modell durch Events wieder bis zum View kommuniziert werden und durchaus mehrere Listener sich für diese Events anmelden können, wäre es dann möglich, das gleiche Modell von mehreren Oberflächen aus zu bearbeiten, wodurch sich ein kooperatives gemeinsames Arbeiten ergeben würde. Auch für dieses Projekt bestehen bereits konkrete Pläne.

Abbildungsverzeichnis

Literaturverzeichnis

[Achauer 1993] Bruno Achauer. The DOWL Distributed Object-Oriented Language. *Communications of the ACM* **36**(9) (September 1993).

[ArgoUML 2000] ArgoUML. ArgoUML-Homepage (2000). www.argouml.org.

[Atkinson et al. 1996] M. P. Atkinson, L. Daynes, M.J. Jordan, T. Printezip und S. Spence. An Orthogonally Persistent Java. *ACM SIGMOD Records* **25**(4) (Dezember 1996).

[Atkinson und Jordan 1998] M. P. Atkinson und M.J. Jordan. Orthogonal Persistence for Java - A Mid-term Report. In „3rd Int. Workshop on Persistence and Java, Tiburon, CA“ (September 1998).

[Baier 1999] Tobias Baier. „Programmierbeispiele in Dejay und Evaluation des Dejay-Systems, Studienarbeit Universität Hamburg“. Universität Hamburg (1999).

[Black et al. 1986] A. Black, E. J. Hutchinson, H. Levy und L. Carter. Object Structure in the Emerald System. In „Proceedings of Conference on Object Oriented Programming Systems“ (Oktober 1986).

[Black et al. 1987] A. Black, E. J. Hutchinson, E. Jul, H. Levy und L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions Software Engineering* **13**(1) (1987).

[Boger et al. 1999a] Marko Boger, Frank Griffel und Winfried Lamersdorf. Dejay: Unifying Concurrency and Distribution to Achieve a Distributed Java. *Integrated Computer-Aided Engineering (ICAE)* (1999).

[Boger et al. 1999b] Marko Boger, Frank Wienberg und Winfried Lamersdorf. Dejay: Concepts for a Distributed Java. In „Proceedings of Distributed Computing on the Web (DCW´99), June 99, Rostock“ (1999).

[Boger et al. 1999c] Marko Boger, Frank Wienberg und Winfried Lamersdorf. Dejay: Unifying Concurrency and Distribution to Achieve a Distributed Java. In „Proceedings of Technology of Object-Oriented Languages and Systems TOOLS Europe 99, Nancy“ (1999).

[Boger et al. 2000] MARKO BOGER, FRANK WIENBERG UND TOBY BAIER. eXtreme Modeling. In „Proceedings of Extreme Programming 2000, Italy“ (2000).

[Boger und Gellersen 1996] MARKO BOGER UND HANS-WERNER GELLERSEN. On Models in Object-Oriented Methods - Critique and a new Approach to Reversibility. In „Proceedings of Technology of Object-Oriented Languages and Systems TOOLS 19, Paris“ (1996).

[Boger 1998] MARKO BOGER. Migrating Objects in Electronic Commerce Applications. In „Proceedings of Trends in Distributed Systems for Electronic Commerce“ (1998).

[Boger 1999] MARKO BOGER. „Java in verteilten Systemem. Nebenläufigkeit, Verteilung und Persistenz“. dpunkt.verlag (1999).

[Brandt und Madsen 1993] S. BRANDT UND O.L. MADSEN. Object-Oriented Distributed Programming in BETA. In „Proceedings of Object-Based Distributed Programming, ECOOP'93 Workshop, Kaiserslautern, Germany, Lecture Notes in Computer Science, Vol. 791, Springer-Verlag“ (1993).

[Braubach und Pokahr 1999] LARS BRAUBACH UND ALEXANDER POKAHR. „Design und Implementierung verteilter Systeme mit UML und Dejay am Beispiel des Containerhafens Altenwerder, Studienarbeit Universität Hamburg“. Universität Hamburg (1999).

[Briot et al. 1997] J. BRIOT, R. GUERRAOUI UND K-P. LÖHR. Concurrency, Distribution and Parallelism in Object-Oriented Programming (Dezember 1997). Technical Report B-97-14, FU Berlin, FB Mathematik und Informatik.

[Caromel 1989] DENIS CAROMEL. Service, Asynchrony and Wait-by-Necessity. *Journal of Object-Oriented Programming* **2**(4) (November 1989).

[Carriero und Gelernter 1989] NICHOLAS CARRIERO UND DAVID GELERNTER. Linda in Context. *Communications of the ACM* **32**(4) (April 1989).

[Carriero und Gelernter 1990] NICHOLAS CARRIERO UND DAVID GELERNTER. „How to write Parallel Programs“. MIT Press (1990).

[Codd 1970] E. F. CODD. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM* **13**(6) (Juni 1970).

[Dahm 2000] MARKUS DAHM. „The Doorastha System, Technical Report B-1-2000, FU Berlin“. FU Berlin (2000).

[Dejay 2000] DEJAY. Dejay Homepage (2000). www.dejay.org.

