Architektur zur Komponentenverwaltung für JavaBeans

Diplomarbeit
von
Karola Krönert
und
Ulrich Dallmann

Fachbereich Informatik
Universität Hamburg
Juni 1998

vorgelegt bei

Dr. Norman Hendrich
Arbeitsbereich Technische Grundlagen der Informatik (TECH)
und
Prof. Winfried Lamersdorf
Arbeitsgruppe Verteilte Systeme (VSYS)

INHALTSVERZEICHNIS

Zusammenfassung

Komponentenarchitekturen gewinnen in der Softwareentwicklung immer mehr an Bedeutung. Mittlerweile kommen die ersten Entwicklungswerkzeuge auf den Markt, mit denen Komponenten auch rein visuell zu Applikationen zusammengebaut werden können. Das an der Universität Hamburg entwickelte HADES-Framework zeigt ein Anwendungsbeispiel für solche Entwicklungswerkzeuge auf. Hierbei werden Simulationsobjekte, die als Komponenten realisiert sind, zu einem größeren System zusammengefügt. Da HADES gerade auch in der Lehre und von Studenten eingesetzt werden soll, ergibt sich die Forderung nach Plattformunabhängigkeit und möglichst einfacher Installation. Java bietet dabei mit Applets und der Plattformunabhängigkeit ideale Voraussetzungen.

Trotz der Verfügbarkeit einer Vielzahl von Komponenten fehlt den meisten Entwicklungswerkzeugen eine benutzerfreundliche Verwaltungsmöglichkeit. In der Regel werden die vorhandenen Komponenten nur mit ihrem Namen und einem kleinen Symbol in einer einfachen Liste angezeigt. Im Rahmen dieser Arbeit wird deshalb eine Architektur zur Verwaltung von JavaBeans-Komponenten entwickelt, die im "COLIBRI" (Component and Library Browser) für HADES einen ersten konkreten Einsatz findet. Zur Darstellung der vorhandenen Beans dient ein mit den Java Foundation Classes (Swing) realisierter Browser. Sofern vorhanden, stellt der Browser zusätzlich zum hierarchischen Tree-View mit eigenen Icons weitere Informationen zu den einzelnen Objekten zur Verfügung, darunter Thumbnail-Images und HTML-Dokumentation. Über die in dieser Arbeit eingeführten CLB-URLs können Beans sowohl auf dem lokalen Rechner als auch im Internet eindeutig referenziert werden. Die Kombination dieser URLs mit einem eigenen ClassLoader ermöglicht erstmals ein Versionsmanagement von JavaBeans und darüberhinaus auch die gleichzeitige Verwendung verschiedener Versionen eines Beans innerhalb einer einzelnen Java Virtual Machine.

Da die Verwendung kommerzieller Architekturen wie CORBA für HADES unter Kostenaspekten ausscheidet, wird in dieser Arbeit zusätzlich eine Serverarchitektur entwikkelt. Die Kommunikation mit den Servern basiert auf dem HTTP-Protokoll; sie kann damit auch über Firewalls hinweg eingesetzt werden. Zu ladende Komponenten werden dabei mitsamt aller referenzierten Ressourcen serialisiert und übertragen; optional ist auch eine auf Java-RMI basierte remote-Simulation vorgesehen. Eigene Suchmechanismen zum Auffinden neuer Komponenten im Internet sind sowohl im Browser als auch im COLIBRI-Server integriert.

1 Einführung

1.1 Komponentenarchitekturen

Die rasante Weiterentwicklung in der Technologie hat zu einem steilen Anstieg in der Rechenleistung moderner Computersysteme geführt. Mit den dadurch verfügbaren Ressourcen ist auch der Anspruch an moderne Software und damit deren Umfang und Komplexität gestiegen. Daher sind neue Anwendungen mit herkömmlichen Techniken kaum noch zu entwickeln und zu bewältigen [COM 95]. Programme, die einen Aufwand von etlichen Mannjahren haben, sind trotz gigantischer Entwicklungskosten nicht fehlerfrei erstellbar. Neue Programmiersprachen und -Paradigmen wie die Objektorientierung wurden benötigt.

Bereits in den 60er Jahren begann die Entwicklung immer größerer Softwaresysteme zunehmend problematischer zu werden. Damals entstand der Begriff des Software-Engineering. Während dabei der prozedurale Aspekt im Vordergrund steht, stellen Objekte das grundlegende Konstrukt der Objektorientierung dar. Sie vereinen sowohl die Datenstrukturen, als auch deren Verhalten in sich [Meyer 88], [Rumbaugh et al. 91].

Neben verbesserten Techniken der Softwareentwicklung selbst brachte diese Entwicklung aber auch den Wunsch nach wiederverwendbarer Software auf. Zunächst wurden umfangreiche Programmbibliotheken entwickelt. Ein weiterer Schritt sind nun Softwarekomponenten, die wie Bausteine zusammengesetzt werden können. Hierbei tritt der ingenieursmäßige Aspekt noch weiter in den Vordergrund. Dieses Konzept der wiederverwendbaren Komponenten, bei dem auf bereits bewährtes zurückgegriffen wird, ist in anderen Bereichen der Industrie, z.B. in der Hardware, schon lange üblich. Anstatt einen Multiplexer oder Analog-Digital Wandler jedesmal erneut zu erfinden, wird auf bereits Bewährtes zurückgegriffen. Die Benutzung bereits vorhandener und getesteter Teile senkt dabei nicht nur die Fehlerquote, sondern auch die Entwicklungskosten.

Softwarekomponenten sind in sich abgeschlossene Gebilde, deren Funktionsweise unterschiedlich weit konfigurierbar ist. Sie stellen eine architekturabhängige standardisierte Schnittstelle zur Komponenteninteraktion bereit. Klassisches Beispiel für Komponenten sind die bekannten GUI-Elemente. Ein Button z.B. stellt ein in sich abgeschlossenes Gebilde dar. Er ist in weiten Teilen individuell konfigurierbar (Größe, Farbe, Form, Inhalt, ...).

Praktische Bedeutung hat das Komponentenkonzept insbesondere mit GUI-Elementen schon seit längerem in Borlands „Delphi“ und Microsofts „Visual Basic“. Beide Entwicklungswerkzeuge basieren auf visuellen Elementen mit Eigenschaften, die zum Teil während des Programmierens (Namen, etc.) und zum Teil während der Programmausführung benutzt werden. Entscheidend dabei ist, das diese Elemente eigenständige Komponenten darstellen, die nicht fest im System integriert sind. Daher können neue Komponenten beliebig entwickelt und benutzt werden. Visual Basic enthielt in der Version 1.0 gerade einmal 16 solcher „VBX“-Komponenten (*Visual Basic Extension*). Mit Version 2.0 kamen bereits die ersten Komponenten anderer Hersteller auf den Markt und schon wenig später war der Erfolg so groß, daß andere Entwicklungstools (z.B. Borlands und Microsofts C++ Compiler) um VBX-Integration erweitert werden mußten [Udell 94]. Der

Erfolg von VBX basierte jedoch nicht auf der durchdachten Architektur, sondern auf den Möglichkeiten dieser neuen Komponenten. Die zugrunde liegende Architektur war dagegen zum einen an Windows gebunden und machte zum anderen auch sonst viele Einschränkungen [Udell 94]. Der überraschende Erfolg machte deshalb die Weiterentwicklung zu OCX notwendig und führte heute zu den bekannten „ActiveX“ Komponenten, die auch im World Wide Web Anwendung finden können.

Problempunkt aller Komponenten ist immer die Schnittstelle zur Außenwelt. Da die meisten Programmiersprachen statische Bindungen verwenden, müssen alle Daten zur Compilezeit feststehen und dürfen sich danach nicht mehr verändern. Um dennoch die nötige Dynamik zu erreichen, wird deshalb in jeder Komponentenarchitektur ein Grundinterface definiert, über das die weiteren spezifischen Eigenschaften einer Komponente ermittelt werden können. Auf die verschiedenen Komponentenmodelle wird in Kapitel 3 näher eingegangen.

1.2 JavaBeans

Die Einführung der Programmiersprache Java vereinfachte die Schnittstellenproblematik, da Java von Natur aus dynamische Bindungen verwendet. Die Interfaces von Java-Objekten können dabei direkt, also ohne Zuhilfenahme eines „Vermittlungsmechanismus“, untersucht werden. Bei den Objekten dürfen sich lediglich die Namen und Parameter der Methoden und Eigenschaften nicht nachträglich ändern. Mit dem seit Java 1.1 vorhandenen sogenannten *Reflection API* [Java Reflect 97] ist es sogar möglich, Objekte während der Laufzeit selbst zu untersuchen und zu verwenden, ohne daß diese zur Compilezeit bekannt gewesen sein müssen.

Noch vor der Einführung von Java 1.1 existierte bereits eine erste Spezifikation einer Komponentenarchitektur in Java, der *JavaBeans* [JavaBeans 96]. Wegen der in Java von vornherein verwendeten dynamischen Bindungen ist im Gegensatz zu anderen Architekturen keine zusätzliche Laufzeitumgebung nötig. Die Spezifikation der JavaBeans fällt in diesen Punkten daher auch relativ schwach aus. Da das gesamte Interface mittels integrierter Reflection-Schnittstelle dynamisch untersucht werden kann, existieren diesbezüglich nur Richtlinien.

Mit dem Erscheinen der *Enterprise JavaBeans* erweitert Sun das bisher lokale Komponenten-Modell um Server-basierte Komponenten, die den kommerziellen Einsatz vorantreiben sollen. Um das Angebot von Beans zu verbessern, sind sämtliche GUI-Elemente der neuen *Java-Foundation-Classes* [Java Swing 97] als JavaBeans implementiert. Inzwischen tauchen auch erste kommerzielle Beans auf, und das Einzige, was den größeren Einsatz zur Zeit noch verhindert, ist der Mangel an darauf aufbauenden professionellen Entwicklungswerkzeugen. Die von Sun entwickelte BeanBox dient hauptsächlich als Technologie-Demonstration. Sie ermöglicht es zwar, Beans visuell miteinander zu verbinden, läßt aber andererseits keine eigene programmiertechnische Einflußnahme zu und verhindert damit einen sinnvolleren Einsatz. Erste kommerzielle Entwicklungswerkzeuge wie Suns „Java Studio“ oder Borlands „JBuilder“ zeigen jedoch schon jetzt das Potential der Komponentenarchitekturen auf.

1.3 HADES

Einen praktischen Einsatz finden JavaBeans in HADES (*Hamburger Designsystem*), das am Arbeitsbereich der technischen Informatik an der Universität Hamburg entwickelt wird. Dieses System entstand aus dem Wunsch, die Lehrinhalte den Studenten besser vermitteln zu können. Gesucht wurden visuelle Werkzeuge, mit denen die Studenten nicht nur an der Uni, sondern auch zu Hause experimentieren können. Java schafft mit seiner Plattformunabhängigkeit dabei ideale Voraussetzungen.

HADES ist als eine Art technischer Baukasten zu verstehen, in dem Komponenten „verdrahtet“ und das Gesamtverhalten des so entworfenen Systems anschließend simuliert werden kann (objektorientierte Simulation). Durch die Realisierung der verwendeten Komponenten als JavaBeans im Zusammenspiel mit universell erweiterbaren Signalen („Drähte“) ist ein Baukasten entstanden, der auch über den technischen Bereich hinaus eingesetzt werden kann. Neben der Grundfunktionalität von elektrischen Signalen und logischen Gattern sind bereits Text-Signale und Bild-Signale mit entsprechenden verarbeitenden Komponenten vorhanden. Durch dieses Erweiterungspotential ist HADES nicht nur als „Spielzeug“ für Studenten interessant, sondern eignet sich auch für professionellere Anwendungen.

1.4 Zielsetzung

Komponenten erlangen in der Softwareentwicklung immer mehr an Bedeutung. Trotzdem gibt es bisher kaum Anwendungsumgebungen und Verwaltungswerkzeuge. Mit HADES kann ein mögliches Einsatzgebiet solcher Softwarekomponenten und die damit verbundene Problematik aufgezeigt werden. Ziel dieser Diplomarbeit ist es, einen Browser zu entwickeln und zu implementieren, der die dynamische Verwaltung dieser Komponenten für HADES übernimmt. Dabei handelt es sich zum einen um selbst erstellte HADES-Designs und zum anderen um bereitgestellte Bibliotheken von Simulationsmodellen, z.B. die mit HADES ausgelieferten Grundbausteine.

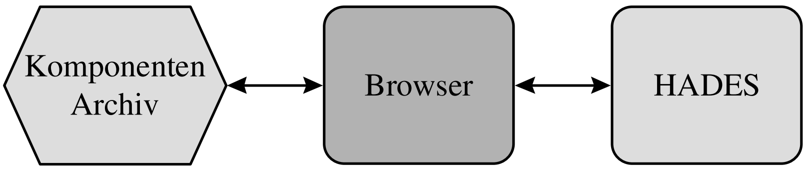

Abbildung 1: Funktion des Browsers

HADES selbst soll keinerlei Ladelogik für Komponenten enthalten und erwartet statt dessen vom Browser fertig geladene Java-Objekte. Wichtige Aufgabe des Browsers ist es deshalb, die Klassen der Komponenten in den Java-Interpreter (Virtual Machine) zu schleusen. Neben dieser grundlegenden Funktionalität ist der Browser auch für die allgemeine Verwaltung der Komponenten verantwortlich. Da JavaBeans nicht nur Programmcode (Klassen), sondern auch anderen Ressourcen wie z.B. Bilder enthalten können, muß für jede Komponente immer eine Menge von Dateien verwaltet werden. Wichtig wird dieser Aspekt insbesondere, wenn Komponenten installiert, gelöscht oder verschoben werden sollen.

Zur Komponentenverwaltung gehört weiterhin eine detaillierte Informationsanzeige und ein integriertes Versionsmanagement. Im Umgang mit fremden Komponenten können auch

Sicherheitsbetrachtungen eine Rolle spielen. Als Quelle von Komponenten dient nicht nur das lokale Dateisystem, sondern auch das Internet. Neben einer netzweiten Suche sollen Komponenten auch direkt von Servern geladen werden können. Im Zusammenhang mit dieser Möglichkeit sollte auch der kommerzielle Aspekt betrachtet werden. Eigene Komponenten sollen darüber hinaus auch auf einem Server veröffentlicht werden können. Neben dem Browser muß deshalb auch ein eigener Server entwickelt werden, der die geforderte Funktionalität liefert.

Der Fokus dieser Arbeit liegt dabei auf der Verwaltung von HADES-Simulationsmodellen. Dennoch soll versucht werden, keine Abhängigkeiten von HADES entstehen zu lassen, um auch allgemeine JavaBeans verwalten zu können. Die dabei auftretenden Probleme sollen deshalb möglichst allgemeingültig betrachtet werden.

1.5 Anforderungsermittlung

HADES dient als Softwarebaukasten, in dem Simulationsobjekte verbunden und so komplexere Designs entwickelt werden können. Ziel dieser Diplomarbeit ist es, einen Browser zu entwickeln und zu implementieren, der die dynamische Verwaltung und den Zugang dieser Komponenten für HADES übernimmt.

HADES stellt derzeit etwa 100 Komponenten zur Verfügung. Der Browser soll eine Übersicht über diese Grundbausteine und Designs geben und Informationen über sie anzeigen, damit auch ungeübte Studenten ohne große Einarbeitung eigene Schaltungen entwerfen können. Die so entstandenen Designs sollen mit Hilfe des Browser abgespeichert und verwaltet werden können. Dabei soll eine Trennung der mitgelieferten System- und Beispielkomponenten und der von Benutzer entworfenen Komponenten aufrecht erhalten werden. Im Rahmen des Einsatzes in verteilten Systemen soll der Browser auch Komponenten auf entfernten Rechner anzeigen und laden können. Dabei sind bestimmte Sicherheitsaspekte zu beachten.

Ein Simulationsmodell kann auch Untermodelle von verteilten Servern enthalten. Bei der Simulation wird dabei die Komponente in die Virtual Machine des Clients geladen und dort lokal ausgeführt. Dies kann jedoch zu Problemen führen, wenn die entsprechende Komponente sich nicht mehr auf dem Server befindet oder die Verbindung unterbrochen ist. Daher sollen entfernte Komponenten auf Wunsch auch lokal installiert werden können, um deren ständige Verfügbarkeit sicherzustellen. Der Benutzer soll zusätzlich in der Lage sein, eigene Komponenten der Öffentlichkeit zugänglich zu machen.

Der Browser soll eine übersichtliche Darstellung lokaler und entfernter Komponenten bieten. Zu diesen Komponenten soll der Browser Informationen über den Leistungsumfang und die Schnittstellen zur Verfügung stellen, die eine Auswahl erleichtern. Simulationsmodelle sind einer ständigen Evolution unterworfen. Daher sollte der Browser zusätzlich eine Versionskontrolle realisieren, um verschiedene Versionen einer Komponente nutzen zu können, ohne die Konsistenz des Gesamtsystems zu gefährden. Dabei sollte es möglich sein, verschiedene Versionen einer Komponente gleichzeitig zu verwenden.

Der Browser ist auch für das Laden und Speichern der Komponenten zuständig. Da Komponenten auch über das Internet verwaltet werden, muß bei der Speicherung eine eindeutige Bezeichnung sichergestellt werden. In diesem Zusammenhang sollen mögliche Namensschemata analysiert werden.

Außerdem soll der Browser eine geeignete Suchfunktion zur Verfügung stellen, die das Suchen von Komponenten lokal und über das Internet ermöglicht. Für eine sinnvolle Verwaltung und eine gezielte Suche sollen außerdem verschiedene Ansätze einer Klassifikation evaluiert werden.

Für den Zugriff auf Komponenten in verteilten Systemen und im Internet ist eine geeignete Serverarchitektur auszuwählen. Neben einer Pull-Technologie ist dabei auch ein Push-Modell möglich. Diese muß die oben erwähnte Suchfunktion zur Verfügung stellen und die Möglichkeit bieten, lokale Komponenten an einen Server zu exportieren. Bei fremden Komponenten sind die Sicherheitsaspekte zu evaluieren. Komponenten, die aus dem Internet geladen werden, sind mit Sicherheitsrisiken behaftet. Hier sollte möglicherweise ein Sicherheitskonzept entwickelt werden, das verschiedene Sicherheitsstufen auf Basis digitaler Signaturen (Zertifikate) bietet.

Das im Rahmen des HADES-Projektes entstehende Framework soll hauptsächlich in der Lehre Einsatz finden. Daher müssen weitere sich daraus ergebende Anforderungen berücksichtigt werden. Zum einen soll das Programm intuitiv bedienbar und damit leicht erlernbar sein. Dazu sollte es erläuternde Texte und eventuell Animationen enthalten. Damit Studenten nicht nur an der Universität, sondern auch zu Hause mit HADES arbeiten können, sollte es kostenfrei und für möglichst viele Rechnerplattformen verfügbar sein. Weiterhin sollten die Installation einfach und die erforderlichen Ressourcen gering gehalten werden. Zukünftige Updates sollten ebenfalls einfach zu installieren sein. In Zukunft soll sich HADES auch als Applet ausführen lassen, wodurch eine lokale Installation entfällt.

Wie oben beschrieben, werden entfernte Komponenten normalerweise in die Virtual Machine beim Client geladen und dort lokal ausgeführt. Eine andere Möglichkeit besteht darin, eine Simulation auch ohne eine lokale Kopie durch Objektkommunikation, z.B. mittels RMI oder CORBA durchzuführen. In dieser Arbeit ist zu evaluieren, ob dies möglich und sinnvoll ist. Im Hinblick auf einen allgemeinen Einsatz des Browsers sollen auch Möglichkeiten zur kommerziellen Nutzung von Komponenten angedacht werden.

Neben eigenen Ansätzen sollen auch bereits bestehende Informationssysteme auf ihre Eignung für bestimmte Teile des Browsers untersucht werden. Dabei können sowohl Lösungsansätze der bestehenden Systeme übernommen, als auch eigene neu entwickelt werden.

1.6 Problemspezifikation

Grundlegende Aufgabe des Browsers ist es, HADES die Komponenten als fertig geladene Java-Objekte zu übergeben. Da JavaBeans im allgemeinen nicht nur aus einer einzigen Datei bestehen, muß der Browser die Komponenten als eine Menge von Dateien verwalten und diese bei Bedarf dem Interpreter (Virtual Machine) übergeben. Hierzu muß der von Java normalerweise verwendete Mechanismus, Daten zu laden, genau untersucht werden. Beim Laden von Komponenten über einen Server kann der Java interne Mechanismus nicht verwendet werden. Hier muß deshalb ein alternativer Ansatz entwickelt werden.

Da verschiedene Versionen einer Komponente zumeist aus gleichnamigen Dateien bestehen, muß das integrierte Versionsmanagement auftretende Konflikte verhindern und einen

Weg finden, parallel verschiedene Versionen zu laden und zu speichern. Dies könnte z.B. durch Anhängen einer Versionsnummer an den Dateinamen erfolgen. Außerdem müssen verschiedene Versionen gleichzeitig geladen werden können.

Durch die Kombination verschiedener Komponenten ergeben sich komplexere Strukturen, die bei HADES ihrerseits wieder als Komponenten Anwendung finden können. Dabei entstehen hierarchische Strukturen, die bei der Verwaltung besonders zu berücksichtigen sind. So muß beim Verschieben oder Löschen einer Komponente beachtet werden, daß diese eventuell von anderen Designs verwendet wird. Allgemein sind Abhängigkeiten von anderen Komponenten zu überprüfen.

Der von uns für den Browser gewählte Name *COLIBRI* steht für Component and Library Browser. Uns ist bewußt, daß der Vogel im Deutschen mit 'K' geschrieben wird und im Englischen völlig anders, nämlich „hummingbird", heißt. Trotzdem fanden wir diesen Namen recht intuitiv und einprägsam.

1.7 Aufbau dieser Arbeit

Im Anschluß an dieses Kapitel, das einen Kurzüberblick über die behandelte Thematik liefert und die Anforderungen an den Komponentenbrowser ermittelt, wird in *Kapitel* 2 die verwendete Programmiersprache Java näher vorgestellt.

Das folgende *Kapitel 3* behandelt verschiedene Komponentenarchitekturen. Zunächst werden Komponenten im allgemeinen und die dahinterstehende Motivation beschrieben. Danach gehen wir dann auf die verschiedenen Komponentenarchitekturen, insbesondere die dieser Arbeit zugrunde liegenden JavaBeans ein. Eine Erweiterung der JavaBeans stellt die Enterprise JavaBeans Architektur dar. Als Werkzeuge zur Anwendungsentwicklung mit JavaBeans stellen wir die BeanBox und das Java Studio von Sun vor. Ein Vergleich der verschiedenen Komponentenarchitekturen beschließt *Kapitel 3.*

Das HADES Framework ist Inhalt von *Kapitel 4*. Nach einem kurzen Überblick wird dazu genauer auf den Funktionsumfang und die Funktionsweise von HADES eingegangen. Um die dahinterstehenden Ziele und Besonderheiten besser darlegen zu können, werden einige andere Simulationssysteme vorgestellt und mit HADES verglichen.

Kapitel 5 beschäftigt sich mit bestehenden Architekturen, die wir auf einen möglichen Einsatz im Rahmen des Browsers untersuchen. Inhalt von *Kapitel 5.1* ist CORBA, ein Framework über heterogene Hardware-Plattformen und Betriebssysteme für Interkommunikation von Anwendungsobjekten. Das von Sun entwickelte Modell für verteilte Objekte ist die Java Remote Method Invocation, die Inhalt von *Kapitel 5.2* ist. Es folgt ein Vergleich der beiden Architekturen. Der Vollständigkeit halber gehen wir kurz auf das Distributed Computing Environment (DCE) der Open Software Foundation (OSF) ein und stellen es CORBA gegenüber. Bezüglich der verwendeten Indizierungs- und Suchmechanismen untersuchen wir NCSTRL, ein verteiltes System zur Verwaltung von Forschungspapieren im Bereich der Informatik.

In *Kapitel 6* werden verschiedene konzeptionelle Voraussetzungen im Hinblick auf die folgende Implementation erläutert. Dazu zählen unter anderem das Laden von Komponenten, das Versionsmanagement und die Serverarchitektur.

Kapitel 7 beschreibt die konkrete Implementation des Browsers. Dazu werden die wichtigsten aufgetretenen Problemstellungen herausgegriffen und die gewählten Lösungswege anhand von Programmauszügen erläutert.

Abschließend gibt *Kapitel 8* eine Zusammenfassung der Arbeit und eine Bewertung des Erreichten. Ein Ausblick zeigt mögliche Erweiterungen des Browsers auf und beendet die Arbeit.

Anhang A enthält das COLIBRI Tutorial. In *Anhang B* beschreiben wir das Format der CLB-Datei, die Informationen über eine Komponente speichert. Ein Stichwortverzeichnis findet sich im *Anhang C*.

2 Java

In diesem Kapitel stellen wir die Programmiersprache Java vor. Wir gehen dabei auf die Besonderheiten dieser Sprache ein und beschreiben die *Java Foundation Classes* und das neue Sicherheitskonzept, daß an Version 1.2 integriert sein wird.

Java wurde 1990 von James Gosling bei Sun Microsystems für den Einsatz in der Unterhaltungselektronik und für Haushaltsgeräte entworfen und findet seit 1995 Anwendung insbesondere im Internet, aber auch zur Entwicklung von Standalone-Applikationen. Innerhalb kurzer Zeit haben bedeutende Softwarehersteller wie Netscape, Microsoft, Lotus, Oracle und IBM Java in ihre Systeme integriert.