[Downing 1998] TROY BRIAN DOWNING. „Java RMI: Remote Method Invocation“. Prentice Hall (1998).

[Fragemann 2000] PER FRAGEMANN. „Ein Gruppierungsmechanismus und ein Namensdienst für Dejay, Studienarbeit Universität Hamburg“. Universität Hamburg (2000).

[Gamma et al. 1995] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON UND JOHN VLISSIDES. „Design Patterns. Elements of Reusable Object-Oriented Software". Addison-Wesley (1995).

[Gelernter 1985] D. GELERNTER. Generativ Communication in Linda. *ACM Transactions Programming Languages and Systems* **7**(1) (1985).

[Gosling et al. 1996] JAMES GOSLING, BILL JOY UND GUY STEEL. „The Java Language Specification". Addison-Wesley (1996).

[Griffel 1998] FRANK GRIFFEL. „Componentware. Konzepte und Techniken eines Softwareparadigmas". dpunkt.verlag (1998).

[Hamilton et al. 1997] GRAHAM HAMILTON, RICH CATTELL UND MAYDENE FISHER. „JDBC Database Access with Java". Addison-Wesley (1997).

[Heyden 2000] OLIVER HEYDEN. „Ausführung von UML-Zustandsdiagrammen durch Referenznetze, Studienarbeit Universität Hamburg". Universität Hamburg (2000).

[Holder et al. 1999] OPHIR HOLDER, ISRAEL BEN-SHAUL UND HOVAV GAZIT. „Dynamic Layout of Distributed Applications in FarGo". Israel Institute of Technology, Haifa (1999).

[Hutchinson 1987] NORMAN C. HUTCHINSON. „Emerald: An Object-Based Language for Distributed Programming". Dissertation, Department of Computer Science, University of Washington (Januar 1987).

[IBM 2000] IBM. JinSight (2000). http://www.research.ibm.com/jinsight/.

[Inprise 1999] INPRISE. Inprise (1999). http://www.inprise.com/.

[Jacobson et al. 1999] IVAR JACOBSON, GRADY BOOCH UND JAMES RUMBAUGH. „The Unified Software Development Process". Addison-Wesley (1999).

[Jul et al. 1987] E. JUL, H. LEVY, N. HUTCHINSON UND A. BLACK. Fine-grained mobility in the Emerald system. In „Proc. of the Eleventh ACM Symposium on Operating System Principles". ACM (November 1987).

[Jul 1989] ERIC JUL. „Object Mobility in a Distributed Object-Oriented System". Dissertation, University of Washington (1989).

[Kanzlers 2000] ANDREAS KANZLERS. „Ausführung von UML-Kollaborationsdiagrammen durch Referenznetze, Studienarbeit Universität Hamburg". Universität Hamburg (2000).

[Kredel und Yoshida 1999] HEINZ KREDEL UND AKITOSHI YOSHIDA. „Threads- und Netzwerk-Programmierung mit Java". dpunkt.verlag (1999).

[Lea 1997] DOUG LEA. „Concurrent Programming in Java: Design Principles and Patterns". Addison-Wesley (1997).

[Maffeis 1997] SILVANO MAFFEIS. iBus- The Java Intranet Software Bus. In „International Conference on Software Engineering (ICSE'97)“ (Februar 1997).

[Magee und Kramer 1999] JEFF MAGEE UND JEFF KRAMER. „Concurrency - State Models and Java Programs“. John Wiley & Sons (1999).

[Merz et al. 1998a] M. MERZ, F. GRIFFEL, M. BOGER, H. WEINREICH UND W. LAMERSDORF. Electronic Contracting with COSMOS - How to Establish, Negotiate and Execute Electronic Contracts on the Internet. In „Enterprice Distributed Objects Computing Workshop (EDOC'98), San Diego“ (1998).

[Merz et al. 1998b] M. MERZ, F. GRIFFEL, T. TU, S. MÜLLER-WILKEN, H. WEINREICH, M. BOGER UND W. LAMERSDORF. Supporting Electronic Commerce Transactions with Contracting Services. *International Journal on Cooperative Information Systems, Vol. 7, No. 4* **7**(4) (1998).

[Merz et al. 1999] M. MERZ, F. GRIFFEL, M. BOGER, H. WEINREICH UND W. LAMERSDORF. Electronic Contracting im Internet. In „GI/ITG-Konferenz 'Kommunikation in Verteilten Systemen' (KIVS'99), Informatik-Aktuell“ (1999).

[Meyer 1990] BERTRAND MEYER. Sequential and Concurrent Object-Oriented Programming. In „Technology of Object-Oriented Languages and Systems 91, Paris“ (Juni 1990).

[Meyer 1997] BERTRAND MEYER. „Object-Oriented Software Construction“. Prentice Hall, Zweite Auflage (1997).

[Neimeyer 1998] PAT NEIMEYER (Herausgeber). „Core Java Networking“. Prentice Hall (1998).

[Nelson 1998] JEFF NELSON (Herausgeber). „Programming Mobile Objects with Java“. John Wiley & Sons (1998).

[Oaks und Wong 1997] SCOTT OAKS UND HENRY WONG. „Java Threads“. O'Reilly, Zweite Auflage (1997).