Sun definiert Java wie folgt [Java Language 95]:

> *Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.*

Im folgenden wird Java anhand dieser Stichworte näher erläutert.

simple
Java ist relativ leicht zu erlernen. Die Anzahl der Sprachkonstrukte ist klein und auf den ersten Blick ist Java C / C++ sehr ähnlich. Dabei verzichtet es auf einige Eigenschaften von C / C++ (z.B. Zugriff auf Zeiger) und integriert dafür einige Ideen anderer Programmiersprachen (z.B. Garbage Collection) [Gosling et al. 96].

object-oriented
Java ist objektorientiert. Bei der Objektorientierung stehen nicht die Prozeduren und Funktionen im Vordergrund, sondern die Daten und deren Manipulierung. Ein Objekt beinhaltet sowohl Daten als auch Methoden [Weber 95]. Es modelliert einen Sachverhalt der realen Welt. Die Implementierung bleibt dabei nach außen verborgen, während das Verhalten durch die öffentlichen Methoden bekanntgegeben wird [Schwarz 97]. In der Objektorientierung spielt die Vererbung eine große Rolle. Objekte können Verhalten von anderen Objekten erben. Dadurch wird der Programmieraufwand reduziert, da die Funktionalität nicht jedesmal neu implementiert werden muß. In Java gibt es nur Einfachvererbung; Mehrfachvererbung kann über Interfaces nachgebildet werden. Ein Interface ist eine Klasse, in welcher nur Funktionsrümpfe definiert werden und die selbst keine Implementation enthält. In Java erben alle Objekte von der Klasse `Object`. Auf die Daten eines Objektes wird nicht direkt, sondern über entsprechende Methoden zugegriffen. Dadurch kann das Datenformat geändert werden, ohne daß sich an der Verwendung des Objekts etwas ändert. Dies wird als Kapselung bezeichnet. In der Objektorientierung kann ein gleicher Methodenaufruf je nach Implementation des Objektes und Typ der Argumente unterschiedliches Verhalten haben. Dies bezeichnet man als Polymorphismus [Schwarz 97].

distributed
Durch seine Klassenbibliothek `java.net` stellt Java verschiedene Ebenen der Netzwerkkonnektivität zur Verfügung. Dazu gehören Sockets und URLs. Applets sind Programme,

die über das WWW geladen und in der Virtual Machine des Browsers ausgeführt werden. Außerdem stellt Java mit der *Remote Method Invocation* eine Architektur für verteilte Systeme zur Verfügung. Über die Datenbankschnittstelle *JDBC* kann auf entfernte Datenbanken zugegriffen werden. Objekte werden mit Hilfe der *Object Serialization* serialisiert und können so über das Netzwerk übertragen werden.

interpreted, architecture neutral

Ein Java Programm wird zunächst in einen plattformunabhängigen Bytecode übersetzt. Dieser wird bei Programmausführung von einem virtuellen Rechner, der *Java Virtual Machine,* interpretiert. Damit ist Java auf jedem System lauffähig, für das eine solche Virtual Machine implementiert ist. Eine Virtual Machine existiert für alle relevanten Plattformen, wie Windows95/NT, MacOS, OS/2, Linux und alle gängigen Unix-Varianten.

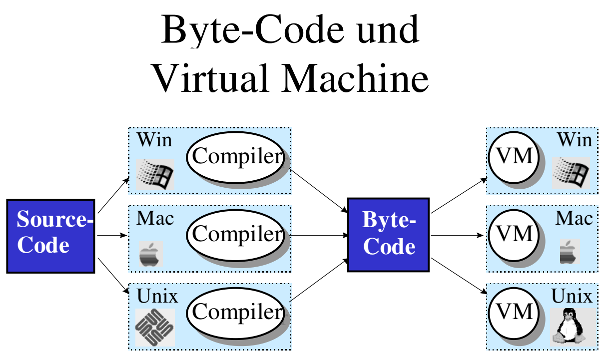

Abbildung 2 Plattformunabhängigkeit von Java

Sowohl der Java-Quellcode, als auch der erzeugte Bytecode sind plattformunabhängig, so daß z.B. ein auf Windows compiliertes Programm ohne Änderung auf einem unter Solaris laufenden Rechner ausgeführt werden kann.

Mittlerweile ist es auch eingeschränkt möglich, aus Java-Programmen direkt ausführbaren (nativen) Code zu erzeugen. Die Plattformunabhängigkeit geht dann jedoch zu Gunsten einer höheren Performanz verloren.

Der Microsoft Internet Explorer und der Netscape Navigator, sowie der von Sun in Java geschriebene Browser HotJava, haben ebenfalls eine eingebaute Virtual Machine. Der Navigator und der Internet Explorer bieten derzeit allerdings nur eine eingeschränkte Unterstützung für Java 1.1. In Java gibt es Programme, sogenannte Applets, die von einer Webseite aus abgerufen, heruntergeladen und dann auf der Client-Seite im Web-Browser ausgeführt werden. Der entscheidende Vorteil liegt hier darin, daß auf der Client-Seite keine Installation nötig ist und daß ein Update nur an einer Stelle auf dem Server durchzuführen ist.

portable

Die Portabilität beinhaltet neben der Plattformunabhängigkeit weitere Punkte [Java Language 95]. So gibt es in Java gibt es keine systemabhängigen Sprachaspekte. Die primitiven Datentypen sind auf allen Maschinen und Implementationen gleich definiert [Gosling et al. 96]. Ein Zugriff auf Betriebssystemfunktionen wird vermieden. In den neuen Java Foundation Classes sind auch alle GUI-Komponenten vollkommen in Java implementiert und greifen nicht mehr auf Systemkomponenten zurück.

robust
Java wurde ursprünglich für Konsumentenprodukte entwickelt und mußte daher sehr zuverlässig und robust sein. Java eliminiert viele typische Programmfehler, z.B. weil der Benutzer keinen Zugriff auf Zeiger erhält. Eine strenge Typisierung ermöglicht eine intensive Überprüfung zur Compilezeit und die automatische Garbage Collection eliminiert Speicher-Allokations-Probleme weitgehend. Die Ausnahmebehandlung (Exception Handling) ermöglicht es, Programmfehler abzufangen und entsprechend auszuwerten. Damit ist die Wahrscheinlichkeit einer Programmabsturzes wesentlich geringer als in C/C++.

secure
Eine Sprache, die für den Einsatz in verteilten Systemen gedacht ist, darf Sicherheitsaspekte nicht außer acht lassen. Da Java gekapselt in der Virtual Machine und von der Hardware entkoppelt abläuft, lassen sich alle System-Zugriffe kontrollieren. Da Java keine Zeiger enthält, wird die Programmierung von Viren erschwert. Java geht von dem Grundsatz aus, daß niemandem zu trauen ist. Aus dem Netz geladener Programmcode wird daher vor und während der Ausführung ständig überprüft. Java Applets haben zunächst wenig Rechte. Sie dürfen z.B. nicht auf das lokale Dateisystem zugreifen und keine beliebigen Netzwerkverbindungen aufbauen. Es gibt jedoch die Möglichkeit, Java Applets zu signieren, um ihnen dann mit expliziter Zustimmung des Benutzers mehr Rechte einzuräumen. Java stellt dazu auf Public-Key-Verfahren basierende Authentisierungs-Mechanismen zur Verfügung.

high-performance
Sun schreibt in seiner Spezifikation, Java sei hoch performant. Dies muß jedoch in Frage gestellt werden. Für eine interpretierte Sprache ist Java zwar schnell, im Vergleich zu C++ ist Java jedoch deutlich langsamer. Just-In-Time-Compiler können die Performanz verbessern, jedoch gehört die Geschwindigkeit nach wie vor zu den größten Problemen von Java.

multithreaded
Java enthält eingebaute Unterstützung für Threads. Ein Thread ist ein eigenständiger Ausführungspfad in einem Prozeß mit einem eigenen Programmzähler und Stack aber vollem Zugriff auf die Objekte innerhalb der Programms [Tanenbaum 95]. Mehrere Programmteile lassen sich dadurch gleichzeitig ausführen. Hierbei können Probleme auftreten, insbesondere wenn parallele Threads im selben Moment auf ein Objekt zugreifen. Java bietet hierzu die Möglichkeit der automatischen Synchronisation von Threads. Allerdings kann es hierbei zu Deadlocks kommen. Hier sollten Monitore verwendet werden.

dynamic
Klassen werden bei Bedarf dynamisch nachgeladen. Java wurde für eine sich schnell entwickelnde Umgebung entworfen. Durch die dynamischen Bindungen kann eine Klasse nachträglich durch eine andere Version ersetzt werden, ohne daß die gesamten Klassen neu gelinkt werden müssen.

Ein Nachteil von Java als relativ neue Programmiersprache ist, daß es noch nicht völlig ausgereift ist und regelmäßig neue Versionen erscheinen. Diese bringen zum einen neue Möglichkeiten, zum anderen aber auch inkompatible Änderungen mit sich. Durch die ständige Weiterentwicklung werden zwar alte Fehler in den Systembibliotheken beseitigt, es kommen aber leider laufend neue hinzu, mit denen die Entwickler zu kämpfen haben.

Mit Version 1.1 wurden die Java Klassenbibliotheken überarbeitet und um zahlreiche Packages erweitert. Außerdem wurde ein neues Eventmodell eingeführt. Ab Version 1.2 wird auch ein verbessertes Sicherheitskonzept in Java integriert sein.

2.1 Java Foundation Classes / Swing

Ein großer Vorteil von Java sind die mitgelieferten umfangreichen Klassenbibliotheken. Sie übertreffen den Umfang der etwa für C / C++ und Pascal standardmäßig bereitgestellten Funktionen bei weitem [Hendrich 97]. Um weitere GUI-Funktionen zur Verfügung zu stellen, hat Sun gemeinsam mit IBM und Netscape die *Java Foundation Classes* (JFC) entwickelt, die einige Neuerungen von Sun mit den von Netscape entwickelten Internet Foundation Classes vereinen [JFC 97]. Die *Java Foundation Classes* erweitern das ursprüngliche AWT-Package, welches zur anspruchsvollen GUI-Entwicklung bisher wenig geeignet war, und werden ab Java 1.2 fest integriert sein.

Das Kernstück der JFC ist Swing, ein Satz von GUI-Komponenten, wie Toolbars, Buttons, Menüs etc., deren Aussehen („Look and Feel") sogar zur Laufzeit umgeschaltet werden kann [Dalheimer 98]. Zusätzlich umfassen die JFC noch eine Reihe weiterer APIs, wie Java 2D, „Drag and Drop" und das Accessibility API, das Behinderten die Bedienung von Java Anwendungen vereinfachen soll. Das normale AWT-Package besteht neben Swing weiter.

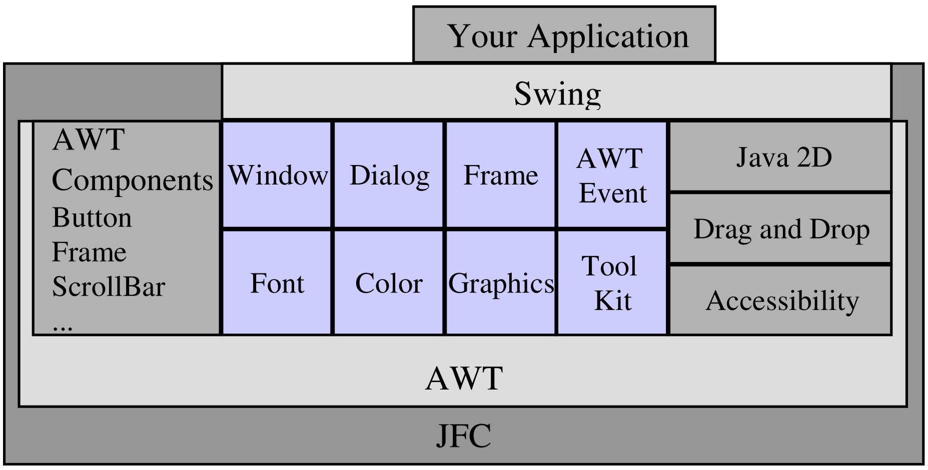

Abbildung 3: Die Swing-Architektur [Andrews 98]

Im herkömmlichen AWT-Package hat eine GUI-Komponente ein entsprechendes plattformabhängiges Gegenstück, welches als „Peer" bezeichnet wird [JFC 97]. Die AWT-Komponente dient dabei als Wrapper [Gamma et al. 94] und bildet die Java-Klasse auf native Aufrufe, bzw. Bibliotheken des zugrundeliegenden Betriebssystems ab. Dazu gibt es für jede Komponente ein plattformspezifisches „Peer"-Interface.

So gibt es zu der Klasse java.awt.Button eine entsprechende Klasse java.awt.peer.ButtonPeer. Dadurch hat eine Java Komponente auf jeder Plattform das entsprechende lokale Aussehen. Der Nachteil dieses Modells liegt an der Komplexität durch die zusätzlichen Klassen und den mangelnden Einflußmöglichkeiten. Das AWT kann auf diese Weise nur eine Schnittmenge der von den einzelnen unterstützten Plattformen zur Verfügung gestellten Features anbieten. Außerdem verhalten sich die Komponenten aufgrund von Implementationsunterschieden nicht auf allen Plattformen vollkommen gleich.

Bei dem in den JFC verwendeten sogenannten „Pluggable Look and Feel“ sind sämtliche Komponenten komplett in Java geschrieben. Man nennt sie „lightweight components“, weil der Zugriff auf die Peers entfällt. Dadurch hat ein Programm immer genau das vom Entwickler spezifizierte Aussehen.

Zum Beispiel kann einem Programm für jede Plattform das entsprechende „Look and Feel“ zugeordnet werden. Der Entwickler kann aber auch ein bestimmtes Aussehen angeben, welches dann auf allen Plattformen gleich bleibt (ausgenommen dem Windows-„Look and Feel“, da Microsoft für andere Plattformen keine Lizenzen erteilt hat). Swing enthält bereits die „Look and Feels“ für Windows, Motif, MacOS und den eigenen Metal-„Look“. Der Entwickler kann aber auch selbst ein eigenes Aussehen entwerfen [Andrews 98]. Im allgemeinen hat die Anforderung an ein jeweils plattformspezifisches „Look and Feel“ durch die weite Verbreitung des Internets und damit des Browsers als Benutzeroberfläche abgenommen und der Wunsch geht in Richtung einer gleichen Oberfläche für alle Plattformen [Dallheimer 98].

Die JFC enthalten neben anspruchsvollen GUI-Komponenten, wie Treeviews und Tabellen auch Unterstützung für Kontext-Menüs, Scrollbalken und alternative Funktionsweisen (Accessibility API). Sie ermöglichen dadurch die Entwicklung benutzerfreundlicher Programme. Für die Zukunft sind weitere Funktionen, wie „Drag and Drop“ und weitere GUI-Komponenten geplant. Die JFC-Komponenten sind als JavaBeans realisiert.

Eines der Ziele des Swing Projektes war es, ein einziges API zur Verfügung zu stellen, das verschiedene „Look and Feels“ ermöglicht. Um dies zu erreichen, basiert Swing auf der „modifizierten Model-View-Controller“-Architektur [Folwer 98].

Das Model-View-Controller-Paradigma wurde in [Krasner & Pope 88] vorgestellt und ist ursprünglich ein Modell für Benutzeroberflächen in Smalltalk. Es teilt den Entwurf einer Benutzeroberfläche in drei Komponenten auf: Das Modell (*Model*) ist das eigentliche Anwendungsobjekt. Eine Sicht (*View*) ist die Bildschirmrepräsentation dieses Modells. Dabei kann es mehrere Sichten auf ein Modell geben (siehe Abbildung). Die Steuerung (*Controller*) ist für die Benutzereingaben verantwortlich.

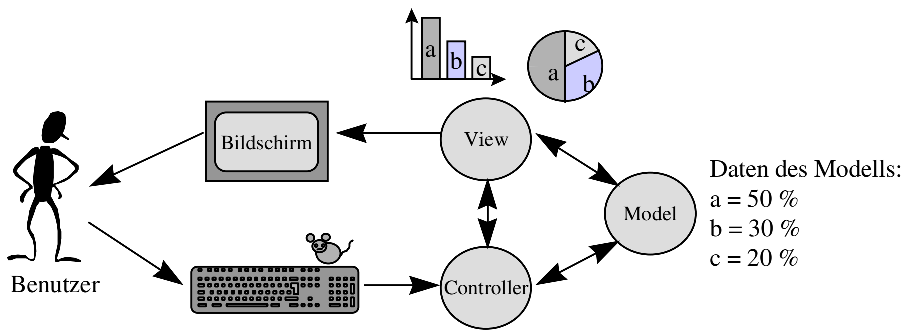

Abbildung 4: Das Model-View-Controller Paradigma

Das Modell sendet alle Änderungen seiner Daten an die Sichten. Daraufhin kann jede Sicht sich selbst durch Zugriff auf das Modell aktualisieren. Für das Modell bleibt es dabei transparent, in welcher Weise die Sichten die Daten des Modells verwenden. Die Steuerung legt fest, wie die Daten des Modells manipuliert werden können [Gamma et al. 94].

In Swing wurde diese Architektur modifiziert. Da View und Controller schwer zu trennen sind, werden sie in Swing zum *UIdelegate* verschmolzen [Fowler 98].

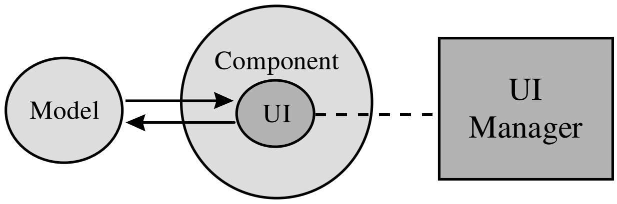

Abbildung 5: modifizierte MVC-Architektur

Eine Komponente übernimmt die View- und Controller-Funktionalität und delegiert die graphischen Aspekte an das UI-Objekt des aktuellen „Look and Feel" weiter.

Für den `JButton` gibt es z.B. folgende Klassen:

- die *Komponente*
  `com.sun.java.swing.JButton.class`
- das *Modell*
  `com.sun.java.swing.ButtonModel.class`
- die *abstrakte Klasse* des UIdelegates
  `com.sun.java.swing.plaf.ButtonUI.class`
- und für die einzelnen Plattformen *konkrete Subklassen* der UIdelegates
  `com.sun.java.swing.plaf.motif.MotifButtonUI.class`
  `com.sun.java.swing.plaf.windows.WindowsButtonUI.class`
  `com.sun.java.swing.plaf.metal.MetalButtonUI.class.`

Für einen einfachen Button wird die Klasse `ButtonModel` normalerweise nicht verwendet. Bei komplexeren Komponenten, wie z.B. Tabellen, Listen oder Bäumen kann es aber durchaus Sinn machen, das entsprechende Modell zu verwenden, um verschiedene Sichten auf eine Datenstruktur zu ermöglichen, obwohl dies nicht notwendig ist.

2.2 Sicherheit in Java

Java bietet seit jeher ein integriertes Sicherheitskonzept, das auf dem sogenannten „Sandbox“-Modell basiert [Fritzinger & Mueller 96]. Nicht vertrauenswürdige Programme laufen dabei innerhalb einer Umgebung (Sandbox) ab, die den Zugriff auf alle sicherheitsrelevanten Ressourcen wie Dateien, Netzwerkverbindungen, native Prozeduren und das System verhindert. Als nicht vertrauenswürdig werden bisher von Java nur Applets angesehen. Lokale Applikationen laufen dagegen immer mit vollen Zugriffsrechten ab. Um zu verhindern, daß das Sicherheitskonzept durch Klassenmanipulationen umgangen werden kann, wird beim Laden jeder Klasse grundsätzlich deren Integrität überprüft (Bytecode verification).

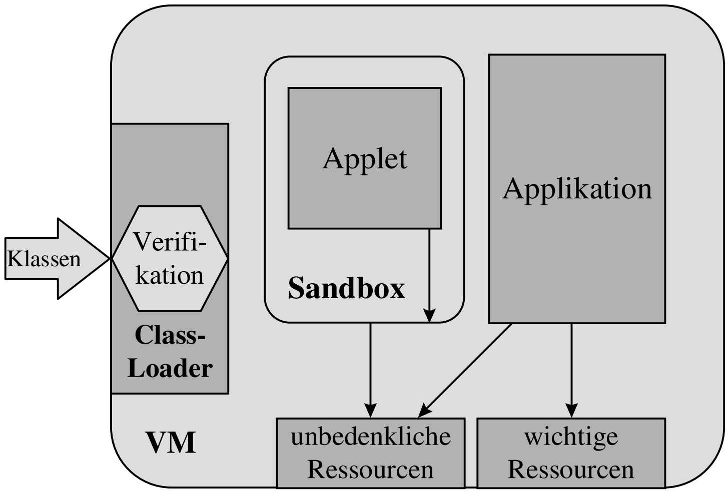

Abbildung 6: Java-Sicherheitskonzept

Zur Verhinderung unerlaubter Zugriffe verwendet Java die Klasse `java.lang.SecurityManager`, die alle sicherheitsrelevanten Aktionen vorher genehmigen muß. Dieser SecurityManager ist immer für die gesamte Applikation zuständig, kann aber durch einen eigenen ersetzt werden (bei Applets natürlich nicht), der dann differenziertere Zugriffsrechte ermöglicht. Die nachfolgende Abbildung verdeutlicht den Ablauf sicherheitsrelevanter Aktionen anhand eines Dateizugriffs.

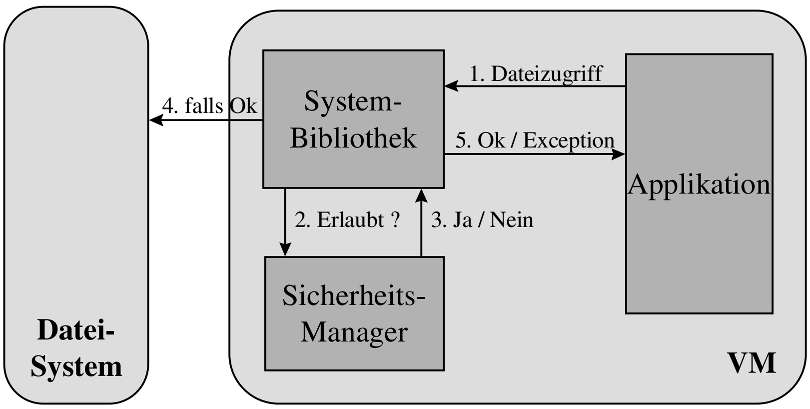

Abbildung 7: Beispiel eines Dateizugriffs

Applets müssen für viele Anwendungsgebiete jedoch auch mehr Zugriffsrechte erlangen können. Mit Java 1.1 wurde deshalb die Möglichkeit eingeführt, Applets mit einer digitalen Signatur zu versehen. Dazu wurde das Security-Package integriert, welches digitale Signaturen, Zertifikate und Verschlüsselungsalgorithmen umfaßt. Signierte Applets können vom Benutzer als vertrauenswürdig eingestuft werden und daraufhin mehr Rechte erlangen, ohne die allgemeine Sicherheit zu gefährden [Fritzinger & Mueller 96].

Im Zusammenhang mit Komponenten ist es allgemein wünschenswert, jeder Komponente unterschiedliche Rechte zuordnen zu können. Wie oben beschrieben, ist dies zwar mittels eines eigenen SecurityManagers möglich, bedeutet jedoch erheblichen Programmieraufwand, der auch Quelle neuer Sicherheitslücken sein kann. Das neue Java 1.2 integriert deshalb diese Möglichkeit und führt sogenannte „Protection Domains" ein, denen unterschiedliche Rechte eingeräumt werden können. Jede Klasse bzw. jedes Objekt gehört dabei immer genau einer Domäne an und unterliegt dessen Beschränkungen. Da beliebig viele Domänen erzeugt werden können, ist die Vergabe von Zugriffsrechten beliebig fein auflösbar. Anstelle des SecurityManagers verwendet Java 1.2 dazu die neue Klasse `java.security.AccessController`, die den Initiator des Zugriffs ermittelt (Stack Verfolgung) und die zugehörigen Domäne-Rechte überprüft. Er bekommt jeweils ein sogenanntes „Permission"-Objekt übergeben, das genau die zu genehmigende Aktion beschreibt. Der aus den vorherigen Java Versionen bekannte SecurityManager wurde aus Gründen der Abwärtskompatibilität bestehen gelassen, seine Funktionsweise jedoch dem neuen Konzept angepaßt [Gong et al. 97].

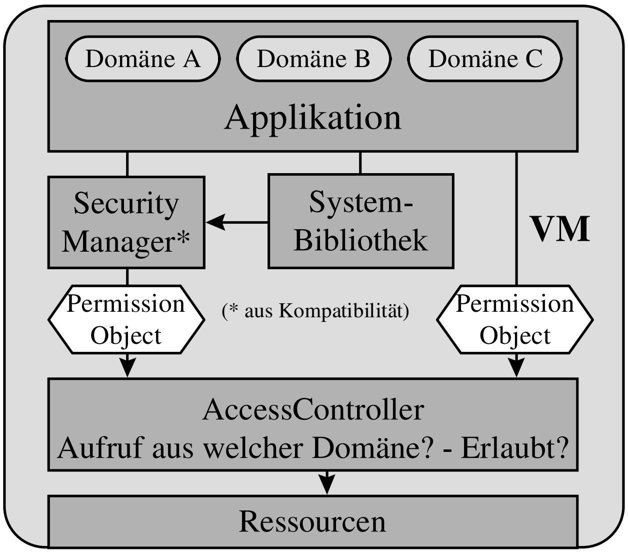

Abbildung 8: Sicherheitskonzept bei Java 1.2

Mit der neuen Klasse `java.security.Main` bietet Java nun erstmals auch die Möglichkeit, Applikationen von vornherein mit eingeschränkten Rechten zu starten. Dazu muß der Interpreter mit „`java java.security.Main Applikation`" gestartet werden. Die Klasse `java.security.Main` sorgt dann dafür, daß die eigentliche Applikation nur eingeschränkte Rechte bekommt.

3 JavaBeans und andere Komponentenarchitekturen

Komponenten erlangen zur Softwareentwicklung immer mehr an Bedeutung. Nach einer kurzen Einführung in Komponenten stellt dieses Kapitel einige Komponentenarchitekturen verschiedener Hersteller vor. Den Anfang macht dabei das von Microsoft stammende COM und dessen Nachfolger ActiveX. Es folgen SOM von IBM und OpenDoc der CI-Labs. Die in HADES verwendeten JavaBeans beschließen dieses Kapitel. Dabei gehen wir auch auf die neuen Enterprise JavaBeans und die BeanBox und JavaStudio als Entwicklungswerkzeuge für Java ein.

3.1 Was sind Komponenten?

Wie bereits in der Einleitung erwähnt, entstand die Komponentenarchitektur aus dem Wunsch nach wiederverwendbarer Software. Durch die rapide ansteigende Leistung von Rechnersystemen werden Programme immer komplexer und umfangreicher. Bestehende Anwendungen sind nur schwer erweiterbar und kaum noch zu pflegen. Bei der Komponenten-basierten Programmierung werden Anwendungen nicht mehr komplett neu geschrieben, sondern Komponenten werden zusammengefügt und formen so das neue Programm.

Eine Komponente ist ein wiederverwendbarer, in sich abgeschlossener Softwarebaustein, der unabhängig von einer konkreten Anwendung ist. Er bietet eine bestimmte Funktionalität und liefert Schnittstellen zur Komponenteninteraktion. Die Schnittstelle einer Komponente gehorcht bestimmten Standards (z.B. CORBA, COM) und erlaubt dadurch eine Verteilung von Komponenten über verschiedene Umgebungen [Components 97]. Komponenten sollen nicht nur von Entwicklern verwendet werden, sondern auch von Endanwendern zu Applikationen zusammengesteckt werden können.

Durch Komponenten sinken der Programmieraufwand und damit die Kosten. Man greift auf bereits Bewährtes zurück. Momentan besteht weltweit ein großer Mangel an qualifizierten Softwareentwicklern. Zur gleichen Zeit steigt aber der Bedarf an zusätzlicher Software immens. In diesem Umfeld wird die Komponententechnologie als ein Weg gesehen, die Produktivität von Entwicklern zu steigern, die Entwicklungszeit zu senken und die Qualität der Software zu verbessern [Andrews 96].

Ziel der Komponenten-Technologie ist die Wiederverwendung, also ein erneuter Einsatz der Komponente bei Entwicklung neuer Softwaresysteme. Es handelt sich um eine sogenannte Black-Box-Wiederverwendung, da nur das Interface wiederverwendet und von der Implementation abstrahiert wird. Der Quellcode der Komponente ist dabei nicht bekannt. Man unterscheidet hiervon die White-Box-Wiederverwendung z.B. bei Klassenbibliotheken, bei denen die Implementation bekannt ist und auf die durch Vererbung zugegriffen wird. Bei der Black-Box-Wiederverwendung entsteht eine losere Kopplung. Sie ist daher besser für dynamische Szenarien geeignet und ermöglicht eine Integration von Komponenten sogar zur Laufzeit.

Komponenten arbeiten auf binärer Ebene zusammen. Die Schnittstelle wird von der Implementation getrennt und die Methodenaufrufe werden über die Schnittstelle an die binäre Implementierung weitergeleitet. Eine Komponente soll im Gesamtsystem austauschbar bleiben, damit sie auch nach dem Compilieren z.B. durch eine neuere Version ersetzt werden kann, ohne das Gesamtsystem zu beeinflussen. Da dies mit statischen Bindungen nicht möglich ist, sind Komponenten auf dynamische Bindungen zur Laufzeit angewiesen.

Eine wichtige Voraussetzung für die Wiederverwendbarkeit von Komponenten ist das Vertrauen der Benutzer zu den Komponenten und deren Entwicklern. Hier kann der Browser mit Informationen über die Komponenten und deren Entwicklern und eingebauten Sicherheitskonzepten helfen, dieses Vertrauen herzustellen.

Komponenten verschiedener Entwickler sollen auch über Prozeß- und Rechnergrenzen hinweg zusammenarbeiten. Dazu gibt es zu den meisten lokalen Komponentenmodellen verteilte Erweiterungen.

3.2 Komponentenarchitekturen

3.2.1 COM / ActiveX

COM ist die Komponentenarchitektur von Microsoft. Sie existiert bereits seit 1993 und ist heute fester Bestandteil der 32-Bit Windows-Plattformen [Kindel 97]. Die Erweiterung von COM zu *DCOM* (Distributed Component Object Model) ergänzt die Komponentenarchitektur um eine netzweite Kommunikations-Architektur. Für die eigentlichen Komponenten ergeben sich dadurch jedoch kaum Änderungen. In [Kindel 97] heißt es dazu "DCOM is just COM with a longer wire".

Konzeptionell ist COM nicht an Windows gebunden, sondern funktioniert auch auf anderen Systemen. Durch die feste Integration in den weit verbreiteten Win32-Plattformen ist COM eine Architektur mit großer praktischer Bedeutung. Bereits heute besteht für COM-Komponenten ein großer Markt, der in den nächsten Jahren rapide weiter wachsen wird. Microsoft selbst hat heute bereits einen großen Teil von Windows in Form von COM-Komponenten realisiert. Die derzeitige Portierung der Architektur auf mehrere Unix Plattformen und den Macintosh wird diese Entwicklung noch verstärken [COM 97].

Grundidee der COM-Architektur sind Interfaces, über die die Komponenten nach einer „Vermittlung“ direkt miteinander kommunizieren können. Im Gegensatz zu anderen Architekturen sind diese Interfaces allerdings fest und können später nicht erweitert werden. Dafür können COM-Komponenten mehrere Interfaces bereitstellen, wodurch eine abwärtskompatible Erweiterbarkeit möglich ist. Alle Interfaces werden in einer Microsoft spezifischen *Interface Definition Language* (IDL) [COM 95] definiert und dann in einer Programmiersprache implementiert. Funktionalität, die sich nicht über die IDL ausdrücken läßt, kann anderen Komponenten nicht zugänglich gemacht werden.

Der Basismechanismus von COM beruht auf einem Minimalinterface `IUnknown`, das von jeder Komponente bereitgestellt werden muß [Chen 97]. Dieses Interface enthält unter anderem eine Funktion namens „QueryInterface“, die für ein abgefragtes Interface entweder einen entsprechenden Interface-Pointer oder Null zurückliefert, falls das angegebene

Interface nicht unterstützt wird. Um Interfaces klar identifizieren zu können, erhalten sie einen global eindeutigen 128-Bit Integer Identifier.

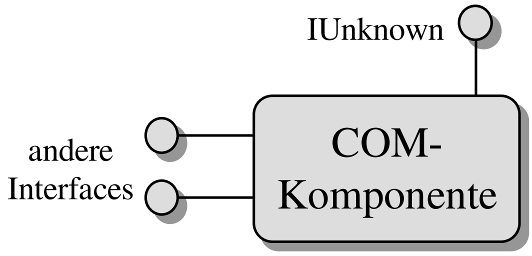

Abbildung 9: COM-Komponente

Hat sich eine Komponente derart über ein vorhandenes Interface informiert, können die enthaltenen Funktionen direkt über den erhaltenen Interface-Pointer aufgerufen werden. Der dadurch entstehende Overhead ist minimal [COM 95], wodurch Performanz-Probleme vermieden werden und der Einsatz von COM auch innerhalb von Applikationen möglich wird (z.B. innerhalb Windows). Mittels des Umweges über Interface-Pointer können Aufrufe von Funktionen auf entfernten Objekten trotzdem noch abgefangen und transparent vollzogen werden (DCOM). Solange die Komponenten auf diesen Vorgang keine Kontrolle benötigen, bleiben sie, wie bereits oben erwähnt, dabei unverändert. Bei der Kommunikation mit entfernten Objekten unterstützt DCOM eine Reihe von etablierten Protokollen, wie TCP/UDP, SPX/IPX, NetBEUI oder HTTP.

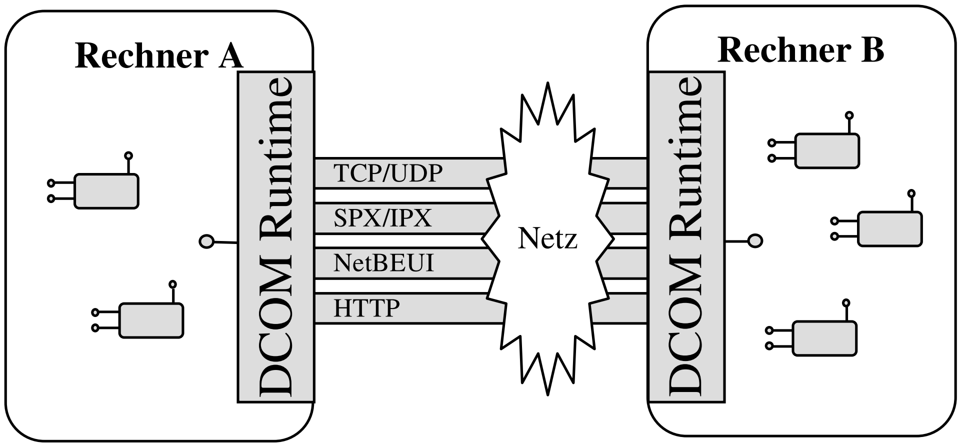

Abbildung 10: DCOM

Ein wichtiger Aspekt der DCOM-Architektur ist die Sicherheit. Während die Hersteller bei CORBA mangels einer fertigen Spezifikation lange Zeit auf eigene und damit inkompatible Lösungen angewiesen waren, unterstützt DCOM zur Netzwerkkommunikation verschiedene auch anderweitig Verwendung findende Sicherheitsprotokolle, wie NTLM, SSL/Certifications, Kerberos oder DCE [COM 97].

COM stellt bei Microsoft heute eine wichtige Basistechnologie dar. So beruht darauf z.B. das bekannte OLE (*Object Linking and Embedding*), welches es ermöglicht, Dokumente bzw. Objekte verschiedener Applikationen ohne vorherige Konvertierung in ein Dokument einzubetten und dort sogar weiter bearbeiten zu können. Dieser Ansatz ist eng verwandt mit der später erläuterten „OpenDoc“-Architektur.

Im Herbst 1997 wurde von Microsoft *COM+* vorgestellt, das die Möglichkeiten von COM erweitern soll. So wird in die Runtime neben anderen neuen Services auch der Microsoft Transaction Server (MTS) integriert, der einen wichtigen Baustein für zukünftige verteilte Applikationen darstellt. Viele der neu hinzukommenden Services spiegeln die intensive Entwicklungsarbeit an den in Konkurrenz stehenden Enterprise JavaBeans wieder. Für spätere Versionen von COM+ hat Microsoft auch eine verbesserte Garbage-Collection und die Möglichkeit der Vererbung angekündigt [Moeller 98].

Das ebenfalls relativ neue *ActiveX* ist aus den Anforderungen des Internets hervorgegangen und stellt ein erweitertes OLE dar [Chen 97]. Die neu hinzugekommenen Features basieren stark auf der speziellen Einbettung in WWW-Browser. Neben der erweiterten multimedialen Unterstützung (Sound, Video, etc.) spielt hierbei z.B. auch ein optimiertes Download-Verfahren eine Rolle. ActiveX basiert damit über OLE letztlich auch auf COM. In einer Microsoft Präsentation heißt es daher auch: "ActiveX is a marketing name for a set of technologies and services, all based on COM" [Kindel 97].

3.2.2 SOM

Bereits 1991 stellte IBM eine Komponentenarchitektur namens SOM (System Object Model) vor, die ähnlich wie COM/DCOM später mit DSOM (Distributed SOM) um den Verteilungsaspekt erweitert wurde.

SOM beruht wie COM auf Interfaces, die über eine erweiterte *Interface Definition Language* (IDL) der OMG (siehe CORBA) spezifiziert werden. Im Gegensatz zu COM ist SOM aber ein objektorientierter Ansatz, bei dem die binär vorliegenden Komponenten auch mittels Vererbung benutzt werden können [Pfister 95]. Erkauft werden diese Vorteile durch eine aufwendigere Laufzeitumgebung. Die engere Anbindung der Laufzeitumgebung an die verwendete Programmiersprache gestaltet die Verwendung von SOM schwierig. Erst eine integrierte SOM Unterstützung („direkt-to-SOM“) der Programmierumgebungen konnte hier Abhilfe schaffen. Microsofts COM hat hier Vorteile, da die verwendeten Interface-Pointer (Sprungtabellen) einen schon vorher gängigen und damit unterstützten Standard darstellten.

DSOM ist eine auf SOM aufbauende Klassenbibliothek, die eine der von SOM unterstützten Bindungsarten[1] („method dispatch“) CORBA kompatibel erweitert [SOM 94]. Leider läßt sich hierbei die so wichtige Möglichkeit der Vererbung nicht mehr benutzen, wodurch die eigentlichen Vorzüge von SOM entfallen und vom Programmierer zwei verschiedene Modelle zu berücksichtigen sind [Box 95]. Nach dem Erscheinen von Version 3.0 1996 ist es still um SOM geworden, und nachdem IBM einen Großteil seiner Aktivitäten auf den Bereich Java konzentriert hat, wurde die Entwicklung von SOM offiziell eingestellt.

[1] Die normalerweise verwendeten statischen Bindungen sind das eigentliche Problem binär wiederverwendbarer Komponenten. Grundlage jeder Komponenten Architektur ist es deshalb, dynamische Bindungen zu erzeugen.

3.2.3 OpenDoc

OpenDoc ist eine Komponentenarchitektur der *Component Integration Laboratories* (CI-Labs), einem Zusammenschluß der Firmen Apple, IBM, Novell, Oracle und anderen. Mit Apple als einem der aktivsten Mitglieder des Konsortiums, lag das Hauptziel von OpenDoc in der Entwicklung von zusammensetzbaren Dokumenten. Dabei sollen Dokumente anderer Applikationen einfach in ein Dokument eingefügt und dort auch weiter bearbeitet werden können. Dieser Ansatz ist eng verwandt mit Microsofts OLE (Object Linking and Embedding) und wurde aus praktischen Gründen auch kompatibel zu diesem gehalten. Durch die spezielle Ausrichtung von OpenDoc wurde auf ansonsten ungewöhnliche Merkmale wie etwa einen Undo-Stack Wert gelegt. So wie Microsofts OLE auf COM als Objekt-Modell basiert, benötigt auch OpenDoc ein darunterliegendes Modell und verwendet dazu IBMs SOM.

Ein eingebundenes Objekt („part") kann bei OpenDoc jederzeit zu einem Hauptdokument gemacht werden, indem es per „Drag and Drop" aus einem Dokument herausgezogen wird (und umgekehrt). Weiterhin können „parts" ihrerseits auch wieder „parts" enthalten, sich gegenseitig sogar überlappen und nicht nur rechteckige, sondern beliebige Formen annehmen Alle diese Punkte sind bei Microsofts OLE erst für die Zukunft angekündigt.

Anfang 1997 hat Apple OpenDoc zugunsten der JavaBeans-Technologie aufgegeben, woraufhin CI-Labs offiziell aufgelöst wurde. Die OpenDoc Implementation im MacOS wird daher wohl die letzte sein, es sei denn, jemand verfolgt den zwischenzeitlichen Gedanken weiter, OpenDoc komplett in Java und damit kompatibel zu JavaBeans zu implementieren.

Einen großen Erfolg verzeichnete Apple mit dem auf OpenDoc basierenden „*Cyberdog*". Diese Internet Applikation besteht aus verschieden OpenDoc Komponenten, wie etwa einen HTML-Viewer, einem Email-Client oder einem News-Reader [Cyberdog 96]. Durch die Realisation in OpenDoc können die jeweiligen Inhalte (Mails, Webseiten, etc.) nahezu beliebig miteinander verknüpft werden, wodurch eine Art völlig konfigurierbarer Multifunktions-Browser entsteht, mit dem herkömmliche WWW-Browser im Punkt Benutzerfreundlichkeit kaum konkurrieren können.

3.3 JavaBeans - Softwarekomponenten in Java

Das JavaBeans Framework ist eine portable, plattformunabhängige Softwarekomponenten-Architektur. Es ermöglicht es den Entwicklern, Softwarekomponenten einmal zu schreiben und sie dann, da vollständig in Java geschrieben, in binärer Form überall einzusetzen. Ein JavaBean ist eine wiederverwendbare Softwarekomponente, die leicht in andere Anwendungen oder Applets eingebunden werden kann. Sun gibt in der JavaBean Spezifikation 1.0 [JavaBeans 96] folgende Definition für ein JavaBean:

> *„A JavaBean is a reusable software component that can be manipulated visually in a builder tool."*

Beans können in einem visuellen Entwicklungswerkzeug manipuliert und zu Applikationen zusammengestellt werden. Dabei ist ein Bean ein beliebiges Java Objekt, das bestimmte Konventionen im Bezug auf Schnittstellen und Eigenschaften erfüllt. Dazu wurde ein Komponentenmodell geschaffen, daß folgende Möglichkeiten bietet:

- *Introspection:* Ein Entwicklungswerkzeug kann in einem gewissen Rahmen feststellen, welche Funktionen eine Komponente (Bean) anbietet. Dazu stellt es fest, welche Methoden die Komponente zur Verfügung stellt. Der Analysevorgang wird als Introspection bezeichnet.
- *Properties:* Eine Komponente weist verschiedene Eigenschaften in Form von Instanzvariablen auf.
- *Customization:* Das Verhalten und Aussehen einer Komponente kann mittels eines (visuellen) Entwicklungswerkzeuges angepaßt werden.
- *Events:* Die Kommunikation zwischen verschiedenen Komponenten basiert auf dem Event-Modell von Java 1.1.
- *Persistence:* Der Zustand einer angepaßten Komponente kann persistent gespeichert werden. Dabei wird alles abgespeichert, was zur Wiederherstellung von Aussehen und Verhalten benötigt wird. Referenzen auf andere Beans werden jedoch nicht mit abgespeichert. Dies ist die Aufgabe des Containers.

Obwohl es auch unsichtbare Beans geben kann, haben JavaBeans einen stark visuell geprägten Charakter. So müssen sich denn auch unsichtbare Beans visuell individualisieren lassen. Es macht daher keinen Sinn, sämtliche Klassenbibliotheken in JavaBeans zu verwandeln. Sie dienen vielmehr den Programmierern von Beans und anderer Software als Grundlage. Trotzdem kann eine duale Nutzung Sinn machen. Als Beispiel hierfür seien die Java Foundation Classes (Swing) genannt, in denen die GUI-Komponenten sowohl als Klassenbibliothek, wie auch als JavaBeans verwendbar sind [Java Swing 97].

Die drei wichtigsten Eigenheiten einer Beans-Komponente sind ihre individualisierbaren Eigenschaften, die bereitgestellten Methoden und die möglichen Events. Das JavaBeans-API gibt bestimmte Entwurfsmuster für Beans an. Ein Entwurfsmuster ist nach [Gamma et al. 94] eine Beschreibung von kommunizierenden Objekten und Klassen, die individuell angepaßt werden, um allgemeine Designprobleme in einem speziellen Kontext zu lösen. In der traditionellen Programmierung hat die Einführung von Datenstrukturen wie Listen, Bäumen und Schlangen allgemeine Probleme gelöst und dabei von den darunterliegenden Strukturen abstrahiert. Entwurfsmuster können nun als Anwendungsstrukturen in der objektorientierten Programmierung bezeichnet werden. Auch sie abstrahieren von den darunterliegenden Objekten [Mössenböck 96]. Entwurfsmuster dienen zum einen zur Dokumentation für menschliche Benutzer. Zum anderen ermöglichen sie die Benutzung von Werkzeugen, die damit die Struktur der Objekte erkennen können.

Das Beans-API beschreibt ein Entwurfsmuster als Konventionen für Namen und Typsignaturen für Mengen von Methoden und / oder Interfaces, die für Standardzwecke verwendet werden. Die folgenden Entwurfsmuster werden im Beans-API angegeben:

- Auf die Eigenschaften eines Beans wird nie direkt, sondern immer über entsprechende „get"- und „set"-Methoden zugegriffen:

  ```
  public <Property Type> get<Property Name>();
  public void set<Property Name>(<Property Type> a);
  ```

  Bei „read-only oder „write-only" Eigenschaften entfällt jeweils eine der beiden Methoden.

- Bei booleschen Eigenschaften wird statt der „get“-Methode die „is“-Methode verwendet:

```
public <boolean> is<Property Name>();
public void set<Property Name>(<Property Type> a);
```

- Für indizierte Eigenschaften gibt es zusätzliche Methoden, die auf einzelne Elemente des Arrays zugreifen:

```
public <Property Element> get<Property Name>(int a);
public void set<Property Name>(int a,<Property Element> b);
public <Property Type> get<Property Name>();
public void set<Property Name>(<Property Type> a);
```

- Events dienen zur Kommunikation der Beans untereinander. Für alle möglichen Events, die ein Bean initiieren kann, müssen entsprechende Methoden vorhanden sein, um Listener hinzuzufügen und zu entfernen:

```
public void add<EventListenerType>(EventListenerType a);
public void remove<EventListenerType>(EventListenerType a);
```

- Die bereitgestellten Methoden unterliegen keinerlei Einschränkungen oder Richtlinien. Wenn nicht anderweitig spezifiziert, werden sämtliche als `public` definierten Methoden bekanntgegeben.

Das JavaBeans-Komponentenmodell basiert auf der sogenannten Reflection-Schnittstelle, die es erlaubt, Klassen und Objekte innerhalb der Virtual Machine zur Laufzeit, also dynamisch, zu inspizieren. Dazu stellt das Beans API die Klasse `Introspector` zur Verfügung. Der Introspector durchsucht die Methoden des Beans, wendet die ihm bekannten Entwurfsmuster an und kann so von den Methoden ableiten, welche Properties, Events und Methoden ein Objekt zur Verfügung stellt. Die Namen der Eigenschaften und Methoden werden dabei direkt aus den Methodennamen entnommen.

Ein Bean kann auch explizit Informationen über sich zur Verfügung stellen. Hierzu dient eine Klasse, die das BeanInfo-Interface implementiert. Darin werden unter anderem die Properties, Events und Methoden aufgelistet. Dabei ist es möglich, daß eine Klasse nur einige Informationen zur Verfügung stellt. Die restlichen Informationen werden dann über die Reflection-Schnittstelle ermittelt.

Der Introspector stellt beim Analysieren eines Beans zunächst fest, ob eine BeanInfo-Klasse zur Verfügung steht. Gegebenenfalls untersucht er diese, um Informationen über das Bean zu erhalten. Danach beschafft er sich die restlichen Informationen über die Introspection.

Die oben als Eigenschaft eines Beans geforderte Persistenz wird durch die seit Java 1.1 vorhandene Objekt-Serialisierung bereitgestellt. Zum Einlesen einer Serialisierung enthält die Klasse `java.lang.Beans` eine Methode `instantiate()`, die das Deserialisieren eines Beans übernimmt.

Für Beans gelten die in Java bestehenden Sicherheitsmechanismen. Läuft ein Bean innerhalb eines Applets, dem nicht vertraut wird, so gilt auch das Bean als nicht vertrauenswürdig. Entwickler sollten daher immer davon ausgehen, daß ihre Beans nur über eingeschränkte Rechte verfügen.

Beans werden in einem *Java Archive* (Jar) verpackt. Zu einem solchen Jar gehört eine sogenannte Manifest-Datei, die angibt, welche Dateien und Beans sich in dem Jar befinden. Im folgenden Codebeispiel enthält das Jar ein Bean, daß durch die Startklasse `Juggler.class` repräsentiert wird. Ein Bean erhält im Manifest den Eintrag `Java-Bean: True`.

```
Manifest-Version: 1.0

Name: sunw/demo/juggler/Juggler.class
Java-Bean: True
Digest-Algorithms: SHA MD5
SHA-Digest: HvNgDbu0tEItNQrN2FxtnLHUB/g=
MD5-Digest: lodCaNW4vjtpiyVtqQojAg==

Name: sunw/demo/juggler/JugglerBeanInfo.class
Digest-Algorithms: SHA MD5
SHA-Digest: aVi52xkXvbrqrBBkW4lmI9GJvSo=
MD5-Digest: cy2MF8RT8c8AncXB7ZKtVA==

Name: sunw/demo/juggler/Juggler0.gif
Digest-Algorithms: SHA MD5
SHA-Digest: BoXVBkl+aKR7/2+f80rqxYbltTc=
MD5-Digest: SOLrOrGbrm+3aJNgJgIwdQ==
```

Außerdem enthält das Jar noch eine Klasse `JugglerBeanInfo.class`, durch die man weitere Informationen über das Bean erhält. Obwohl Beans normalerweise als Jars ausgeliefert werden, ist es nicht vorgeschrieben, daß sie sich in Jar-Dateien befinden müssen. Auch ein Manifest zur Auflistung des Inhalts einer Jar-Datei und eine Klasse BeanInfo zur Beschreibung des Beans sind nicht zwingend erforderlich, wobei die BeanBox jedoch ein Jar mit einem Manifest zum Einlesen der Beans benötigt.

3.3.1 Enterprise JavaBeans

JavaBeans wurden als ein lokales Komponentenmodell entworfen. Zur Zeit arbeitet Sun an einer Komponentenarchitektur zur Entwicklung verteilter objektorientierter Anwendungen. Diese Plattform vereinigt verschiedene APIs, wie Objekt-Serialisierung, Java Database Connectivity (JDBC), Server-seitige Programme (Servlets) und die JavaBeans-Architektur [Pountain & Montgomery 97].

Ein Enterprise JavaBean ist eine verteilte transaktionsorientierte Komponente. Eine Instanz des Beans wird zur Laufzeit von einem Container erstellt und verwaltet. Die Umgebungsvariablen können zum Einsatzzeitpunkt angepaßt werden. Metadaten, wie der Transaktionsmodus oder Sicherheitsattribute, werden von der Bean-Klasse getrennt und können durch den Container manipuliert werden. Clients greifen über den Container auf das Enterprise JavaBean zu.

Das Enterprise Beans API unterscheidet in der Architektur sechs verschiedene Rollen und definiert Verträge, die sicherstellen, daß die Produkte jeder Rolle kompatibel sind. Der Enterprise Bean Provider entwickelt wiederverwendbare Komponenten. Dabei braucht er sich nicht mit systemnahen Funktionen wie Transktionsmanagement, Sicherheit, Zustandsverwaltung, Multithreading und Ressourcenverwaltung zu beschäftigen. Diese Dienste werden vom Container zur Verfügung gestellt. Auf Wunsch kann der Entwickler jedoch darauf zugreifen [Java Enterprise 98]. Das Enterprise Bean wird in einer ejb-jar-Datei zur Verfügung gestellt.