[ObjectDesign 1998] OBJECTDESIGN. ObjectStore (1998). www.odi.com.

[ObjectSpace 1998] OBJECTSPACE. Voyager (1998). www.objectspace.com.

[Peitgen et al. 1992] HEINZ-OTTO PEITGEN, HARTMUT JÜRGENS UND DIETMAR SAUPE. „Bausteine des Chaos. Fraktale“. Springer-Verlag (1992).

[Philippsen und Zenger 1996] MICHAEL PHILIPPSEN UND MATTHIAS ZENGER. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience* **9**(11) (November 1996).

[Poppendiek 1999] NILS POPPENDIEK. „Ein komponentenorientierter Persistenzmechanismus, Studienarbeit Universität Hamburg“. Universität Hamburg (1999).

[Raap 1999] JAN RAAP. „Virtuelle Prozessoren - ein Migrationskonzept für Gruppen von Objekten in Dejay, Diplomarbeit Universität Hamburg“. Universität Hamburg (1999).

[Redlich 1999] JENS-PETER REDLICH. „CORBA 3.0, eine praxisorientierte Einführung“. Addison-Wesley Longman, Zweite Auflage (1999).

[Reese 1997] GEORGE REESE. „Database Programming with JDBC and Java“. O'Reilly (1997).

[Robbins 1999] JASON ELLIOT ROBBINS. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. In „Construction of Software Engineering Tools (CoSET'99)“. University of South Australia (Juni 1999).

[Rousselle 1998] P. ROUSSELLE. Dynamic Distributed Systems in Java, Using agent technology to build flexible systems. *Dr. Dobb's Journal, www.ddj.com/articles/1998/9804/9804i* (April 1998).

[Schaffert 1986] CRAIG SCHAFFERT. An Introduction to Trellis/OWL. In „OOPSLA '86 Proceedings, Portland“. SIGPLAN Notices 21 (11) (September 1986).

[Scheurer 2000] MICHAEL SCHEURER. „Laufzeitmonitoring und Konfigurationsmanagement von Komponenten, Studienarbeit Universität Hamburg“. Universität Hamburg (2000).

[Siegel 1996] JON SIEGEL. „CORBA Fundamentals and Programming“. John Wiley & Sons (1996).

[Softwired 1998] SOFTWIRED. „Programming iBus Applications“. Softwired AG, Zürich, www.softwired.ch, version 0.5 Auflage (1998).

[Spiegel 1999a] ANDRE SPIEGEL. „Automatische Verteilung in Pangaea“. FU Berlin (1999). Technical Report, FU Berlin.

[Spiegel 1999b] ANDRE SPIEGEL. „Object Graph Analysis, Technical Report B-99-11, Preliminary Version, FU Berlin“. FU Berlin (1999).

[Stevens und Wright 1994] W. RICHARD STEVENS UND GARY R. WRIGHT. „TCP/IP illustrated“. Addison-Wesley (1994).

[Sturm 2000] THORSTEN STURM. „Entwicklung von Sprachkonzepten zur Vereinheitlichung von Nebenläufigkeit und Verteilung in Dejay, Diplomarbeit Universität Hamburg“. Universität Hamburg (2000).

[Sun 1999] SUN. Javasoft Homepage (1999). www.javasoft.com/nav/whatis/lightweight.html.

[Tatsubori 1997] M. TATSUBORI. OpenJava (1997). www.softlab.is.tsukuba.ac.jp/ mich/openjava/.

[Vanter 2000] Michael L. Van De Vanter. Displaying and Editing Source Code in Software Engineering Environments. In „Second International Symposium on Constructing Software Engineering Tools (CoSET'2000), Limerick, Ireland, June 2000". University of Limerick, Ireland (Juni 2000).

[Vitek und Tschudin 1997] Jan Vitek und Christian Tschudin (Herausgeber). „Mobile Object Systems - Towards the Programmable Internet. Second International Workshop, MOS´96". Nummer 1222 in LNCS. Springer-Verlag (1997).

[Waldo et al. 1994] Jim Waldo, Geoff Wyant, Ann Wollrath und Sam Kendall. A Note on Distributed Computing, Technical Report (1994). www.sunlabs.com/technical-reports/1994/abstract-29.html.

[Wienberg et al. 1999] Axel Wienberg, Florian Matthes und Marko Boger. Modeling Dynamic Software Components in UML. In B. Rumpe R. France (Herausgeber), „UML'99 - The Unified Modeling Language. Proceedings of the Second International Conference, Fort Collins, Colorado, USA", Band 1723 aus „Lecture Notes in Computer Science", Seiten 204–219. Springer-Verlag (10 1999).

[Wittmann und Zitterbart 1999] Ralph Wittmann und Martina Zitterbart. „Multicast - Protokolle, Programmierung, Anwendung". dpunkt.verlag (1999).

Erklärung

Hiermit versichere ich, die vorliegende Dissertation ausschließlich unter Verwendung der angegebenen Quellen und Hilfsmittel selbstständig angefertigt zu haben.

Hamburg, den 18. Oktober 2000

Marko Boger