Der Application Assembler entwickelt Anwendungen, die Enterprise JavaBeans verwenden. Die Implementation der EnterpriseBeans bleibt ihm dabei verborgen. Der *Deployer* paßt die Beans und den Container an ihre Umgebung an. Dazu verwendet er die vom Container bereitgestellten Werkzeuge. Der EJB Server Provider stellt mit dem Server die Systemumgebung zur Verfügung. Dazu gehören Transaktionsmanagement, verteilte Objekte und andere Systemdienste. Der EJB Container Provider entwickelt einen skalierbaren, sicheren, transaktionsfähigen, Container, in dem die Beans verwendet werden. Typischerweise stellt ein Container auch Dienste wie Versionsmanagement und Persistenz zur Verfügung. Ein Container stellt einem Bean eine einfache standardisierte Schnittstelle zur Verfügung, die von den Serverdiensten abstrahiert. Der Client kommuniziert mit dem Container in dem sich das Bean befindet. Der Systemadministrator überwacht das gesamte System. Dazu verwendet er Werkzeuge der Servers und des Containers.

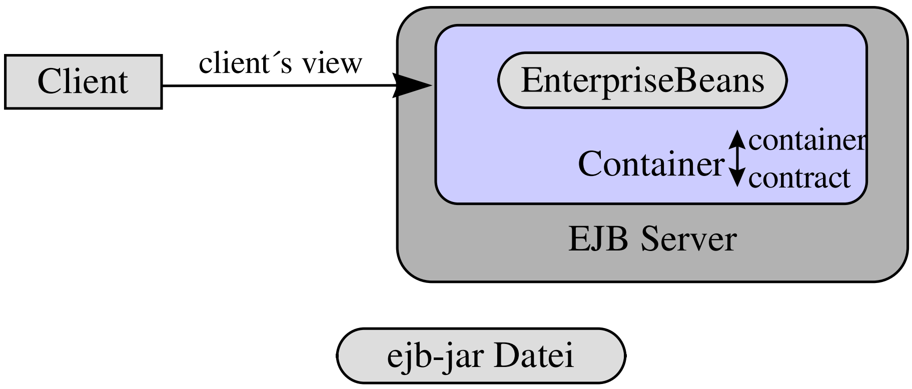

Abbildung 11: Die Enterprise JavaBeans Architektur

Aufgrund ihrer Vorteile hat die Enterprise JavaBeans-Architektur eine weite Akzeptanz gefunden. Zu diesen Vorteilen gehören die Skalierbarkeit, die Einfachheit, die Sicherheit, die Interoperabilität mit bestehenden Systemen, und das integrierte Transaktionskonzept.

Die Spezifikation der Enterprise JavaBeans ist erst im März dieses Jahres herausgegeben worden und konnte daher nicht als Grundlage für diese Arbeit dienen.

3.3.2 Bridges für JavaBeans

Eines der Hauptziele von Java ist die Plattformunabhängigkeit. Wenn nun aber ein Bean in einen plattformabhängigen Container, wie zum Beispiel MS-Office eingebettet werden soll, dann soll das API des Beans auch in die lokale Komponentenarchitektur eingegliedert werden.

Daher stellt Sun verschiedene Brücken zur Verfügung, mit denen man ein JavaBean z.B. für ActiveX und COM verwenden kann. Eine Komponentenarchitektur gewinnt an Bedeutung, je mehr Komponenten angeboten werden. Durch die Brücken können bereits existierende ActiveX und COM-Komponenten als JavaBeans verwendet werden.

Das Ziel ist, daß ein Bean in einer Vielzahl von Systemen lauffähig ist und sich dort jeweils wie eine Komponente aus der lokalen Architektur verhalten kann.

3.3.3 BeanBox

Mit der BeanBox hat Sun ein Werkzeug geschaffen, das als Testumgebung für selbst entwickelte Komponenten dient. Die BeanBox ist ein Anwendungsbeispiel für ein Werkzeug, mit dem Beans miteinander kombiniert werden können, und sie dient als Referenzumgebung für Entwicklungswerkzeuge Da die BeanBox jedoch keine eigene programmiertechnische Einflußnahme zuläßt, verhindert sie einen sinnvolleren Einsatz. Sie ist nicht dazu entwickelt, aufwendige grafische Oberflächen bequem zu erstellen [Campione & Walrath], verdeutlicht aber die Konzepte hinter den Beans.

Die BeanBox liest alle vorhandenen Beans bei Programmbeginn ein und analysiert sie. Dazu müssen sich die Beans jedoch alle in einem festgelegten Verzeichnis befinden (`bdk\jars`). Beans in anderen Verzeichnissen kann die Beanbox nicht anzeigen. Nachträglich hinzugefügte Beans müssen sich ebenfalls in diesem Verzeichnis befinden und vor Verwendung erst explizit eingelesen werden. Diese Lösung ist sehr unflexibel und ermöglicht keine hierarchische Ordnung der Beans. Eine sinnvolle Verwaltung von Komponenten in der BeanBox ist daher nicht möglich.

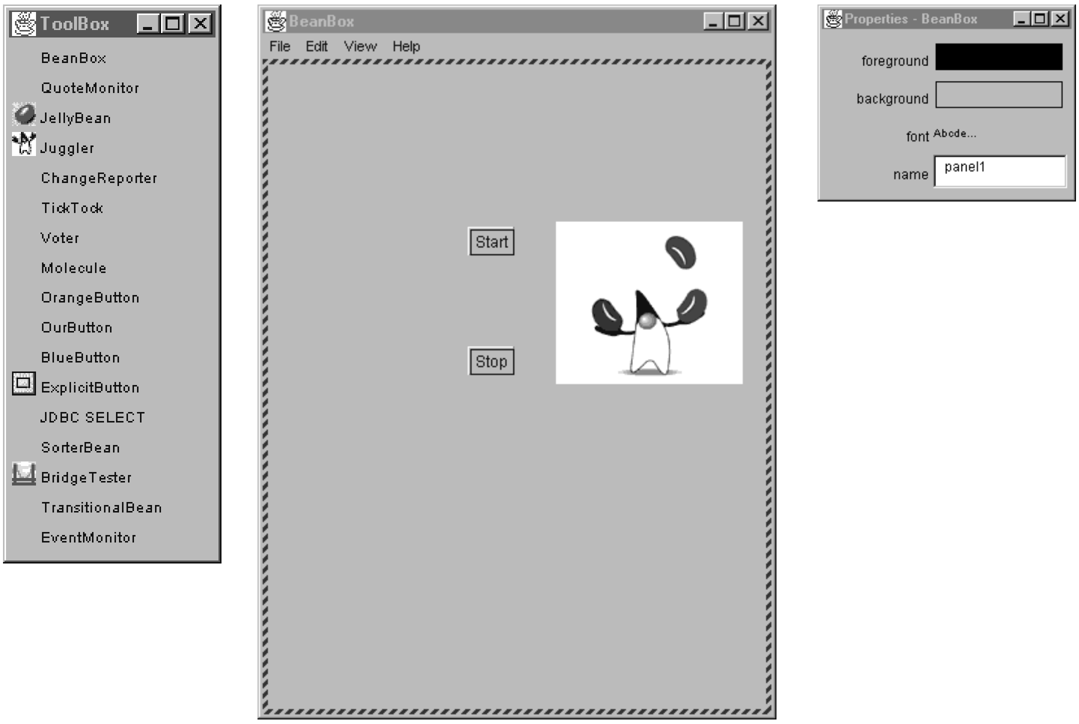

Abbildung 12: Die Java BeanBox mit der Toolbox, die alle verfügbaren Beans auflistet, dem eigentlichen Editor und einem Property-Dialog, in dem die konfigurierbaren Eigenschaften des Beans angezeigt werden

Eine Auswahl der Beans erfolgt durch „Drag and Drop“. In einem Property-Fenster werden die individualisierbaren Eigenschaften des ausgewählten Bean angezeigt. Diese wurden zuvor vom Introspector ermittelt (Introspection oder BeanInfo-Klasse). Die BeanBox bietet die Möglichkeit, einen Bericht der Introspection zu erstellen.

Für die primitiven Datentypen stellt die BeanBox Standard-Property-Editoren zur Verfügung. Ein Bean kann jedoch auch eigene Property-Editoren implementieren.

Über das Menü werden die verschiedenen Events, die ein Bean senden kann, angezeigt. Beans können nur über diese Events miteinander kommunizieren. Ein Bean (Event Target) kann sich bei einem anderen (Event Source) als EventListener für einen bestimmten Event eintragen. Dabei wird zunächst über das Menü der Event des Quell-Beans festgelegt. Danach wird das Ziel-Bean angeklickt und die Methode ausgewählt, die bei Auslösen des Events ausgeführt werden soll. Die BeanBox generiert nun eine Adapterklasse, die einen passenden EventListener implementiert. Diese wird von dem Event benachrichtigt und ruft dann die entsprechende Methode des Quell-Beans auf.

Das folgende Codebeispiel zeigt die Adapterklasse, die die Animation des „Jugglers" beim Anklicken des „Stop"-Buttons anhält:

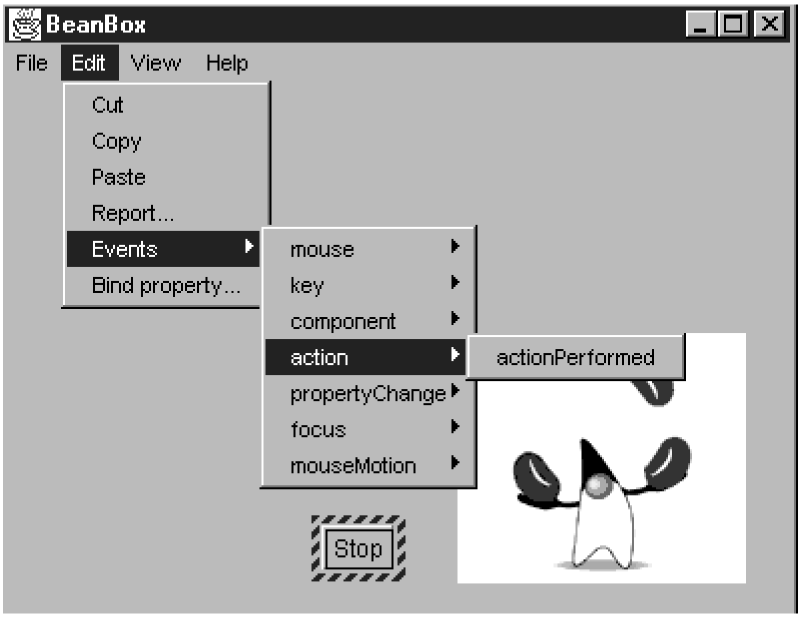

Abbildung 13: Anzeige und Auswahl eines Events der Quell-Beans

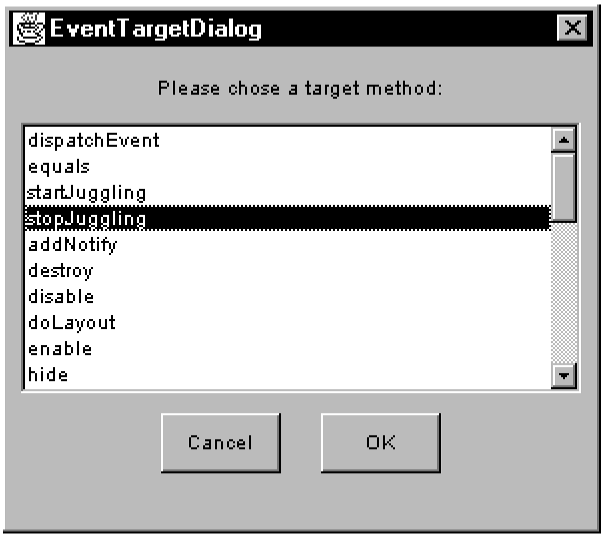

Abbildung 14: Auswahl einer Methode des Ziel-Beans

```
// Automatically generated event hookup file.
package tmp.sun.beanbox;
import sunw.demo.juggler.Juggler;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ___Hookup_14a157b09d
   implements  java.awt.event.ActionListener,
            java.io.Serializable
{

   public void setTarget(sunw.demo.juggler.Juggler t)
   {
      target = t;
   }

   public void actionPerformed(ActionEvent arg0)
   {
      target.stopJuggling(arg0);
   }

   private sunw.demo.juggler.Juggler target;
}
```

Ein Nachteil dieser Vorgehensweise ist, daß es zu einer Inflation der Klassen kommt, da zu jeder Kommunikations-Verbindung zwischen zwei Beans eine neue Klasse generiert wird.

Zum Einlesen einer Jar-Datei wird in der BeanBox die Klasse `JarLoader` verwendet. Jedes Jar wird in einer eigenen Sandbox von einem eigenen JarLoader geladen. Für das eigentliche Laden der Klassen und Ressourcen ruft der JarLoader die Klasse `SimpleClassLoader` auf. Das Laden von Ressourcen läuft bei der Beanbox über eine URL. Der SimpleClassLoader stellt die Methode `getLocalResource()` zur Verfügung, die eine `URL` in der Form `simpleresource://null/hashcode/+/name` zurückgibt. Mit `URL.getContent()` wird dann die Ressource übergeben. Dafür ist eine selbst implementierte `SimpleResourceConnection` zuständig. Die genaue Funktionsweise dieses Mechanismus wird in Kapitel 7 erläutert.

Wie oben bereits erwähnt wurde, ist ein Nachteil der BeanBox, daß alle Beans sich in Jar-Dateien in einem festgelegten Verzeichnis befinden müssen. Diese Lösung ist sehr unflexibel. Zum einen können später nach dem Programmstart keine Beans mehr hinzugefügt werden, ohne diese explizit einzulesen. Zum anderen ermöglicht dieser Ansatz keine hierarchische Strukturierung der Beans. Eine Verwaltung ist daher nicht sinnvoll möglich. Außerdem werden alle Beans zum Programmstart eingelesen, egal ob sie benötigt werden oder nicht.

Die BeanBox ist daher für einen Einsatz in HADES ungeeignet, liefert aber einige gute Ansätze für spezielle Probleme, die wir später aufgreifen werden.

3.3.4 Java Studio

Anhand des ebenfalls von Sun entwickelten Java Studio soll hier deshalb kurz der Umgang mit einem professionelleren Werkzeug dargestellt werden. Java Studio ist ein Entwicklungswerkzeug, das nicht nur Programmierern, sondern auch Endanwendern ermöglichen soll, JavaBeans zu einer Applikation zusammenzufügen. Dabei bietet es nach [Java Studio 97] eine intuitive Entwicklungsumgebung, ein GUI-Fenster, in dem die Komponenten „live“ hinzugefügt und getestet werden können und eine Reihe mitgelieferter Beans. Es besteht außerdem die Möglichkeit, neue Beans hinzuzufügen und eigene Beans zu exportieren. Die mitgelieferten Beans decken die Bereiche Multimedia, Datenbankzugriff, Visualisierung, Berechnung und GUI ab.

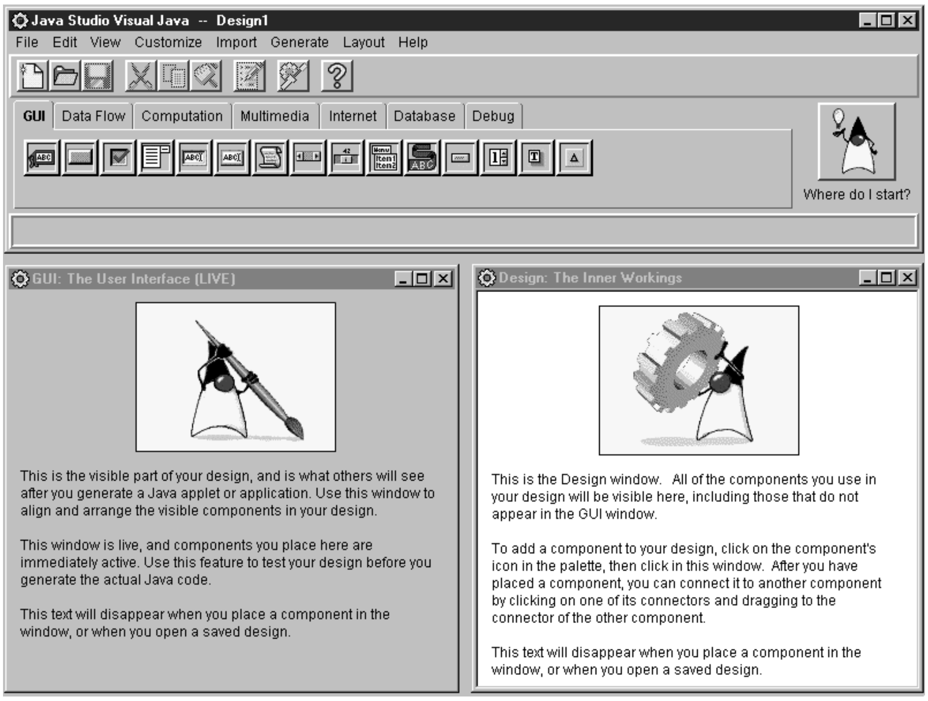

Abbildung 15: Java Studio Screenshot

Java Studio besteht aus einem Hauptfenster mit Menüs, Toolbars und einer Palette von Komponenten. Zur dieser Palette können neue Beans hinzugefügt werden. Auf der linken Seite sieht man das GUI-Fenster, in dem die Komponenten graphisch zusammengestellt werden. Im rechten Fenster kann dann Funktionalität hinzugefügt werden, indem Komponenten verbunden werden.

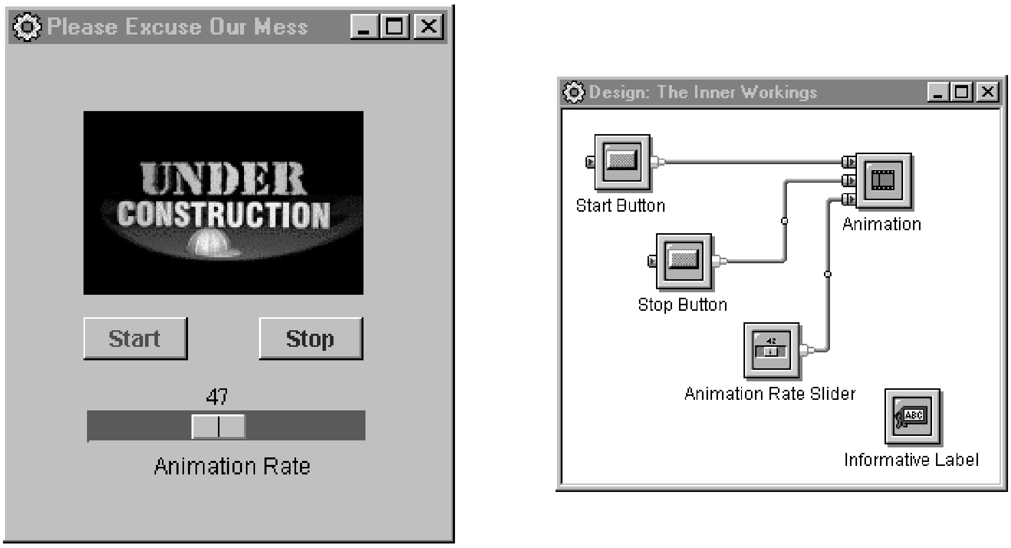

Abbildung 16: Anwendungsbeispiel Java Studio

Im gezeigten Anwendungsbeispiel sind bereits verschiedene Beans in das GUI-Fenster eingefügt und im Design-Fenster verknüpft worden. Das Ergebnis läßt sich als neues Bean abspeichern und in die Palette einfügen.

Schon bei dieser kurzen Darstellung des Java Studio werden die Unterschiede zur Bean-Box deutlich und zeigen auf, welche Möglichkeiten integrierte Entwicklungswerkzeuge bieten können.

3.4 Vergleich von JavaBeans mit anderen Architekturen

Im Rennen um die zukünftige Vorherrschaft der Komponentenarchitekturen stellt sich die Frage, wer die besseren Voraussetzungen dafür haben könnte. Nachdem OpenDoc offiziell eingestellt wurde und IBM sich auf die JavaBeans anstatt das eigene SOM konzentriert, können diese beiden bereits als ausgeschieden angesehen werden. Bleiben nur noch die JavaBeans und COM. Durch seine Integration in allen Windows-Plattformen kann COM als derzeit am weitesten verbreitete Architektur angesehen werden. Nahezu sämtliche kommerzielle Produkte dieser Plattformen unterstützen inzwischen OLE/COM und bieten Programmierern Zugriff auf unzählige interne Komponenten. Alleine das MS-Offfice 97 stellt ca. 500 Objekte [Maslo 97] bereit, die nicht nur intern mit VBA (*Visual Basic for Applications*), sondern auch extern verwendet werden können. C++ Programmierern erleichtern die MFC (*Microsoft Foundation Classes*) die Komponenten-Erstellung, und Visual Basic 5.0 erledigt sämtlichen anfallenden Verwaltungsaufwand völlig selbständig. Gegen diese Voraussetzungen hat es jede noch so überlegende Architektur schwer.

JavaBeans stehen dagegen erst am Anfang, erfordern jedoch keine grundlegende Systemunterstützung und sind plattformunabhängig. Die Verfügbarkeit von Java Interpretern auf praktisch allen Plattformen ermöglicht ein absolut übergreifendes Modell selbst auf binärer Ebene. Alle anderen Architekturen ermöglichen es zwar, mit Objekten anderer Plattformen zu kommunizieren; eine Wiederverwendung in binärer Form auf unterschiedlichen Systemen ist jedoch nur mit Java gegeben. Die Tatsache, daß IBM viele seiner Aktivitäten auf Java konzentriert hat, zeigt daß JavaBeans in Zukunft einen wichtigen Stellenwert erlangen könnten. Die Entwicklung der Enterprise JavaBeans bietet die für den kommerziellen Einsatz wichtigen Voraussetzungen wie z.B. Transaktionsunterstützung. Das entsprechende Microsoft Pendant dazu ist COM+, das ganz ähnliche Erweiterungen vornimmt. [Moeller 98] nennt als Quelle der COM Ergänzungen sogar explizit die Entwicklung der Enterprise Beans.

Weiterer Pluspunkt der JavaBeans ist die von Sun angebotene ActiveX-Bridge, die eine Verwendung von Beans als COM-Objekte ermöglicht. Sie kann zumindest für nicht allzu komplexe Komponenten einen Architekturwechsel glätten. Größter Nachteil von Java ist und bleibt (zumindest im Moment noch) die relativ schlechte Performanz, insbesondere im graphischen Bereich. Für anspruchsvolle Benutzeroberflächen bietet Java inzwischen die JFC, aber auch deren Performanz ist eher mäßig. Angesichts immer schnellerer Prozessoren und üppigeren Speicherausstattungen wird dieser Nachteil jedoch früher oder später in den Hintergrund rücken.

4 HADES

Ziel dieses Kapitels ist es, das HADES Framework, in das der Komponenten-Browser integriert werden soll, vorzustellen. Nach einem Überblick über Hintergrund und Ziele des Projekts in Kapitel 4.1, wird in Kapitel 4.2 dazu auch auf die genauere Funktionsweise des Systems eingegangen. Kapitel 4.3 beschreibt die Verwendung von JavaBeans in HADES. Um die besonderen Vorteile von HADES besser verstehen zu können, werden in Kapitel 4.4 noch verschiedene andere Simulationssysteme vorgestellt. Das abschließende Kapitel 4.5 enthält eine Zusammenfassung der Ergebnisse und einen Vergleich der vorgestellten Systeme.

4.1 Übersicht

An der Universität Hamburg existiert am Arbeitsbereich TECH der Informatik schon lange der Wunsch nach interaktiver Software zur Visualisierung und Erprobung der Lehrinhalte. Kommerzielle CAD-Software zum Schaltungsentwurf besteht zumeist aus sehr teuren und komplexen Paketen, die sich nicht für Einsteiger eignen. Freie Software in diesem Bereich, mit der die Studenten auch zu Hause experimentieren können, ist zwar im allgemeinen benutzerfreundlicher, besitzt dafür aber zumeist einen recht geringen Funktionsumfang (siehe hierzu auch Abschnitt 4.4).

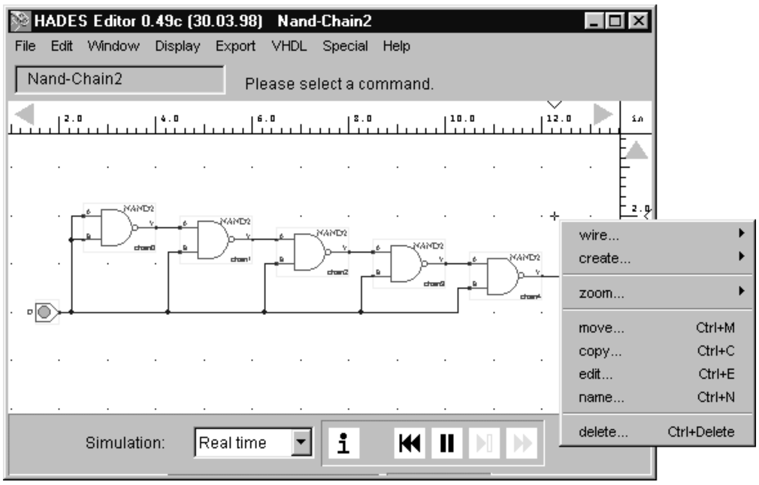

Abbildung 17: HADES Editor

Aus diesem Bedarf heraus entstand das „Hamburger Designsystem“ (HADES). HADES ist ein Java-Framework zur objektorientierten Simulation, das durch eine offene Struktur und einen größtmöglichen Anwenderkreis die Lehre im Bereich der technischen Informatik unterstützt. Dabei werden alle relevanten Entwurfsebenen von der Gatterebene bis zur Systemebene berücksichtigt. Im Rahmen von Studien- und Diplomarbeiten zuvor entwikkelte Java Applets z.B. für KV-Diagramme und endliche Automaten (FSMs) konnten mit in das System integriert werden.

Da HADES in der Lehre eingesetzt werden soll, entsteht die folgende grundlegende Anforderung an das System: Es muß von praktisch jedem Studenten einsetzbar sein. Daraus resultiert zum einen, daß HADES intuitiv bedienbar, also ohne längere vorherige Lernphase anwendbar sein muß. Zum anderen muß das System kostenfrei sein, damit es von den Studenten auch zu Hause benutzt werden kann. Durch diesen Aspekt entfällt leider die Möglichkeit, auch kostenpflichtige Technologien wie z.B. kommerzielle CORBA-Implementationen in HADES zu verwenden. Weiterhin soll das System möglichst Plattformunabhängig sein, um von den Studenten auch Zuhause auf unterschiedlichsten Rechnern eingesetzt werden zu können.

4.2 Architektur

HADES basiert auf eigenständigen Simulationsobjekten, die als JavaBeans realisiert werden. Diese Beziehung wird in Kapitel 4.3 näher erläutert. Der Einsatz von JavaBeans ermöglicht neben der durch Java gegebenen Plattformunabhängigkeit auch eine optimale Erweiterbarkeit des Systems.

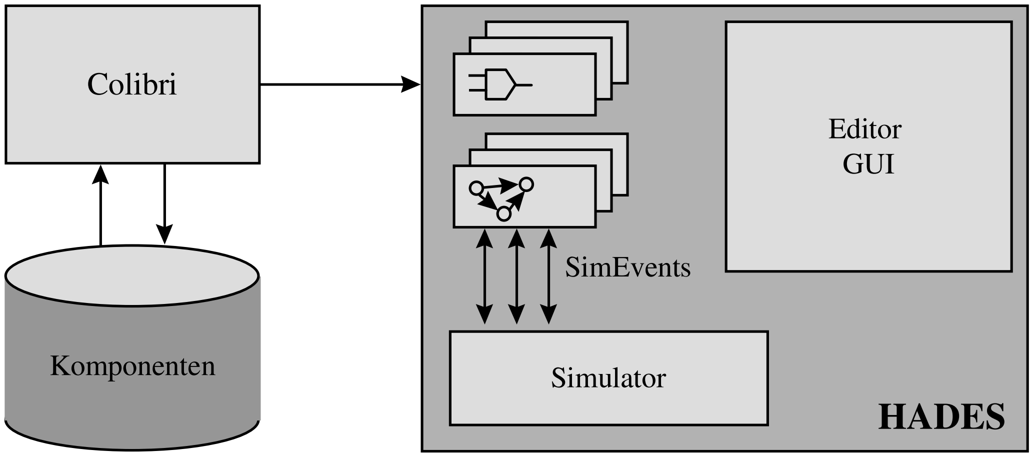

Abbildung 18: HADES-Architektur

Die Verwendung einer eigenen Simulationssprache wäre zwar ebenfalls Plattformunabhängig, aber in den Möglichkeiten stark eingeschränkt. JavaBeans stehen dagegen sämtliche Ressourcen offen (Netzwerk, Grafik, GUI, etc.), wodurch sich vielfältigste Anwendungsmöglichkeiten ergeben. Da die die Simulationsobjekte verbindenden Signale ebenfalls als eigenständige erweiterbare Java-Klassen realisiert sind, löst sich HADES auch vom rein technischen Einsatz. So kann das System mittels der bereits existierenden Image-Signale z.B. auch in der Bildverarbeitung oder Mustererkennung Anwendung finden.

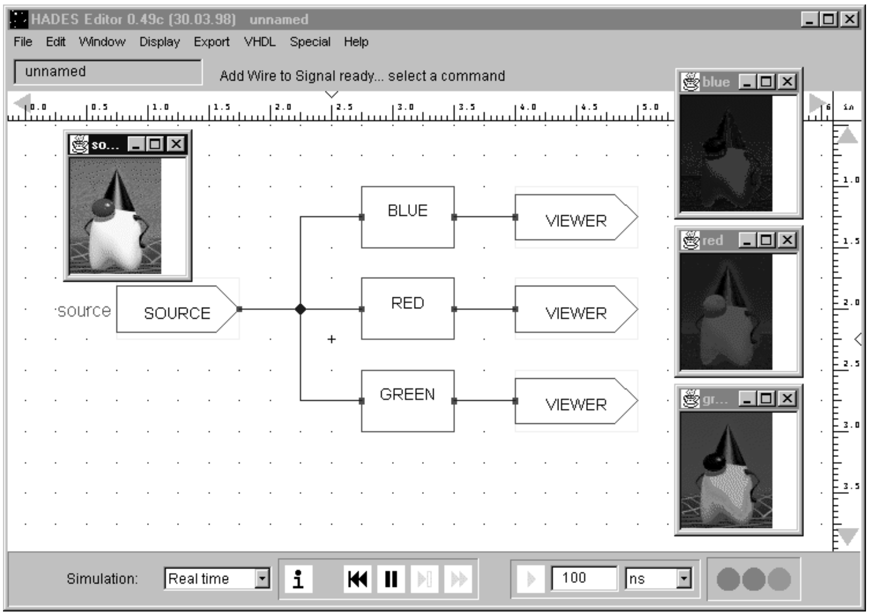

Abbildung 19: Einfache Bildverarbeitung in HADES

Der in HADES verwendete Simulationskern ist Event-gesteuert. Dies bedeutet, daß bei einer Signal-Änderung die daran angeschlossenen Objekte durch einen Event benachrichtigt werden. Diese nehmen daraufhin eine Evaluierung ihrer Ausgänge vor und generieren bei Bedarf neue Events für andere Simulationsobjekte. Dieser Ansatz reduziert den erforderlichen Rechenaufwand erheblich, setzt dafür aber auch eine gewisse Intelligenz der Komponenten voraus. Die Realisierung der Simulationsobjekte als JavaBeans bringt also auch hier Vorteile.

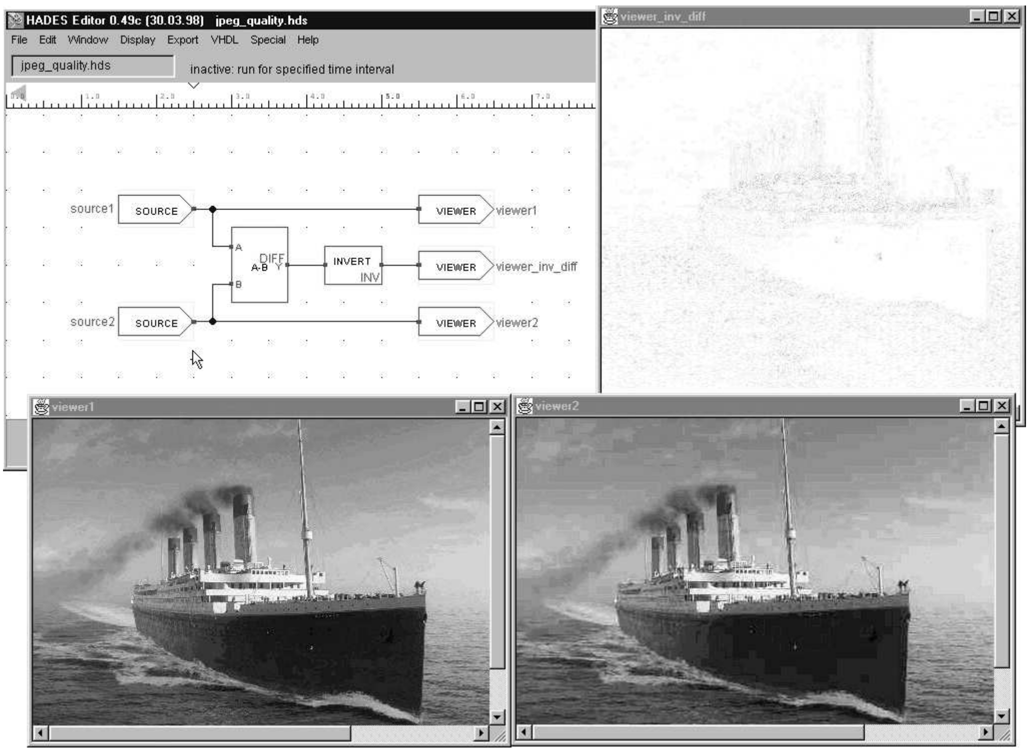

Abbildung 20: Anspruchsvolle Bildverarbeitung: Differenzbild (rechts oben) eines JPEG-Bildes mit hoher Qualität (links) und einem JPEG-Bild mit geringer Qualität (rechts unten)

Ein wichtiger Aspekt bei HADES sind hierarchische Designs. Dabei können Designs in andere Designs eingefügt und dort wie normale Komponenten verwendet werden. Dies ist von besonderer Bedeutung, wenn Teile eines Designs entweder mehrfach vorkommen oder anderweitig Verwendung finden können. So könnte ein 32-Bit-Addierer z.B. aus vier 8-Bit- oder 32 1-Bit-Addierern aufgebaut werden. Die Verwendung separater Funktionsblöcke verbessert die Übersichtlichkeit und verkürzt die Entwurfszeit. Unterdesigns können weiterhin einzeln getestet werden und erleichtern so die Fehlersuche. HADES zeigt enthaltene Designs normalerweise als „Blackbox“ an. Für häufig verwendete Komponenten können jedoch auch beliebige graphische Repräsentationen erstellt werden. Da die Eigenschaften solcher Komponenten durch das enthaltene Subdesign bestimmt werden, öffnet der Menüpunkt „edit“ des Kontextmenüs konsequenterweise ein neues Editor-Fenster, in welchem das Subdesign verändert werden kann. Normale Komponenten öffnen hierbei ein Property-Fenster, in dem Eigenschaften, wie z.B. der Name veränderbar sind.

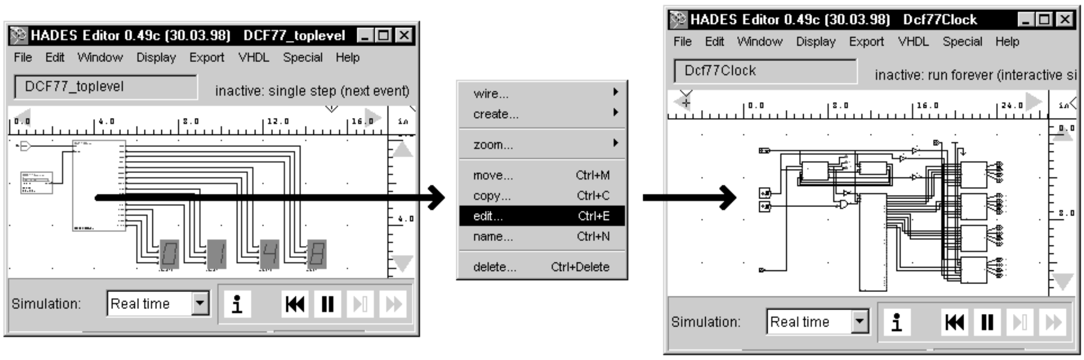

Abbildung 21: Hierarchisches Design

Die Simulation in HADES kann sowohl interaktiv, als auch Skript-gesteuert stattfinden. Im interaktiven Simulationsmodus kann der Benutzer die Eingänge direkt mit der Maus manipulieren und das Verhalten des Systems daraufhin beobachten. Ein sogenannter „Glow Mode“ stellt die aktuellen Zustände aller Signale farbig dar und erlaubt es, Veränderungen auf den Leitungen direkt zu beobachten. Dieser Modus geht jedoch deutlich zu Lasten der Performanz und sollte bei komplexeren Designs deaktiviert werden. Der Schwerpunkt der Skript-Steuerung liegt in der Echtzeit-Simulation, bei der z.B. Gatterlaufzeit eine Rolle spielen können. Da die Simulationsobjekte bei HADES vollen Zugriff auf den eigentlichen Simulationskern und die zugehörigen Datenstrukturen besitzen, sind jedoch auch benutzerdefinierte Simulationsmodi denkbar.

Durch die Ausrichtung von HADES auf die Vermittlung von Lehrinhalten, haben alle bisherigen Simulationsobjekte einen stark visuell geprägten Charakter. Durch die Realisierung als JavaBeans, die vollen Zugriff auf alle graphischen Ressourcen genießen, sind hier kaum Grenzen gesetzt. Andere Simulationssysteme, die interne Simulationssprachen verwenden, müssen hier passen. Durch die mögliche farbige Darstellung der Signale (Glow Mode), können nicht nur das Verhalten einzelner Komponenten, sondern auch Abläufe innerhalb von ganzen Systemen visualisiert werden.

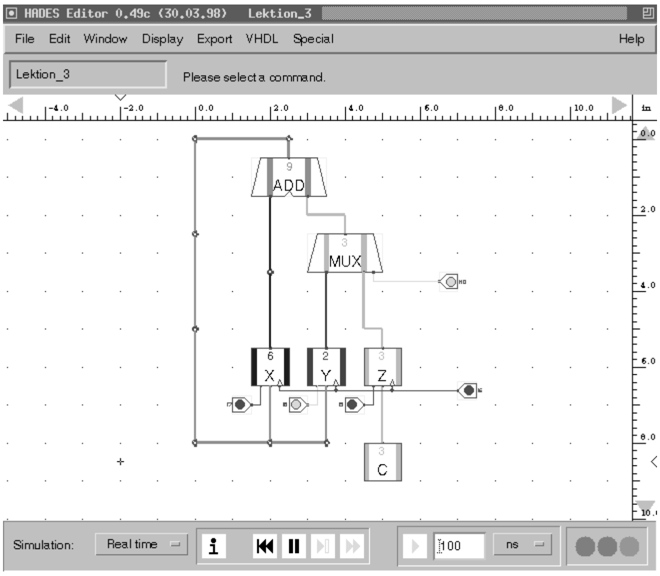

Abbildung 22: Visueller "Rechnerbaukasten" in HADES

Um den gesamten Simulationsverlauf darzustellen, bietet HADES ein Impulsdiagramm, welches alle momentan verfügbaren Signaltypen unterstützt. Neue Signaltypen können nachträglich mittels einer „Waveform-Factory" integriert werden. Für die Zukunft ist mit „*Smart Waves*" noch eine verbesserte Version in Planung, die den Entwickler, u.a. durch komplexe Suchfunktionen schon beim Entwurf besser unterstützten soll.

Zum Zeitpunkt dieser Arbeit enthält HADES bereits Signale vom Typ StdLogic1164, Integer, String und Image. Durch die Möglichkeit der einfachen Integration neuer Signale werden in Zukunft jedoch viele weitere Signaltypen mit passenden Simulationsobjekten verfügbar sein.

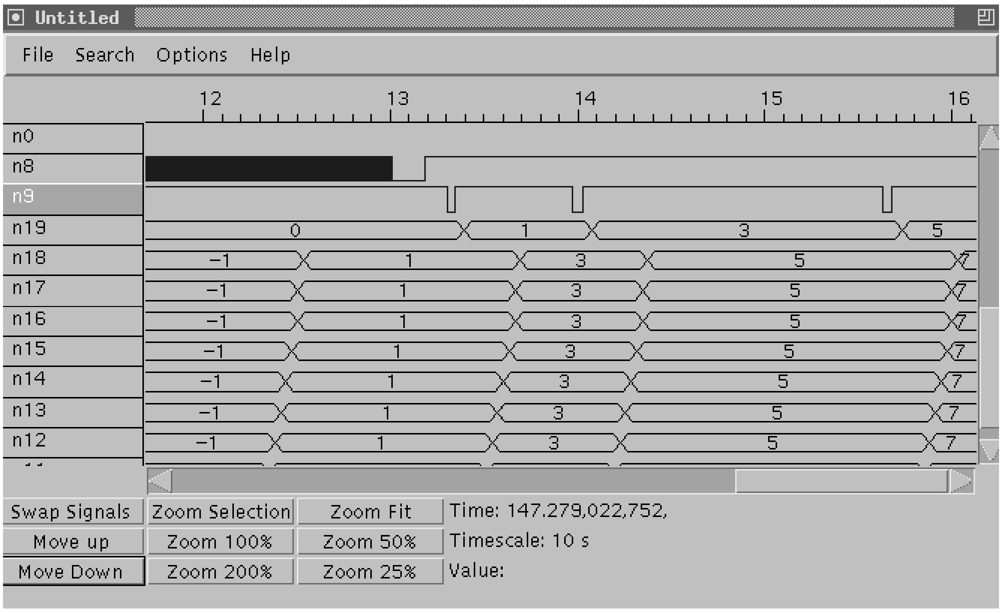

Abbildung 23: HADES-Impulsdiagramm

Um HADES nicht nur zum „Herumspielen“ verwenden zu können, sollen zukünftig auch diverse Exportmöglichkeiten unterstützt werden. Am Bemerkenswertesten dürfte dabei der VHDL-Export sein, bei dem erstellte Designs in die weitverbreitete Entwurfs- und Simulationssprache VHDL konvertiert werden können. Um dies mit beliebigen Komponenten (Gatter, FSMs, etc.) zu ermöglichen, muß jedes Simulationsobjekt sich selbst konvertieren können. HADES fügt dann die einzelnen Teile zu einem Gesamtentwurf zusammen.

Durch das große Spektrum an Einsatzmöglichkeiten von HADES ist für die Zukunft eine große Menge unterschiedlichster Komponenten zu erwarten. Im Rahmen dieser Diplomarbeit soll deshalb ein Komponenten- und Bibliotheks-Browser entwickelt werden, der die Verwaltung aller Simulationsobjekte und Designs übernimmt. Dazu zählen die zu Beginn dieser Arbeit genannten Punkte, wie Versionsmanagement oder weltweite Suche. Bisher bieten lediglich große kommerzielle Entwicklungssysteme wie das in Abschnitt 4.4 vorgestellte „Cadence“ die Möglichkeit, ihre Simulationsmodelle strukturiert zu verwalten. Einfachere Werkzeuge verwalten ihre Modelle dagegen meist nur in einer flachen Liste.

4.3 HADES und JavaBeans

Wie oben beschrieben, bringt die Realisierung der Simulationsobjekte als eigenständige Komponenten in Form von JavaBeans für HADES entscheidende Vorteile. Dazu gehören:

- Plattformunabhängig durch Java
- Beliebig komplexe Komponenten möglich
- Simulationsobjekte können individualisiert werden
- Größtmögliche Erweiterbarkeit
- Durch abstrakte Signale nicht an Schaltkreise gebunden

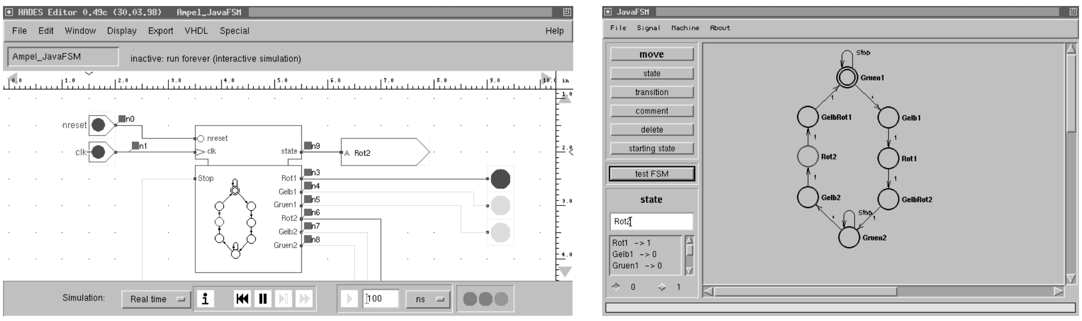

Abbildung 24: Endlicher Automat (FSM) als komplexe Komponente mit eigenem Editor

Bei näherer Betrachtung der HADES-Simulationsmodelle stellte sich jedoch heraus, daß es keine 100-prozentigen JavaBeans sind, da sie alle von einer Klasse aus dem HADES-Package erben müssen. Solche Abhängigkeiten sind für echte JavaBeans nicht erlaubt, da sie ansonsten ihren eigenständigen Charakter (siehe Kapitel 3) verlieren würden. Wollte HADES absolut Bean-kompatibel sein, dürfte es jede Komponente nur über das Reflection-API ansprechen, was einen erheblichen Performanzverlust zur Folge hätte. Durch die von HADES gewählte Alternative wird dieser Overhead zwar vermieden, dafür verlieren die Simulationsobjekte aber ihre JavaBeans-Kompatibilität.

Natürlich sind auch JavaBeans streng genommen nicht wirklich unabhängig. Auch sie erben definitionsgemäß zumindest von der Systemklasse `java.lang.Object`, benutzen aber zusätzlich fast immer auch noch weitere Klassen aus der Java Systembibliothek. Der Unterschied für HADES-Komponenten liegt darin, das ihre gemeinsame Basisklasse außerhalb dieser Bibliothek liegt, wodurch sie ohne das HADES-Package nicht ladbar sind. Diese Problematik wird in Kapitel 7 unter dem Titel „ClassLoader Problematik“ näher untersucht.

Ein weiterer Aspekt der HADES-Beans (der jedoch COLIBRI nicht betrifft) ist das verwendete Event-Modell. Das von JavaBeans normalerweise verwendete Modell garantiert keine definierte Reihenfolge der Event-Auslieferung. Für eine präzise Simulation ist eine gleichbleibende definierte Benachrichtigung jedoch Voraussetzung. Aus diesem Grund basiert HADES auf einem eigenen Event-Modell, das durch den Simulationskern gesteuert wird. Auf die Weise kann auch in einem möglichen Multithreading-Betrieb eine zuverlässige Synchronisation sichergestellt werden.

Vor der Verwendung von COLIBRI verwaltete HADES seine Simulationsmodelle in Unter-Packages, die vom normalen CLASSPATH abgedeckt wurden. Die bereits verfügbaren Komponenten waren hierarchisch in ein Kontextmenü einprogrammiert. Um weitere Komponenten laden zu können, gab es einen „Create-by-name“-Dialog, in dem die zugehörige Hauptklasse über ihren vollständigen Namen manuell ausgewählt wurde.

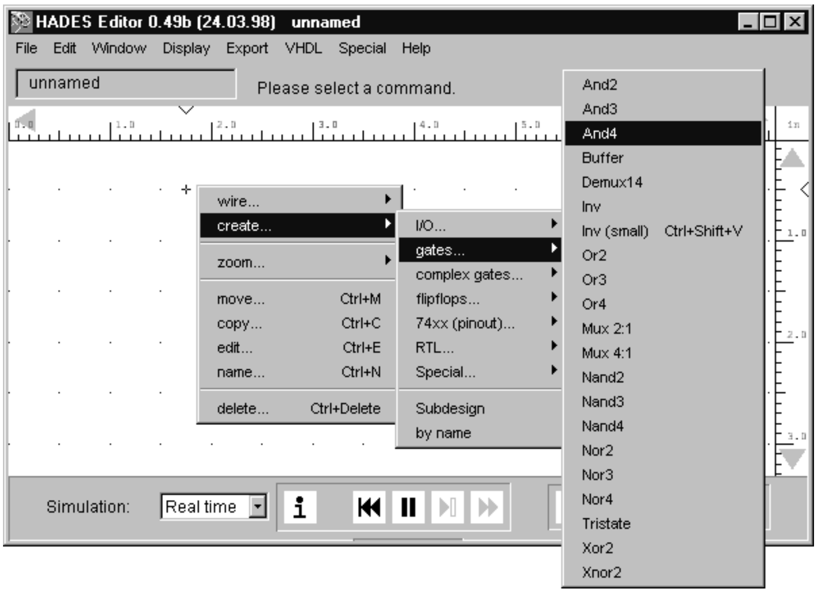

Abbildung 25: Ursprüngliches HADES: Auswahl der Komponenten in Kontextmenüs

4.4 Vergleich von HADES mit anderen Werkzeugen

Nach der Vorstellung von HADES sollen in diesem Kapitel einige andere Simulationssysteme vorgestellt und mit HADES verglichen werden. Da sich HADES prinzipiell auf den technischen Bereich konzentriert, wurden die zu vergleichenden Systeme ausschließlich aus diesem Bereich gewählt.

4.4.1 DigLOG / AnaLOG

LOG ist ein frei verfügbares Simulationswerkzeug der Universität Berkeley, das im C-Quellcode vorliegt [Lazzaro 97]. Es basiert auf den X11-Grafikbibliotheken und sollte auf den meisten Unix Plattformen laufen. Anfangs beherrschte LOG nur die Simulation digitaler Schaltungen, wurde später aber um analoge Fähigkeiten erweitert. Neben dem reinen Simulator werden auch einige Zusatzprogramme mitgeliefert, die z.B. die Erstellung von Netzlisten ermöglichen. Ebenfalls enthalten ist eine Bibliothek von Bauteilen, die in einer separaten Datei abgelegt ist. Wer weitere Bauteile benötigt, kann diese in einem externen Editor erstellen. Das Verhalten solcher Elemente wird dabei mittels einer einfachen integrierten Sprache spezifiziert. Für komplexere Elemente besteht weiterhin die Möglichkeit auf gesonderte C-Prozeduren zurückzugreifen.

Durch die Erweiterungsmöglichkeiten ist das Einsatzspektrum von DigLOG und AnaLOG bereits sehr groß. Die feste Integration der Signaltypen beschränkt die Anwendungen jedoch auf technische Bereiche. Durch die Veröffentlichung im Quellcode ist die Portabilität zumindest auf Unix Systemen gegeben. Unerfahrenen Benutzern kann die anfängliche Compilierung und das dazu nötige Anpassen der „Make“-Dateien jedoch beträchtliche Probleme bereiten. Die Bedienung des Simulators ist dann wiederum recht einfach. Allerdings ist sein Funktionsumfang auch recht begrenzt. Die interaktive Simulation von LOG diente HADES in vielen Punkten als Vorbild [HADES 97]. Dabei werden Signale anstelle von Simulationsskripten direkt mit der Maus manipuliert und die Veränderungen daraufhin sofort angezeigt. Der in LOG vorhandene „Glow Mode“, bei dem sämtliche Leitungen ihren Status farbig anzeigen, diente HADES ebenfalls als Vorbild.

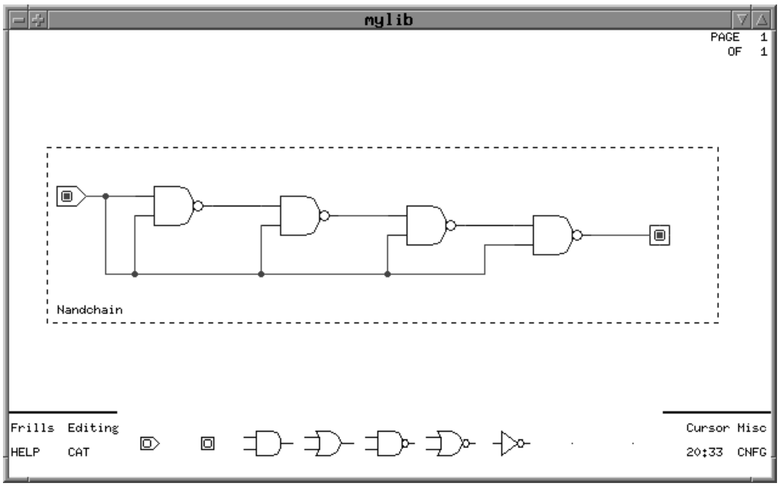

Abbildung 26: Digitales Design in LOG

LOG beherrscht wie HADES weiterhin hierarchische Entwürfe, dies allerdings nur für digitale Schaltungen. Zur besseren Übersicht verwaltet LOG neun parallele Entwurfsseiten, die auch zur Erstellung hierarchischer Designs genutzt werden können. Die Standardgatter können direkt per „Drag and Drop“ aus dem unteren Bereich des Fensters in das Design eingefügt werden. Alle anderen Simulationsobjekte werden in einer eigenen Bibliotheksansicht (siehe Abbildung) ausgewählt: Alle Bauteile, die in dieser Bibliotheksansicht erscheinen sollen, müssen vorher aus einer einfachen Namensliste ausgewählt werden.

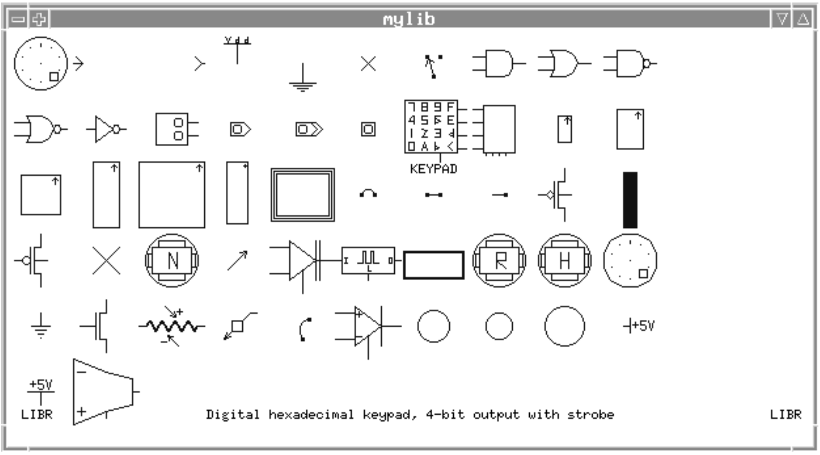

Abbildung 27: Bibliotheksansicht bei LOG

Neue Bauteile werden in einem gesonderten Editor erstellt. Zur graphischen Repräsentation stellt dieser mehrere Vektor-orientierte Zeichenoperationen zur Verfügung. In diesem graphischen Editor werden ebenfalls alle bereitgestellten Anschlüsse (Pins) positioniert. Das Verhalten des Bauteils wird (wie oben erwähnt) in einer integrierten GDL (*Gate Definition Language*) definiert. Diese unterstützt neben Variablen und Wertzuweisungen auch eine *CALL*-Anweisung, mit der auf native C-Prozeduren zurückgegriffen werden kann. Die von den mitgelieferten Bauteilen (Switch, LED, 7-Segment, etc.) benötigten Prozeduren befinden sich direkt im Simulator-Quellcode.

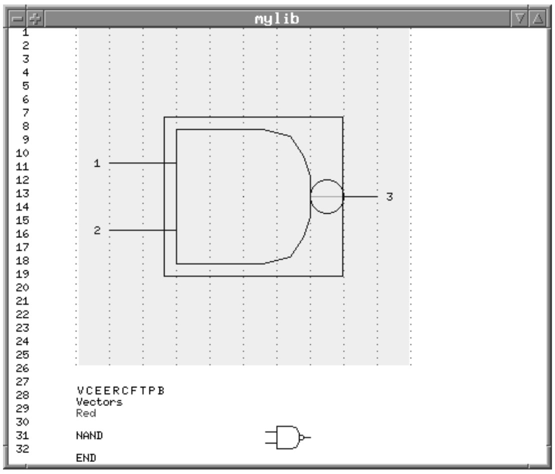

Abbildung 28: Bauteil-Editor

Bei der Erstellung komplexerer Bauteile wird der Vorteil des HADES-Konzeptes deutlich. Es fällt zwar bei beiden Systemen Programmieraufwand an, die Realisation als eigenständige Komponenten vermeidet jedoch eine gegenseitige Verflechtung und ermöglicht so eine einfache Installation, Weitergabe oder Löschung von Elementen. Die Erweiterbarkeit auch auf Signalebene eröffnet HADES Anwendungsbereiche, die über das rein technische hinaus gehen. Die Verwendung von Java bietet weiterhin eine Portabilität auch in binärer Form. Dadurch braucht weder der Quellcode des Systems, noch der von Komponenten veröffentlicht werden. Ein eventuell problematisches Übersetzen der Quellcodes entfällt ebenfalls.

4.4.2 DigSim

DigSim ist wie HADES komplett in Java geschrieben und liegt in Form eines Applets vor [DigSim 96]. Der Editor bietet nur grundlegende Funktionen (z.B. ist kein Zoomen möglich), basiert aber auf gängigen GUI-Elementen, wie Menüs und Toolbars, und ist so deutlich intuitiver zu bedienen als z.B. LOG. Die Simulation findet in DigSim ebenfalls interaktiv statt. Der bereits erwähnte „Glow-Mode“ ist dabei fester Bestandteil und läßt sich nicht deaktivieren. Für den Simulationsverlauf enthält DigSim einen „Logic Analyzer“, der jedoch nicht einmal horizontales Scrolling bietet. Für ernsthafteres Simulieren ist DigSim deshalb nicht geeignet.

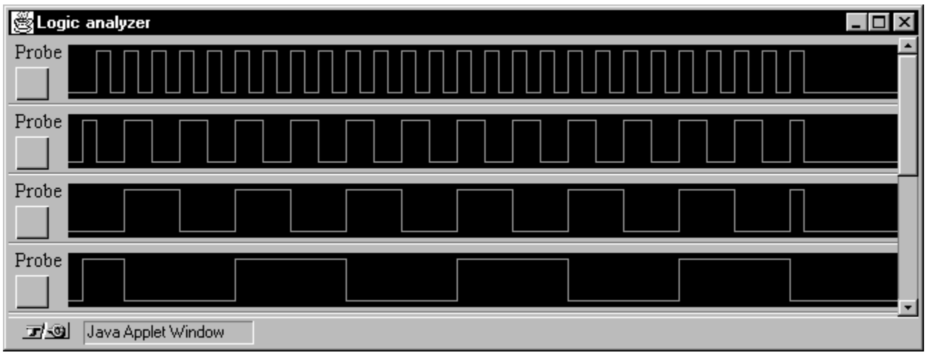

Abbildung 29: DigSim Logic-Analyzer

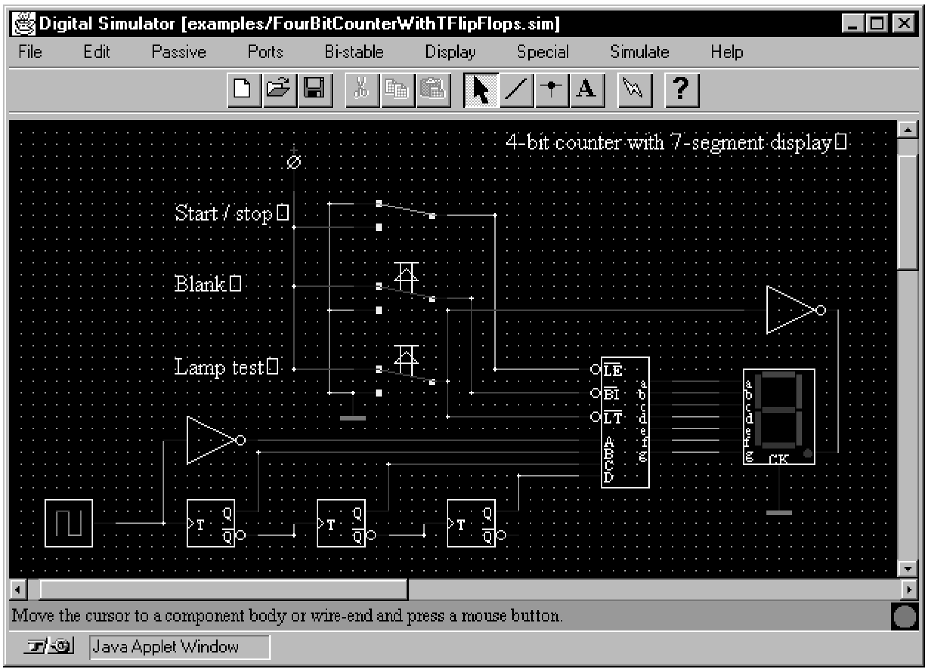

Abbildung 30: DigSim-Editor

Die enthalten Bauteile liegen wie bei HADES als externe Klassen vor, die auch als JavaBeans interpretiert werden könnten. Allerdings erfolgt die Auswahl fest über Menüs, wofür alle Komponenten fest einprogrammiert werden müssen. Als Signale dienen im Gegensatz zu HADES keine eigenständigen Klassen, sondern nur Integer-Werte, wodurch DigSim wie LOG an technische Anwendungsgebiete gebunden ist. Zum Experimentieren mit einfachen Schaltungen ist DigSim aber gut geeignet. Die einzelnen Komponenten machen reichlich Gebrauch graphischer Darstellungsmethoden und ermöglichen so eine gute Visualisierung. Die Realisierung als Java-Applet erleichtern dem unerfahrenen Anwender den Einstieg. Der geringe Funktionsumfang verhindert andererseits aber einen ernsthafteren Einsatz, wodurch DigSim sich im Gegensatz zu HADES, nicht für anspruchsvollere Entwürfe eignet.

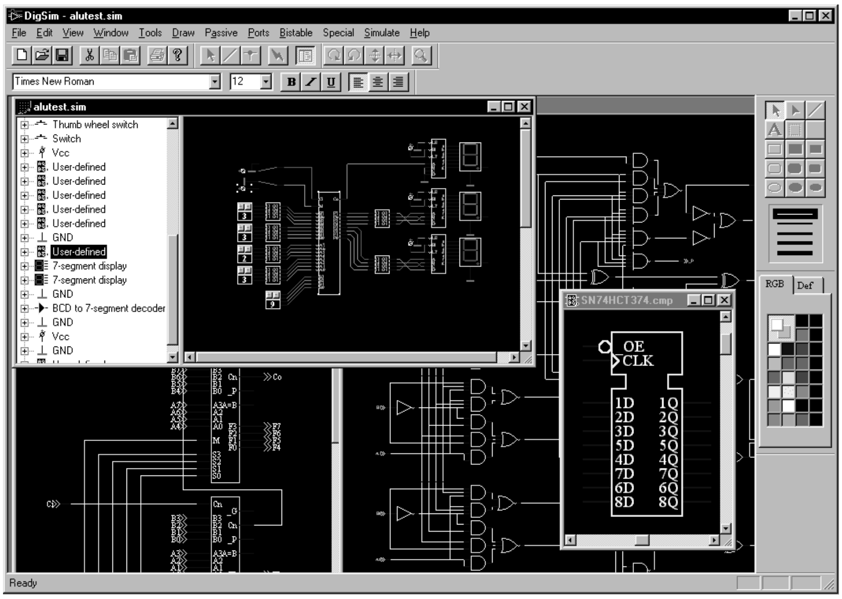

Abbildung 31: DigSim V2

Auf der WWW-Seite des Autors wird eine neue Version von DigSim angekündigt, die einen erheblich gesteigerten Funktionsumfang besitzen soll. DigSim V2 wird dann unter anderem auch hierarchische Designs unterstützten. Die neue Version wird allerdings nicht mehr in Java geschrieben sein und verliert dadurch ihre Plattformunabhängigkeit. Über die Realisation der Simulationselemente wird in der Ankündigung keine genaue Aussage getroffen. Es soll jedoch die Möglichkeit geben, eigene Komponenten zu erstellen.

4.4.3 Cadence

Als Beispiel für ein kommerzielles System zum Schaltungsentwurf wird in diesem Abschnitt das Cadence Design Framework II vorgestellt. Dieses Framework kann sicherlich als eines der komplexesten erhältlichen Softwaresysteme angesehen werden. Die zum Zeitpunkt dieser Arbeit aktuelle Version benötigt inklusive Online-Dokumention – jedoch noch ohne Bauteilbibliotheken – knapp 3 Gigabyte Festplattenspeicher und muß eigentlich ständig von einem Operator gepflegt werden. Cadence ist konsequent auf die Belange professioneller Entwickler ausgelegt. So ist die gebotene Funktionsvielfalt ohne vorherige Schulung kaum zu meistern.

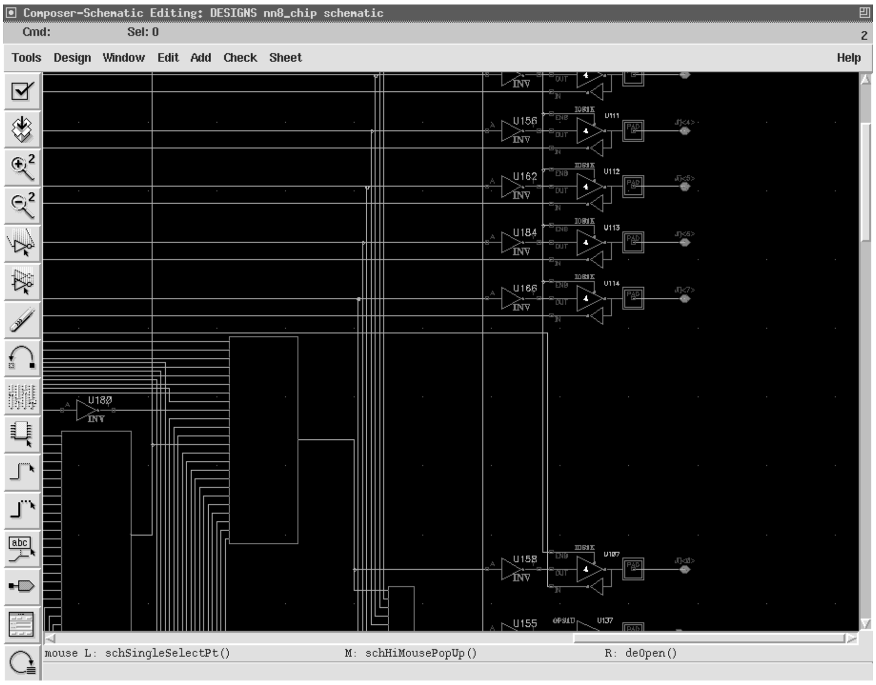

Abbildung 32: Cadence-Editor

Das gesamte Cadence-Paket besteht genau genommen aus einer Reihe von einzelnen Programmen (Manager, Editor, Layout, Simulator, etc.), die auf eine gemeinsame Datenabasis zugreifen. Eine integrierte (Lisp-ähnliche) Sprache „SKILL“ erlaubt dabei dem Benutzer auf das System und die Entwurfsdaten Einfluß zu nehmen. Während des Entwurfsvorgangs sind normalerweise eine Reihe verschiedener Fenster geöffnet:

- Cadence (Ausgaben der Programme, „SKILL“-Kommandointerpreter, etc.)
- Editor (Entwurf als schematic, layout, ... und Start der Simulation)
- Browser (hierarchische Darstellung der Cadence-Bibliotheken)
- Ausgabe (Simulationsergebnisse und Auswertungen)
- Auswahl (zusätzliche Menüs)
- Formular (Einstellungen von Optionen und Parametern)

Als eines der professionellsten Entwicklungswerkzeuge sind für Cadence eine Flut von Komponenten verfügbar. Da aus ihnen jedoch direkt Chips gefertigt werden können, müssen sie käuflich erworben werden. Zur Verwaltung der Komponenten und Designs bietet Cadence im Gegensatz zu einfacheren Werkzeugen einen Bibliotheks-Browser, der die vorhandenen Modelle hierarchisch strukturiert darstellt [Cadence 97].

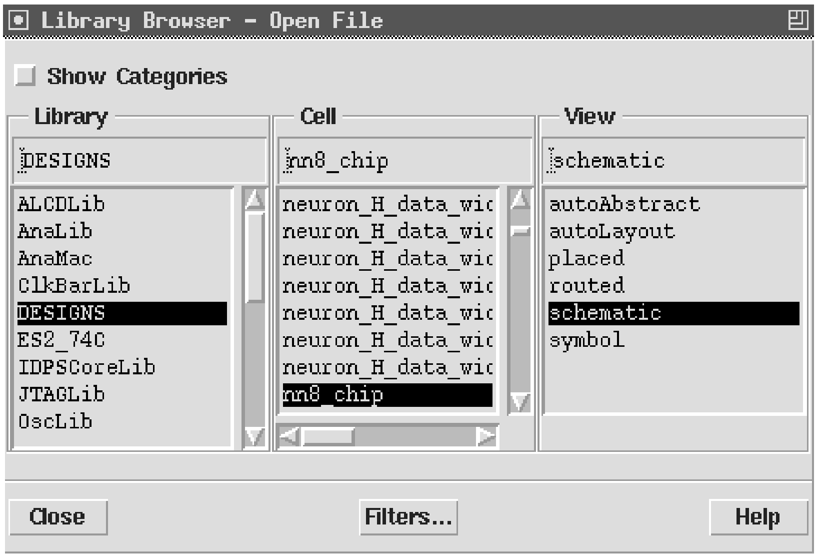

Abbildung 33: Cadence Bibliotheks-Browser

4.5 Zusammenfassung

Frei verfügbare Simulationssysteme sind im allgemeinen gut für Einsteiger geeignet, haben darüber hinaus aber zum Teil gravierende Mängel. Sie bieten meist nur eine eingeschränkte Funktionalität und unterstützen (bis auf DigLOG) keine hierarchischen Entwürfe. Zur Erweiterung wird häufig auf eigene Simulationssprachen zurückgegriffen, die im graphischen / visuellen Bereich kaum Möglichkeiten bieten. Komplexere Objekte, wie z.B. endliche Automaten mit integriertem Editor, lassen durch solche speziellen Sprachen überhaupt nicht erstellen. Weiterhin verhindern mangelnde Exportmöglichkeiten eine Weiterverwendung der erstellten Entwürfe.

Professionelle Entwurfs- und Simulationswerkzeuge (wie Cadence) bieten zwar ein reichhaltiges Funktionsangebot, sind aber von Einsteigern kaum zu beherrschen und stellen aufgrund des großen Ressourcenverbrauchs auch hohe Anforderungen an die verwendeten Rechner. Der Preis solcher Systeme übersteigt leicht die Grenze von $10.000 pro Lizenz und ist daher für einen Einsatz in der Lehre nicht geeignet.

Als frei verfügbares System, geht HADES mit seinen gebotenen Möglichkeiten weit über das Niveau der „Mitbewerber“ hinaus. Bereits jetzt besteht HADES aus über 350 Klassen mit mehr als 90.000 Zeilen Programmcode. Das oben vorgestellte DigSim besteht demgegenüber nur aus 64 Klassen mit gut 10.000 Programmzeilen. Das gesamte DigLOG/AnaLOG Paket ist mit ebenfalls über 90.000 Zeilen C-Quellcode vom Umfang her mit HADES vergleichbar. Allerdings stehen HADES die umfangreichen Java-Klassenbibliotheken zur Verfügung. Müßte man die davon verwendete Funktionalität selber bereitstellen (wie DigLOG/AnaLOG), würde der Umfang von HADES deutlich wachsen.

Die einfache Bedienung und die Plattformunabhängigkeit erlauben die Verwendung auch von praktisch Jedermann. Obwohl der Schwerpunkt von HADES auf der Simulation digitaler Systeme liegt, erlaubt die Integrierbarkeit neuer Signaltypen im Gegensatz zu den meisten anderen Systemen auch nicht-technische Anwendungen. Hauptkritikpunkt von

HADES ist die immer noch relativ schlechte Performanz von Java. Mit den derzeit verfügbaren Rechnern und Interpretern können Entwürfe bis etwa 10.000 Komponenten simuliert werden. Für den semi-professionellen Bereich ist dies jedoch durchaus vertretbar. Abschließend könnten die Eigenschaften von HADES wie folgt zusammengefaßt werden:

- frei verfügbar
- portabel
- einfache Installation (idealerweise als Applet)
- intuitiv Bedienbar
- hierarchische Entwürfe
- Integration von Simulation, Visualisierung und Animation
- Schwerpunkt auf digitalen Systemen (Gatter-, Register- und Systemebene)
- voller Zugriff auf Simulationskern und Datenstrukturen
- erweiterbar durch neue Simulationsmodelle, -signale und -kerne
- gute Exportmöglichkeiten (z.B. nach VHDL)

5 Architekturen

In diesem Kapitel sollen verschiedene bestehende Architekturen vorgestellt werden, die für einen möglichen Einsatz im Browser in Betracht kommen. Der erste Teil gibt eine Einführung in die Common Object Request Broker Architecture (CORBA), eine Architektur für verteilte Objekte in heterogenen Systemen, und die in diesem Rahmen spezifizierten Dienste. Im Anschluß wird eine Bewertung von CORBA im Rahmen eines Einsatzes für HADES und den Komponentenbrowser durchgeführt. Die Java Remote Method Invocation ist Inhalt von Kapitel 5.2. Zur Vollständigkeit erwähnen wir hier auch kurz das Distributed Computing Environment der Open System Foundation, das jedoch durch den großen Erfolg von CORBA an Bedeutung verloren hat. Das System NCSTRL zur Verwaltung von Forschungspapieren beschließt dieses Kapitel.

5.1 CORBA

5.1.1 Einführung

Die Object Management Group (OMG) ist ein 1989 gegründetes Firmenkonsortium mit dem Ziel der Förderung der objektorientierten Programmierung. Zu den Gründungsmitgliedern gehörten American Airlines, Canon, Data General, Sun, Hewlett-Packard, Unisys, 3Com und Philips. Mittlerweile sind über 800 Firmen Mitglied der OMG, darunter alle namhaften Computerfirmen außer Microsoft (→COM). Die OMG realisiert ihre Ziele durch die Erschaffung von Standards, jedoch ohne selbst Software herzustellen oder Implementationsrichtlinien festzulegen. Es werden *Requests for Information* (RFI) und *Requests for Proposals* (RFP) herausgegeben, zu denen die Mitglieder Stellung nehmen. Aus diesen werden die Standards abgeleitet, deren Realisierung dann durch die Mitglieder erfolgt.

Die *Object Management Architecture* (OMA) ist ein Framework über heterogene Hardware-Plattformen und Betriebssysteme für Interkommunikation von Anwendungsobjekten. Sie beschreibt eine verteilte Systemumgebung und besteht aus vier Komponenten. Der Hauptbestandteil ist der *Object Request Broker* (ORB), ein netzweiter Kommunikationsbus für Client/Server-Applikationen in heterogenen verteilten Systemen, über den Objekte sich gegenseitig aufrufen können. Die *Object Services* stellen eine Sammlung von Diensten dar, die Grundfunktionalität zur Kommunikation bieten. Weitere Dienste werden durch die *Common Facilities* bereitgestellt. Die *Application Objects* sind Objekte auf der Anwendungsebene.

Die *Common Object Request Broker Architecture* (CORBA) ist ein Bestandteil *der Object Management Architecture* und spezifiziert die hierfür benötigten Dienste. CORBA Objekte sind wiederverwendbare Softwarekomponenten, die auf semantischer Ebene miteinander kooperieren. Der *Object Request Broker* (ORB) stellt die Infrastruktur für eine Kommunikation zur Verfügung. Ein Client kann über einen ORB eine Methode auf einem Server Objekt aufrufen. Dabei bleibt es für den Client transparent, wo und auf welchem System sich dieses Objekt befindet. Objekte können dabei sowohl Clients als auch Server sein. Der ORB empfängt einen Aufruf und muß nun ein passendes Objekt finden, welches diesen Aufruf ausführen kann. Er übergibt diesem Objekt die Parameter, ruft die entsprechende Methode auf und gibt die Rückgabewerte an den Client zurück.

Im Vergleich zu den herkömmlichen Remote-Procedure-Calls findet ein Aufruf bei CORBA auf ein bestimmtes Objekt statt [Orfali & Harkey 97]. Wird der Aufruf auf ein anderes Objekt durchgeführt, so kommt es zu einem unterschiedlichen Ergebnis. Diese Eigenschaft wird als Polymorphismus bezeichnet. Bei den RPCs werden die Methoden ohne Kontext aufgerufen und gleichnamige Methoden werden auch immer gleich implementiert.

Seit CORBA 2.0 existiert auch ein Standard für die Interoperabilität zwischen ORBs verschiedene Hersteller, damit diese miteinander kommunizieren können. Es wurde das *Internet Inter-ORB Protocol* (IIOP) definiert, das auf TCP/IP mit einem durch CORBA definierten Nachrichtenaustausch aufsetzt. Hersteller müssen entweder das IIOP oder eine Brücke dazu implementieren.

Das *General Inter-ORB Protocol* (GIOP) stellt eine Standard-Transfersyntax zur Verfügung, die es ermöglicht, Brücken zwischen ORBs verschiedener Hersteller zu implementieren. Dabei unterscheidet man zwischen direkten und indirekten Brücken. Direkte Brükken verbinden zwei ORBs direkt miteinander. Indirekte Brücken werden zunächst an den Domänengrenzen in eine allgemeine Form, transferiert, auf die man sich zuvor geeinigt hat (z.B. IIOP). Die erste Variante ist zwar schneller, dafür ist die zweite jedoch allgemein gehalten.

Ein Client greift auf einen Service eines Objektes über eine fest definierte Schnittstelle zu. Diese wird durch die *Interface Definition Language* (IDL) beschrieben. Da diese IDL neutral ist, ist sie portierbar über Werkzeuge, Sprachen, Betriebssysteme und Hardwareplattformen. Es kann jede Programmiersprache gewählt werden, für die eine IDL-Abbildung existiert. Zur Zeit sind dies C, C++, Java, Ada und Smalltalk. Dadurch kann auch bereits existierender Code eingebunden werden. Durch die IDL wird die Implementation von der Spezifikation getrennt.

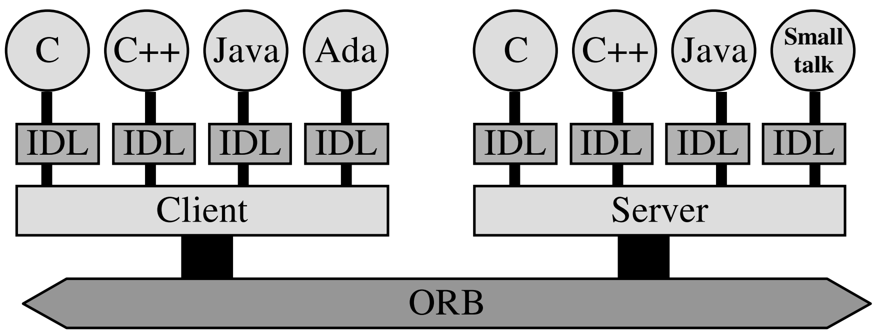

Abbildung 34:Client und Server, die in verschiedenen Sprachen geschrieben wurden, können über den ORB miteinander interoperieren

In der IDL werden die Methoden, Ein- und Ausgabeparameter und deren Datentypen beschrieben. Außerdem werden die Klassen, von denen das Objekt erbt und dessen Events und Exceptions angegeben. Dabei werden sprachneutrale Datentypen verwendet, die die Datentypen der meisten prozeduralen Sprachen abdecken.

Zur Kommunikation der Clients mit dem ORB gibt es zwei verschiedene Möglichkeiten. Zum einen werden statische Bindungen verwendet, bei der zweiten Methode sind die Bindungen dynamisch.

Bei den statischen Bindungen werden sogenannte *Stubs* (= Coderümpfe) verwendet, die zuvor von einem Precompiler aus dem Interface erzeugt wurden. Ein *Stub* verhält sich wie lokaler Aufruf und legt fest, wie der Client Methoden aufruft. Der *Stub* ist für das *Marshaling* verantwortlich: er kodiert und dekodiert die Methodenaufrufe und Parameter und verpackt sie in ein flaches Nachrichtenformat zum Versenden über das Netz.

Das Gegenstück zum *Stub* auf der Server-Seite ist der *Skeleton*. Dieser nimmt sowohl statische als auch dynamische Client-Anfragen entgegen. Dabei läuft die eigentliche Kommunikationsanbindung über den *Object Adapter* (OA). Jeder ORB muß mindestens einen sogenannten *Basic Object Adapter* (BOA) implementieren.

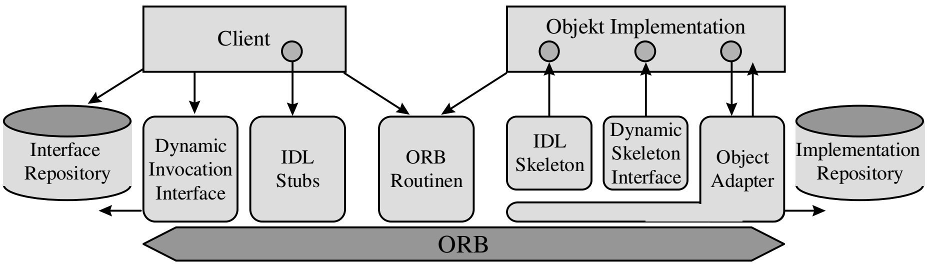

Abbildung 35: Kommunikation über den ORB mit statischen oder dynamischen Bindungen

Bei der dynamischen Bindung verwendet der Client das *Dynamic Invocation Interface* (DII). Über dieses können Clients auch auf Objekte zugreifen, deren Definitionen oder Interfaces sie zur Compilezeit nicht kennen. Das DII bietet hierzu Methoden, um Objekte zur Laufzeit zu entdecken. Dazu muß der nachgefragte Dienst vorher in das *Interface Repository* eingetragen worden sein. Das *Interface Repository* enthält alle Interfacedefinitionen und zusätzliche Daten (Metadaten) der registrierten Komponenten, so daß sich die Objekte zur Laufzeit entdecken können. Durch die Metadaten können Komponenten selbstbeschreibende Interfaces haben. Die Speicherung der Objektschnittstellen-Definitionen im *Interface Repository* ist persistent. Das *Implementation Repository* enthält Informationen über die Klassen, die ein Server anbietet. Es verwaltet die instanziierten Objekte und deren Objektreferenzen.

Analog zum DII gibt es auf der Server-Seite das *Dynamic Skeleton Interface* (DSI). Ist auf der Client-Seite kein *Stub* vorhanden, erfolgt eine Kommunikation zwischen ORB und Objekt über eine Schnittstelle, die Zugriff auf Namen und Parameter des Objektes über das *Interface Repository* ermöglicht. Die Bindung erfolgt dynamisch zur Laufzeit.

Statische Bindungen sind leichter zu programmieren als dynamische. Außerdem sind sie performanter und ermöglichen eine Typüberprüfung zur Compilezeit. Dynamische Bindungen bieten dafür eine hohe Flexibilität und ermöglichen Werkzeuge, die die Objekte zur Laufzeit auffinden.

Object Services
Die *CORBA Object Services* sind ein Sammlung von Diensten, die Grundfunktionalität zur Verfügung stellen. Mit ihnen läßt sich eine Komponente nach Wunsch mit bestimmten Services kombinieren, um gewünschte Dienste zu verwenden. Die Komponente erbt von den passenden Services und erhält so die gewünschten Eigenschaften.

Da die Services im Rahmen eines möglichen Einsatzes in HADES eine größere Rolle spielen, widmen wir ihnen ein eigenes Unterkapitel.

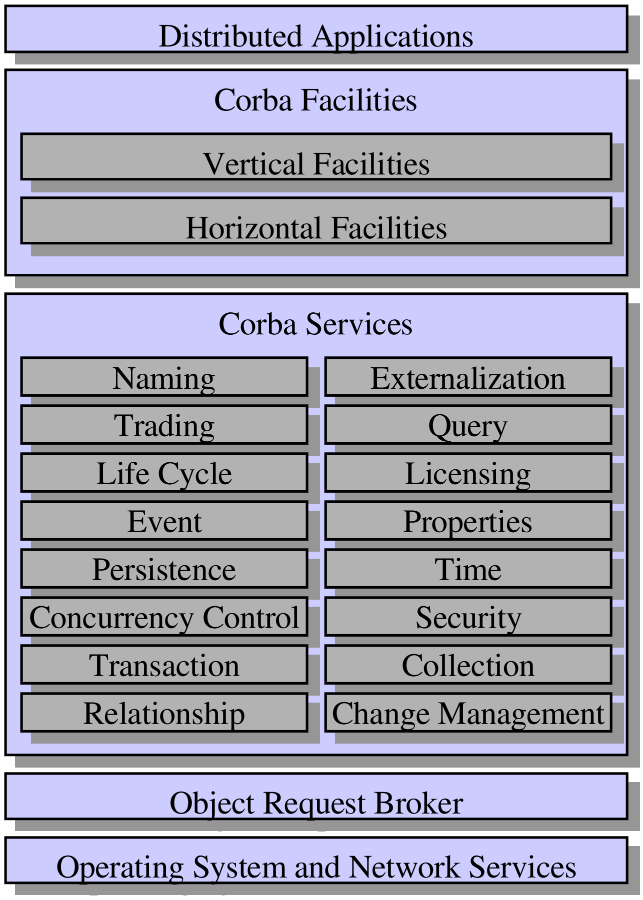

Abbildung 36: OMG CORBA Architektur

Common Facilities
Common Facilities sind Dienste, die semantisch eine Stufe höher einzuordnen sind als *Object Services*. Sie können von vielen Applikationen verwendet werden, sind aber nicht ein so fundamentaler Bestandteil von CORBA wie die *Object Services*. Man unterscheidet zwischen horizontalen Diensten, die benutzerorientiert sind, und vertikalen Diensten, die spezielle Anwendungsgebiete unterstützen. Vertikale Dienste sind z.B. Buchführung, Computer-Integrierte-Fertigung, Währung, Internationalisierung und Telekommunikation. Horizontale Dienste werden nochmals in vier Kategorien eingeteilt: *User Interface Facilities*, wie z.B. Desktop Management, *Information Management*, wie z.B. Datenaustausch und Daten-verschlüsselung, *System Management Facilities*, wie z.B. Qualitätssicherung und Event Handling, *und Task Management Facilities*, wie z.B. mobile Agenten.

Application Objects
Anwendungsobjekte werden von der OMG nicht standardisiert. Sie beschreiben die oberste Schicht im Referenzmodell und stellen die Benutzer der CORBA Infrastruktur dar.

5.1.2 CORBA Services

5.1.2.1 Naming

Um ein anderes Objekt nutzen zu können, muß ein Objekt Kenntnis von diesem haben. Der Naming-Service dient zum Auffinden anderer Objekte. Er ordnet ihnen für Menschen verständliche Namen zu (Bindung). Ein Name ist in einem bestimmten Namensraum eindeutig. Ein Objekt kann jedoch in einem Namensraum an mehrere Namen gebunden sein. Ein Namensraum ist ein Objekt, das eine Menge von Namensbindungen enthält, und in welchem jeder Name eindeutig ist.

Der Naming-Service beinhaltet bestehende Namensschemata wie DCE CDS, ISO X.500 und Sun NIS+ [Orfali et al.96]. Er ermöglicht eine hierarchische Struktur von Namensräumen. Ein Name für ein Objekt besteht dann aus einer Folge von Namen, wobei jeder Name außer dem letzten einen Namensraum benennt.

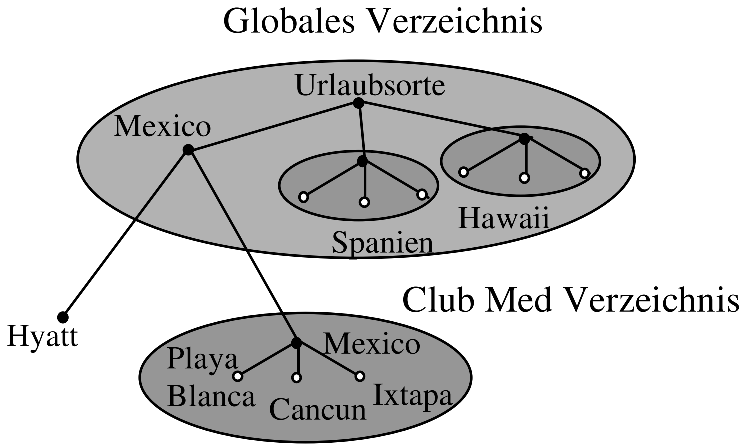

Abbildung 37: Hierarchische Struktur von Namensräumen

Um ein Objekt zu lokalisieren, muß der Name aufgelöst, d.h. die einzelnen Namensräume bestimmt werden. Ein Name wird dann in einem bestimmten Namensraum an das Objekt gebunden.

Der Naming-Service ist für die Verwaltung der Namensräume und das Auffinden von Objekten in Namensräumen zuständig. Er ist mit einem Telefonbuch vergleichbar.

5.1.2.2 Trading

Um Dienste nutzen zu können, müssen diese dem Benutzer bekannt sein. Neben dem Naming Service als dem „Telefonbuch von CORBA“ kann der Trading-Service als die „Gelben Seiten“ bezeichnet werden. Die Provider registrieren hier ihre Dienste und die Benutzer konsultieren den Trader, um Informationen über passende Dienste zu erhalten. Ein Dienst wird durch ein oder mehrere Objekte mit einem oder mehreren Interfaces bereitgestellt. Zu einem Dienst gibt es ein Service-Angebot, in dem dieser Dienst beschrieben wird. Es enthält das Interface, welches der Dienst erfüllt, dessen Typ und Angaben über die Qualität des Dienstes. Dabei gibt es statische und dynamische Qualitätsmerkmale. Für einen Druckdienst wäre z.B. die dpi-Anzahl ein statisches und die Länge der Druckerwarteschlange ein dynamisches Qualitätsmerkmal.

Der Client erhält vom Trader eine Objekt Referenz, mit der er auf den Dienst zugreifen kann und eine Beschreibung des Diensttyps, der angeboten wird. Dieser enthält Informationen über die Methodennamen, deren Parameter und Rückgabewerte.

5.1.2.3 Life Cycle

Der Object-Life-Cycle-Service ist ein Dienst zum Erstellen, Kopieren, Verschieben und Löschen von Objekten. Dabei können Life-Cycle Operationen auf Objekte an verschiedenen Orten durchgeführt werden. Eine Life-Cycle Operation wird von einem Client initialisiert.

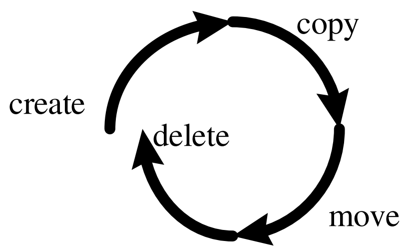

Abbildung 38: Object Life Cycle

5.1.2.4 Event

Der Event Service hat die Aufgabe, die asynchrone Kommunikation zwischen anonymen Objekten zu ermöglichen. Er tritt dabei als eine Art Vermittlungsstelle zur Entkopplung der Aufrufer und der Aufgerufenen auf. Objekte, die an einem bestimmten Ereignis interessiert sind, registrieren sich hierfür beim Ereignisdienst. Dieser benachrichtigt dann bei Auftreten dieses Ereignisses alle angemeldeten Objekte. Dies wird als das Push-Modell bezeichnet. Im alternativen Pull-Modell fragen die Clients beim Ereignisdienst nach.

5.1.2.5 Persistence

Objekte in verteilten Systemen müssen ihren Zustand aufrechterhalten, auch wenn das Programm, welches sie erzeugt hat, beendet wurde. Dazu muß der Zustand des Objektes in einem beständigen Speicher abgelegt werden. Der Persistent-Object-Service (POS) stellt ein Interface für das Management und die Erhaltung des persistenten Zustandes von Objekten bereit. Das Objekt wird in einem dauerhaften Speicher abgelegt und bei Bedarf wieder eingelesen. Dabei hat das Objekt selbst die Verantwortung für die Verwaltung seines Zustandes, kann diese aber an den Dienst übertragen.

Der POS definiert die Schnittstelle für persistente Objekte. Die Implementation kann dann durch verschiedene Arten von Speicher, wie Dateien, relationale und objektorientierte Datenbanken und Dokumentablagensysteme realisiert werden. Dabei entscheidet das Objekt, welche Form gewählt wird.

5.1.2.6 Concurrency Control

Dieser Dienst erlaubt es mehreren Objekten, ihren Zugriff auf gemeinsame Ressourcen zu koordinieren. Wenn mehrere Objekte gleichzeitig auf die gleiche Ressource zugreifen, ist der konsistente Zustand dieser Ressource gefährdet. Der Concurrency-Control-Dienst bringt die Zugriffe in Einklang und vergibt Sperren für Ressourcen an die Objekte. Um auf eine Ressource zugreifen zu können muß ein Objekt nun zuerst so eine Sperre erwerben. Dabei werden verschiedene Stufen für lesenden und schreibenden Zugriff realisiert.

5.1.2.7 Transaction

Der Transaction-Dienst stellt sicher, daß eine Berechnung, die aus mehreren Operationen besteht oder auf mehreren Objekten durchgeführt wird, komplett ausgeführt werden kann, um ein konsistentes Ergebnis zu liefern. Tritt während der Ausführung ein Fehler auf, muß der Ursprungszustand wiederhergestellt werden.

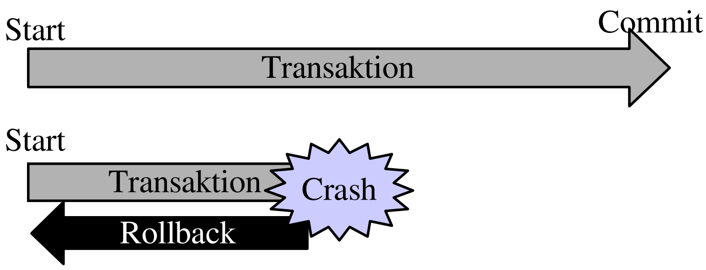

Abbildung 39: Eine Transaktion wird entweder komplett ausgeführt und mit einem Commit bestätigt, oder der Ursprungszustand muß wiederhergestellt werden.

Es gibt dabei flache und geschachtelte Transaktionen. Der Transaction Service baut auf dem Concurrency Control Service und dem Persistent Object Service auf.

5.1.2.8 Relationship

Dieser Dienst ermöglicht es, zwischen verschiedenen Objekten dynamische Beziehungen zu erstellen und zu verwalten. Dabei bleiben die Beziehungen für die Objekte selbst transparent.

Der Relationship-Service definiert zwei neue Objektarten: die Beziehungen und die Rollen. Eine Rolle ist ein CORBA-Objekt in einer Beziehung. Eine Beziehung entsteht durch Übergabe von Rollen an eine Beziehungs-Fabrik.

5.1.2.9 Externalization

Der Externalization-Service stellt Schnittstellen zur Verfügung, mit denen Objekte zum Transport in einen Stream geschrieben (Externalization) und am Zielort wieder ausgelesen werden (Internalization). Ein Stream kann sich im lokalen Speicher, auf Festplatte oder irgendwo im Netz befinden. Das Schreiben in den Stream und das anschließende Wieder-Auslesen ist mit dem Kopieren zu vergleichen. Es handelt sich hierbei jedoch um zwei getrennte Schritte und das Zwischenergebnis kann getrennt manipuliert, z.B. auf einen anderen Rechner transportiert werden.

5.1.2.10 Query

Der Query-Service dient zum Auffinden von Objekten, die den Suchkriterien einer Anfrage genügen. Die Suchanfrage kann jedoch nicht auf den internen Zustand des Objektes zugreifen, sondern nur auf die nach außen zur Verfügung gestellten Methoden.

Der Query-Service basiert auf SQL und der Object Query Langugage (OQL). Eine Anfrage kann in SQL, OQL oder Untermengen dieser Sprachen gestellt werden [Orfali et al.96]. Dabei sind auch geschachtelte Anfragen möglich.

5.1.2.11 Licensing

Damit CORBA auch für kommerzielle Komponenten-Anbieter interessant wird, müssen Methoden zur Abrechnung von Komponenten zur Verfügung gestellt werden. Der Licensing-Dienst ermöglicht es, die Benutzung von Komponenten zu messen und zu berechnen. Dabei sind verschiedene Abrechnungsmodi möglich und können den Bedürfnissen der Anbieter angepaßt werden.

5.1.2.12 Properties

Die in IDL definierten Attribute sind nur durch „get"- und „set"-Methoden veränderbar. Zur Laufzeit können keine neuen Attribute hinzugefügt werden. Der Property-Service ermöglicht es, Objekten zur Laufzeit Attribute außerhalb der IDL zuzuordnen. Diesen Attributen kann dann ein Name gegeben, ein Wert zugeordnet und der Wert über den Namen abgefragt werden. Außerdem können die Attribute wieder gelöscht werden. Die Attribute können dabei jeden Wert annehmen, der im OMG IDL-Typ-System repräsentiert ist. Hierzu gibt es einen speziellen Datentyp „ANY".

5.1.2.13 Time

Der Time-Service dient zum Synchronisieren von Uhren in verteilten Systemen. Wenn mehrere Objekte zeitkritische Operationen durchführen, muß sichergestellt sein, daß als Basis die gleiche Uhrzeit verwendet wird.

Probleme treten dabei beim Synchronisieren und Stellen der Uhren auf: Wenn ein Objekt beim Server die Uhrzeit nachfragt, vergeht zwischen dem Absenden der Uhrzeit beim Server und der Ankunft beim Client Zeit, die beim Stellen der lokalen Uhr berücksichtigt werden muß. Der Client sollte hier die Zeit bei verschiedenen Servern anfragen und jeweils die Antwortzeit mit einrechnen. Außerdem kann es zu Problemen führen, wenn die Zeit beim Client abrupt umgestellt wird, da der zwischen der alten und der neuen Zeit liegende Zeitraum entweder gar nicht oder doppelt vorkommt. Natürlich müssen auch verschiedene Zeitzonen vom Time-Service berücksichtigt werden.

Die CORBA-Spezifikation gibt stellt hierzu lediglich Anforderungen an den zu implementierenden Dienst auf. Die Lösung der dargestellten Probleme bleibt dem Hersteller überlassen.

5.1.2.14 Security

CORBA bietet mit dem Security-Service ein System für die Sicherheit verteilter Objekte. Dazu gehören folgende Aspekte:

- Im Rahmen der Vertraulichkeit werden Informationen nur an autorisierte Benutzer gegeben.
- Objekte dürfen nur durch autorisierte Benutzer und nur in erlaubter Weise modifiziert werden, um die Integrität des Systems sicherzustellen.
- Ein Benutzer kann für seine sicherheitsrelevanten Aktionen verantwortlich gemacht werden.

- Einem autorisierten Benutzer kann der Zugriff auf das System nicht verweigert werden.

Der Security-Service stellt hierzu Methoden zur Identifikation und Authentisierung von Objekten und Benutzern und eine Zugangskontrolle zur Verfügung.

Bei der Versendung von Nachrichten muß sichergestellt werden, daß keine Manipulation durch Dritte stattgefunden hat. Der Empfänger erhält einen Nachweis der Identität des Senders. Dies wird durch Signaturen und Verschlüsselungsalgorithmen erreicht. Durch eine Prüfsumme kann sichergestellt werden, daß die Nachricht unterwegs nicht verändert wurde.

5.1.2.15 Collection

Eine Kollektion oder Sammlung ermöglicht es, Objekte in einer Gruppe zu manipulieren. Beispiele hierfür sind Schlangen, Stapel, Liste, Arrays, Bäume und Mengen. Der Collection Service stellt einheitliche Methoden zur Erstellung und Manipulation der am häufigsten vorkommenden Sammlungen zur Verfügung.

5.1.2.16 Change Management

Dieser Dienst verwaltet verschiedene Versionen eines Objektes. Dabei wird sichergestellt, daß ein Objekt eine konsistente Implementation verwendet. Die Version wird direkt in die Objekt-Referenz mit aufgenommen. Mehrere Versionen eines Objektes können gleichzeitig verwendet werden. Eine Version kann in einem bestimmen Kontext stehen, wie Demonstration, Test, Auslieferungsversion. Die standardmäßig verwendete Version ist dabei immer die jüngste. Der Change Management Service ist von der OMG noch nicht spezifiziert.

5.1.3 Eignung für HADES

CORBA ist zur Zeit die einzige Architektur, die eine sprachunabhängige Definition von Komponenten und eine plattformübergreifende Kommunikation bietet. Die spezifizierten und teilweise implementierten Dienste bieten eine Infrastruktur, die allen Anforderungen gewachsen sein sollte. Der offene Standard mit guten theoretischen Ansätzen und langen Fristen bis zu konkreten Spezifikationen garantiert dabei eine ausgereifte Basis. Die direkte Mitarbeit zahlreicher Firmen resultiert in einer breiten Unterstützung. Die vielen dabei entstehenden Implementationen schaffen eine Konkurrenz-Situation, die sich positiv auf die Qualität auswirkt [Baker 97]. Die seit CORBA 2.0 vorhandene Interoperabilität ermöglicht auch die Zusammarbeit verschiedener Implementationen untereinander.

Für einen möglichen Einsatz in Hades ergeben sich verschiedene Anforderungen an eine Implementation. Zunächst ist HADES komplett in Java implementiert. Die Java-Sprachbindungen sind jedoch noch nicht vollständig spezifiziert. Die entsprechende CORBA-Implementation muß daher in jedem Fall Java unterstützen. Ein Einsatz von CORBA wird insbesondere durch die verschiedenen zur Verfügung gestellten Object Services reizvoll. Die gewählte Implementation sollte daher die für Simulationskomponenten relevanten Dienste wie Naming, Trading, Persistence und Change Management enthalten. HADES ist ein bereits bestehendes System für das schon eine Reihe von Simulationsobjekten existieren. Diese Komponenten müssen daher ohne Anpassungen übernommen werden können. Für den Entwickler von Komponenten soll der Einsatz von CORBA transparent bleiben. Schließlich soll HADES bei Studenten eingesetzt werden und muß daher kostenfrei sein. Es darf sich also nicht um eine kommerzielle Implementation handeln.

Entscheidend für den Einsatz von CORBA ist nicht die Spezifikation, sondern die verfügbaren Implementationen. CORBA wird von vielen Anbietern unterstützt. Mittlerweile sind eine Vielzahl von CORBA-Implementationen erhältlich, darunter auch einige kostenfreie. Jedoch umfassen diese Implementationen noch nicht die gesamte Spezifikation. So sind die Services von der OMG bisher noch nicht alle spezifiziert und von den verfügbaren Implementationen nur zum Teil implementiert.

OrbixWeb von der Firma Iona und der ObjectBroker von Digital Equipment schneiden bei einem Vergleich von fünf verschiedenen Implementationen in [Frey 97] am besten ab. Dabei wird aber auch gesagt, daß noch viele entscheidende Object Services fehlen.

Ein weiterer Nachteil von CORBA im Rahmen von HADES ist, daß der ORB auf jedem Rechner installiert sein muß. Eine Installation ist zwar in der Regel einfach [Frey 97], trotzdem sollen Studenten nicht damit belastet werden.

Sinn von CORBA ist es, die Komponenten jeweils auf dem Server auszuführen. Dabei kann es zu Rechnerausfällen und Verzögerungen kommen. Bei HADES ist eine Ausführung aufgrund der Simulation aber sehr zeitkritisch. Daher soll in jedem Fall die Möglichkeit geschaffen werden, die Simulation lokal auszuführen. Für allgemeine Komponenten ist ein entfernter Aufruf aber sicherlich sinnvoll. Jedoch kann hier im Rahmen eines reinen Java-Systems auch RMI zum Einsatz kommen.

Hersteller	Java	DII	DSI	IR	Naming	Lifecycle	Event	Trading	Cuncurrency	Externalization	Persistent Objects	Transactions	Query	Collections	Time	Properties	Security	Licensing	Solaris	Linux	Windows 95	Windows NT	Mac
Expersoft	y	y	y	y	y	#	y				#								y			y	
Sun	y	y	y	y	y	y	y				#					y			y			y	
IONA	y	y	y	y	y	?	y	y	?		?	y					y		y			y	
Visibroker	y	y	y	y	y	?	y		?		?	y					y		y			y	
ICL	-	y	y	y	y	y	y	y	?		?	y					y		y			y	
HP	97	y	y	y	y	y	y	y				y					?		y			y	
IBM	97	y	y	y	y					y												y	
OOT	-	97	-	-	y	+				y							#		y	y		y	
BBN	?	y	?	y	y	y					y								y				
SNI	-	y	y	y	y	y	y									y			y			y	
ParcPlace	+	y	?	y	y	y,#	y		y	#	#	y				#	#		y			y	
TIBECO	y	y	y	y	y		y												y			y	
Suite	?	y	y	y	y	y	y	y		y	y				y	y	y		?			y	
Fujitsu	y	?	?	?	y	y	y		y	y	y	y	+	+	+	+	+		?	?	?	?	?
Nortel	-	-	-	-	y	y	y												y			+	
kostenfreie Implementationen																							
Voyager	y				y	y	y				y				y								
U Colorado	y	-	-	-															y				
Xerox	y	-	-	-	#	#													y	y	y	y	
TAO	-	y	y	y	y	y	y	y	y						y	y			y	y	y	y	
Hersteller	Java	DII	DSI	IR	Naming	Lifecycle	Event	Trading	Cuncurrency	Externalization	Persistent Objects	Transactions	Query	Collections	Time	Properties	Security	Licensing	Solaris	Linux	Windows 95	Windows NT	Mac

Tabelle 1: Vergleich einiger CORBA-Implementationen [Eng 98]

5.2 Java Remote Method Invocation

In verteilen Systemen müssen Objekte über verschiedene Adreßräume und unterschiedliche Rechner miteinander kommunizieren können. Java stellt hierzu zunächst grundlegende Netzwerkverbindungen zur Verfügung. Diese setzen allerdings voraus, daß der verteilte Zugriff in der Applikationen selbst mit implementiert werden muß. Wünschenswert ist jedoch ein System, in dem die Objekte selbst sich nicht um den verteilten Zugriff kümmern müssen, sondern den Eindruck haben, einen lokalen Methodenaufruf durchzuführen.

Remote Method Invocation (RMI) ist ein solches Modell für verteilte Objekte in Java. Ein verteiltes Objekt ist ein Objekt, dessen Methoden aus anderen Virtual Machines heraus aufgerufen werden können. Es implementiert ein oder mehrere Interfaces, in denen die Methoden beschrieben werden, die das Objekt exportiert. Das RMI-Modell fügt sich nahtlos in das Java Objektmodell ein und führt das „write once run anywhere“-Konzept von Java weiter. Die Behandlung entfernter Objekte erfolgt dabei analog zu der von lokalen Objekten. Über das *Java Native Method Interface* (JNI) und die Java Datenbank

Schnittstelle (JDBC) kann über RMI auch mit „nicht-Java“-Objekten kommuniziert werden [RMI Whitepaper 98].

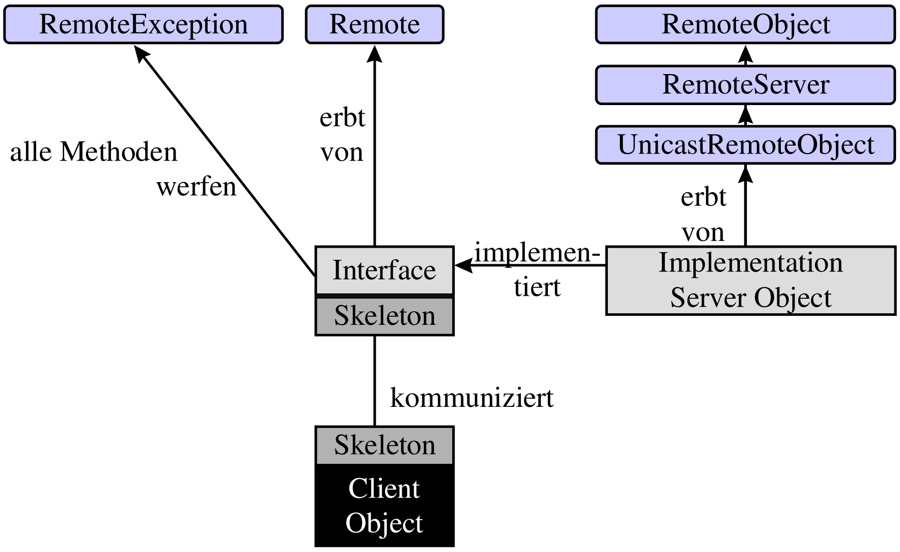

Abbildung 40:Verteiltes Objektmodell bei RMI

Um ein entferntes Objekt zu entwerfen, wird zunächst das zu exportierende Interface definiert. Es erbt direkt oder indirekt von dem Interface `Remote`, das selbst keine Methoden hat, sondern nur als Kennzeichnung für ein entferntes Objekt dient. In dem Objekt-Interface werden die Methoden definiert, die das entfernte Objekt zu Verfügung stellt. Der Client kommuniziert nicht direkt mit dem entfernten Objekt, sondern nur über dieses Interface.

Als Beispiel sei hier das bekannte HelloWorld-Programm angeführt, welches bei Aufruf eine freundliche Nachricht ausgibt:

```
public interface Hello extends java.rmi.Remote
   {
    String sayHello() throws java.rmi.RemoteException;
   }
```

Bei der *Remote Method Invocation* können diverse Fehler auftreten. Daher gibt es zusätzliche Exceptions, die von der Klasse `RemoteException` erben. Jede Methode eines entfernten Objektes muß also zusätzlich zu eventuell vorhandenen diese Exception deklarieren.

Das entfernte Objekt selbst implementiert sein Interface und erbt dabei von der Klasse `UnicastRemoteObject`. Multicasting wird zur Zeit noch nicht unterstützt, ist aber in der Architektur vorgesehen. Das `UnicastRemoteObject` erbt selbst von `RemoteServer` und damit wiederum von `RemoteObject` (siehe Abbildung). Der Konstruktor erzeugt das Objekt und ruft die Methode `exportObject()` auf.

Einige Methoden der Klasse `Object` sind für entfernte Methodenaufrufe speziell anzupassen. Hierzu stellt die Klasse `RemoteObject` eigene Implementationen von `hashCode()`, `equals()` und `toString()` bereit. Zum Beispiel wird in der Methode `toString()` die Herkunft des Objektes berücksichtigt.

Die Klasse `RemoteServer` vererbt die Semantik der entfernten Referenzierung und des Managements von Objekten. Sie enthält Methoden, um Serverobjekte zu erzeugen und zu exportieren.

```
public class HelloImpl
   extends UnicastRemoteObject
   implements Hello
{
   public HelloImpl() throws RemoteException
   {
      super();
   }

   public String sayHello() throws RemoteException
   {
      return "Hello World!";
   }

   public static void main(String args[])
   {
      System.setSecurityManager(new RMISecurityManager());
      try
      {
         HelloImpl obj = new HelloImpl();
         Naming.rebind("//134.100.13.201/HelloServer", obj);
         System.out.println("HelloServer bound in registry");
      }
      catch(Exception e)
      {
         System.out.println("Exception while binding in
            Registry"+e);
      }
   }
}
```

Ein RMI-Server kann auch weitere nicht im Interface angegebene Methoden implementieren. Diese sind dann allerdings nur lokal verfügbar. Die main-Methode registriert den

`RMISecurityManager` (siehe unten). Danach erzeugt sie eine Exemplar des Objektes. Über den Konstruktor wird auch der Konstruktor des `UnicastRemoteObject` aufgerufen, der das Objekt exportiert. Ab diesem Zeitpunkt wartet das Objekt auf eingehende Aufrufe. Anschließend wird das Objekt noch beim RMI-Nameserver registriert.

Zur Lokalisierung entfernter Objekte läuft auf dem Server ein Dämonprozeß (rmiregistry) der als Nameserver für die Serverobjekte dient und deren Bezeichnungen / Namen verwaltet. Von diesem beschafft sich der Client mit Hilfe einer URL eine Referenz auf das gewünschte Serverobjekt und ruft dann auf dem entfernten Objekt die Methode auf.

Das Registrieren und Auffinden der Serverobjekte erfolgt über die Server-URL und einfache Namen für die Serverobjekte (`rmi://host:port/name`). Dazu dient die Klasse `java.rmi.Naming`, mit deren Methode `bind()` ein Serverobjekt an seinen Namen gebunden wird. Ein Nachteil der Registry ist jedoch, daß sie nicht persistent ist. Wird der Dämon gestoppt und wieder gestartet, gehen die Registrierungen der Objekte verloren.

Ein Java Programm kann einen Aufruf an ein entferntes Objekt senden, nachdem es eine Referenz darauf erhalten hat. Diese erhält es, indem es entweder beim RMI Nameserver (rmiregistry) nachfragt oder in dem es eine Referenz als Argument oder Rückgabewert erhält.

```
System.setSecurityManager(new RMISecurityManager());
try
{
   HelloImpl obj = new HelloImpl();
   Naming.rebind(„//myhost/HelloServer“, obj);
}
catch(Exception e){}
```

Ein lokales Stellvertreterobjekt, welches als *Stub* bezeichnet wird, führt den Aufruf auf dem entfernten Objekt aus. Auf der Serverseite ist ebenfalls ein Stellvertreterobjekt – der *Skeleton* – vorhanden, der den RMI-Aufruf entgegennimmt. Der Stub implementiert die gleichen Methoden wie das entfernte Objekt. Daher können die normalen Java-Operatoren für Casting und Typüberprüfung verwendet werden.

Stubs und Skeletons werden durch den RMI-Compiler (rmic) erzeugt. Im Normalfall befinden sich die Stub-Klassen bereits beim Client. RMI ermöglicht jedoch auch ein dynamisches Laden der Stubs. Hierzu werden ein eigener `ClassLoader` und ein eigener `SecurityManager` benötigt. Die Klassen werden mittels Object-Serialisierung übertragen. Die URL zum Laden der Klassen erhält der Client dabei entweder vom Server oder aus der Umgebungsvariable `java.rmi.server.codebase`. Ein anderer Fall des dynamischen Ladens tritt dann ein, wenn der Client ein Applet ist und die Stub-Klassen auf dem Ursprungsserver liegen.

Bei der Übergabe von Parametern an und Rückgabewerten von entfernten Objekten sind einige Besonderheiten zu beachten. Lokale Argumente werden als Kopie (call by value) übergeben, da Referenzen auf Objekte nur innerhalb einer Virtual Machine sinnvoll sind. Außerdem werden nur nicht statische und nicht transiente Felder mit übergeben. Bei entfernten Objekten wird das Stellvertreter-Objekt (Stub / Skeleton) übergeben (call by reference), da eine Referenz auf dieses ja bereits existiert. Zur Übertragung der Parameter werden diese serialisiert. Die Parameterklassen müssen daher das Interface Serializable erfüllen. Die meisten JavaKlassen erfüllen dieses Interface, insbesondere die Remote Objects.

Das RMI-Modell ist in drei Schichten aufgeteilt. Die erste Schicht ist der Stub / Skeleton Layer, der für das Marshaling und Unmarshaling von Parametern verantwortlich ist und die tiefere Schicht anweist, den Aufruf durchzuführen. Der Remote Reference Layer regelt die Kommunikation mit der Transportschicht. Aufgabe der Transportschicht ist der Aufbau, die Verwaltung und die Überwachung von Verbindungen zu entfernten Adreßräumen. Außerdem verwaltet sie die Referenztabelle (rmiregistry) der aktuellen entfernten Objekte im lokalen Adreßraum. Verbindungswünsche von Clients werden ebenfalls von der dritten Schicht angenommen.

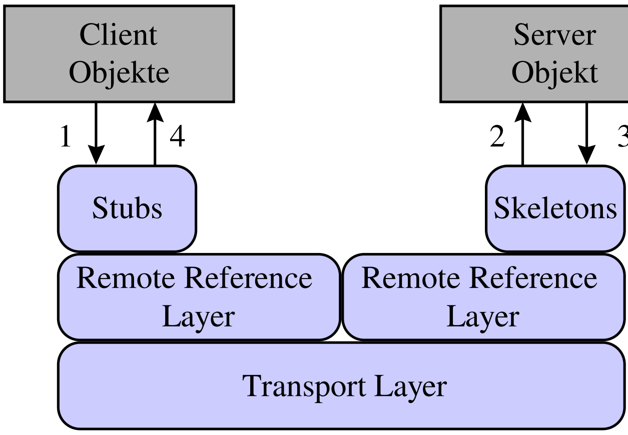

Abbildung 41: Dreischichten Modell bei RMI

Der Ablauf eines Aufrufs ist dabei folgendermaßen:

1. Stub: Parameter verpacken (Marshaling) und Aufruf weiterleiten
2. Skeleton: Parameter aus dem Aufruf extrahieren (Unmarshaling) und an das Serverobjekt weiterleiten
3. Skeleton: Rückgabewerte verpacken und Antwort weiterleiten
4. Stub: Rückgabewerte aus der Antwort extrahieren

Für RMI existiert ein eigener `SecurityManager`. Dieser muß dem System zu Beginn bekanntgegeben werden, um die Kontrolle über die über das Netz gehenden Methodenaufrufe zu erhalten. So kann er überwachen, von welchen Servern welche Stub-Klassen geladen werden dürfen und welche Rechte Stub-Objekte haben (Zugriff auf lokale Platten etc.). Reichen diese Sicherheitsmaßnahmen nicht aus, so kann auch ein eigener `ClassLoader` eingesetzt werden, der die applikationsspezifischen Sicherheitsgegebenheiten berücksichtigt.

Auch bei entfernten Objekten ist die Garbage Collection in Java automatisiert. Wenn ein Client eine Referenz auf ein entferntes Objekt erhält, wird ein Referenz-Zähler erhöht. Benötigt der Client diese Referenz nicht mehr, wird der Zähler erniedrigt. Es kann allerdings zu Problemen kommen, wenn die Netzwerkverbindung zusammenbricht. Dadurch kann es vorkommen, daß ein Objekt zu früh gelöscht wird. Da Java keine Möglichkeit zum automatischen Rebind bietet, kann die referenzielle Integrität nicht sichergestellt werden [Java RMI 97]. In solchen Fällen kommt es zu einer Remote Exception.

5.3 Vergleich von CORBA und RMI

Der Hauptunterschied zwischen RMI und CORBA ist die Sprachunterstützung. Während RMI rein auf Java basiert und daher alle Vorteile der Sprache (z.B. Garbage Collection) ausnutzen kann, unterstützt CORBA verschiedene Programmiersprachen. Dadurch ist CORBA zwar universeller einsetzbar als RMI, muß andererseits aber auf Interoperabilität mit anderen Systemen achten.

Beim Einsatz von RMI entfällt die Verwendung einer zusätzlichen Interface Definition Language und damit auch die Übersetzung der Java-Datentypen in CORBA-Datentypen. Mittlerweile gibt es für CORBA mit „Caffeine“ auch eine Implementation, die aus dem Java Code vollautomatisch Stubs und Skeletons erzeugt und damit die IDL dem Benutzer transparent werden läßt [Orfali & Harkey 97].

Bei einem in [Orfali & Harkey 97] durchgeführten Performanz-Test schnitt RMI ca. 42% schlechter ab, als CORBA. Interessant ist hierbei, daß RMI über ein Netzwerk ca. 5% schneller war, als auf dem lokalen Rechner. Generell ist bei einem solchen Vergleich jedoch zu beachten, daß sich auch die verschiedenen Implementationen von CORBA selbst in ihrer Performanz unterscheiden.

Ein großer Vorteil von CORBA sind die spezifizierten Services, wobei hier der Implementationsgrad berücksichtigt werden muß. Die momentan von RMI gebotenen Dienste bieten dagegen nur das Notwendigste. Für die Zukunft soll jedoch auch RMI diese Dienste unterstützen [Caribou 97]. Dann hängt es nur noch davon ab, wie schnell beide Architekturen sämtliche Services implementieren können.

Der Einsatz von CORBA oder RMI hängt von den Anforderungen ab. Handelt es sich um ein heterogenes System mit verschiedenen Programmiersprachen, so wird CORBA benötigt. In einem homogenen Java-System ist kann es durchaus sinnvoll sein, RMI einzusetzen [Caribou 97]. Für einen Einsatz im COLIBRI zum Aufruf entfernter Komponenten eignet sich eher RMI, da die Komponenten und das System komplett in Java implementiert sind.

5.4 DCE

Der Vollständigkeit halber wollen wir hier kurz auf *das Distributed Computing Environment* (DCE) der *Open Software Foundation* (OSF) eingehen.

DCE unterstützt die Konstruktion und Integration von C-basierten Client-Server-Applikationen in heterogenen verteilten Systemen [Brando 95]. Kernstück der DCE-Architektur ist der Remote Procedure Call (RPC). Der RPC ist ein Programmaufruf zwischen Anwendungen in unterschiedlichen Prozeßräumen auf verschiedenen Rechnern. Dabei wird zunächst der Server lokalisiert, der die gewünschte Prozedur zur Verfügung stellt. Danach werden der Aufruf und die Parameter nach einem einheitlichen Schema verpackt (Marshaling) und vom Client an den Server übertragen. Dort werden die Argumente entpackt (Unmarshaling) und die Prozedur ausgeführt. Das Ergebnis dieses Aufrufs wird verpackt und an den Client gesendet, wo es wieder entpackt wird [Schwarz 97].

Die von einem Server zur Verfügung gestellten Prozeduren werden mit Hilfe der Interface Definition Language (IDL) beschrieben. Aus dieser Schnittstellenbeschreibung werden durch einen Compiler Stubs generiert, die für das Marshaling und Unmarshaling verantwortlich sind. Die IDL ist der von CORBA ähnlich, basiert aber auf C.

Auf der DCE Executive Ebene werden folgende Services zur Verfügung gestellt: der *Directory Service*, der *Distributed Time Service*, der *Security Service* und der *Thread Service*.

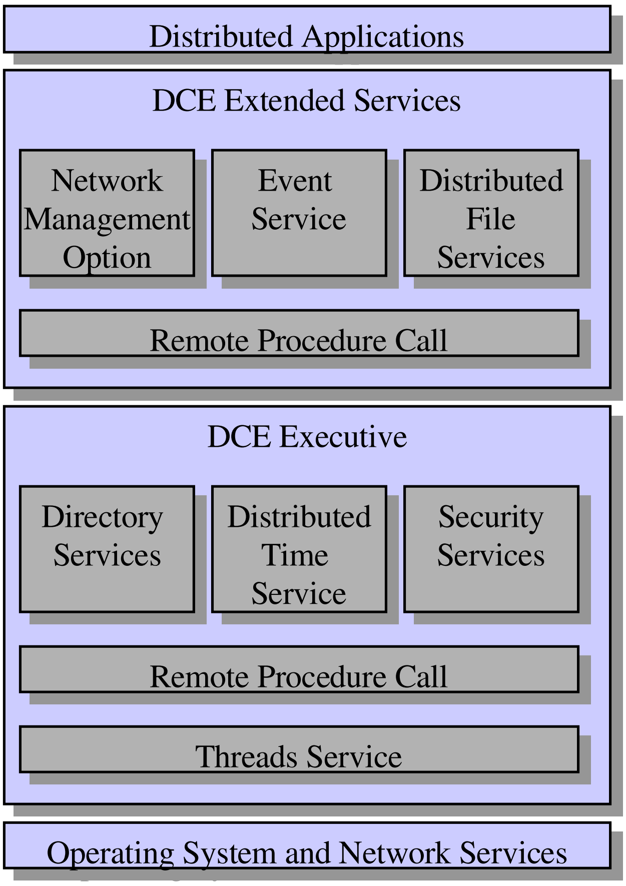

Abbildung 42: OSF DCE Architektur

5.4.1 Vergleich zu CORBA

Der Hauptunterschied zwischen DCE und CORBA liegt darin, daß DCE für die prozedurale Programmierung entworfen wurde, während CORBA objektorientiert ist [Brando 95]. DCE stellt keine Mechanismen wie Kapselung, Vererbung oder Polymorphismus zur Verfügung. Außerdem erlaubt es im Gegensatz zu CORBA nur statische Interfaces. CORBA bietet durch seine Object Services eine Reihe von Diensten an, von denen DCE nur einen Teil abdeckt. Allerdings sind die CORBA Dienste bisher nur teilweise spezifiziert und implementiert. Ein Vorteil von DCE ist die mitgelieferte Referenzplattform der DCE. Diese vereinfacht die Interoperabilität zwischen Produkten verschiedener Hersteller. Bei CORBA ist eine Interoperabilität erst seit Version 2.0 gegeben.

DCE hat durch den großen Erfolg von CORBA an Bedeutung verloren und wurde hier nur der Vollständigkeit halber erwähnt. Eine differenziertere Betrachtung findet sich in [Brando 95], [Griffel 96] und [Yang & Duddy 96].

5.5 NCSTRL

5.5.1 Was ist NCSTRL

Die Networked Computer Science Technical Report Library (NCSTRL) ist eine verteilte elektronische Bibliothek zur Verwaltung von Forschungspapieren im Bereich der Informatik [Davis & Lagoze 96]. Sie soll hier im Hinblick auf Suchmechanismen und Indizierung von Objekten untersucht werden. NCSTRL basiert auf `Dienst`, einer Implementation für verteilte digitale Bibliotheksserver. Obwohl der Schwerpunkt von NCSTRL auf Forschungspapieren im Bereich der Informatik liegt, ist das System jedoch prinzipiell für andere Schwerpunkte genauso geeignet. Eine Site kann entweder einen vollwärtigen NCSTRL-Server installieren oder lediglich einen FTP-Server, auf dem die Dokumente liegen, die aber von einem anderen Server aus verwaltet werden.

Das Dienst zugrundeliegende Dokumentenmodell ist in Abbildung 41 zu sehen. Für ein Dokument gibt es einen eindeutigen Dokumentennamen, die `docid`. Diese besteht aus dem Verleger und einem String. Der Verlegername ist weltweit eindeutig und wird von einer zentralen Stelle vergeben und verwaltet. Ein Verleger gehört immer zu genau einer Site. Der String ist eine für diesen Verleger eindeutige Bezeichnung. Diese kann von der Site frei gewählt werden. Es wird von NCSTRL keine Verzeichnishierarchie erzwungen. Dies wird dem Administrator des jeweiligen Servers überlassen. Zu einem Dokument gibt es jeweils verschiedene Anzeigeformate. So dient ein *gif*-Bild dazu, das Dokument online zu lesen. *Thumbnails*, also kleine Bilder der einzelnen Seiten, dienen zum Browsen des gesamten Dokuments. Hoch-auflösende *tiff*-Bilder liefern sehr gute Qualität und dienen zum Archivieren. *PostScript* dient zum Herunterladen und Drucken des Dokuments. Neben den verschiedenen Formaten gibt es auch verschiedene Anzeigeformen. Zum einen kann das Dokument physikalisch eingeteilt werden, z.B. in Seiten. Zum anderen ist eine logische Einteilung nach Kapiteln oder Sektionen möglich.

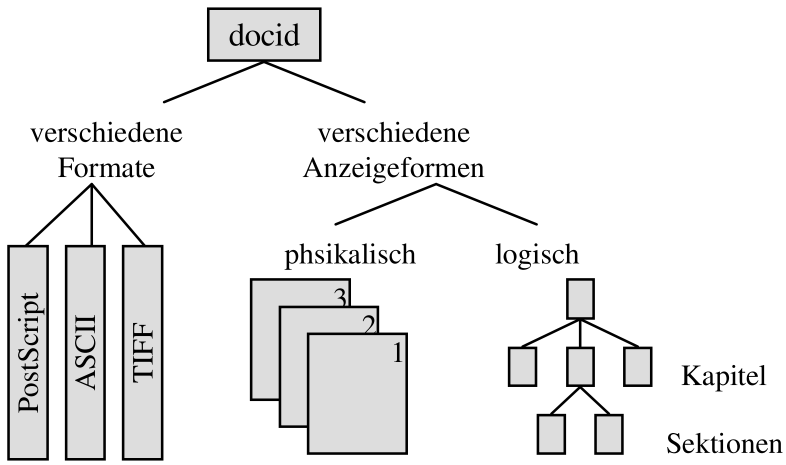

Abbildung 43: Dienst-Dokumentenmodell

Ein NCSTRL-Bibliotheksserver bietet drei verschiedene Dienste an:
- *repository service*, Speicherung und Zugang zu Dokumenten
- *index service*, Suche über Dokumente
- *user interface service,* Benutzerschnittstelle

Zur Verwaltung der Verleger existiert eine zentrale Datenbank, welche regelmäßig von den Index-Servern heruntergeladen wird. Sie bildet die docids auf die entsprechenden Server ab, auf denen sich die jeweiligen Dokumente befinden. Die Suche nach einen konkreten Dokument erfolgt dann über diese Datenbank. Zur Indizierung der Dokumente wird ein standard Buchlisten-Format (RFC1375) verwendet. Dieses ermöglicht eine Suche basierend auf bibliographischen Feldern, wie Autor, Titel oder Abstract.

Eine allgemeine Suchanfrage generiert parallele Anfragen an die verteilten Index-Server. Ein ausgewählter Index-Server fungiert dabei als Suchmaschine für die Dokumente auf den FTP Servern. Jeder Index-Server liefert eine Liste von docids zurück, die auf die Suchanfrage passen, sowie einen Vermerk auf die Server, die nicht geantwortet haben. Das Ergebnis aller Server wird geparst und als Hypertext-Liste (siehe Abbildung 43) angezeigt. Für den Benutzer entsteht der Eindruck einer einzelnen Sammlung von Dokumenten, obwohl diese über viele verschiedene Rechner verteilt liegen. Die Verweise der einzelnen docids gehen direkt auf die entsprechenden Server, auf denen das jeweilige Dokument liegt. Der Benutzer kann nun ein Dokument auswählen. Auf der folgenden Seite bekommt er über das Dokument Informationen und die möglichen Anzeigeformate angezeigt. Der Zugriff auf ein Dokument erfolgt direkt über den Server, auf dem das Dokument sich befindet. Dieser wird über die docid und die Zuordnungstabelle ermittelt. Die Serverkommunikation bleibt für den Benutzer transparent.

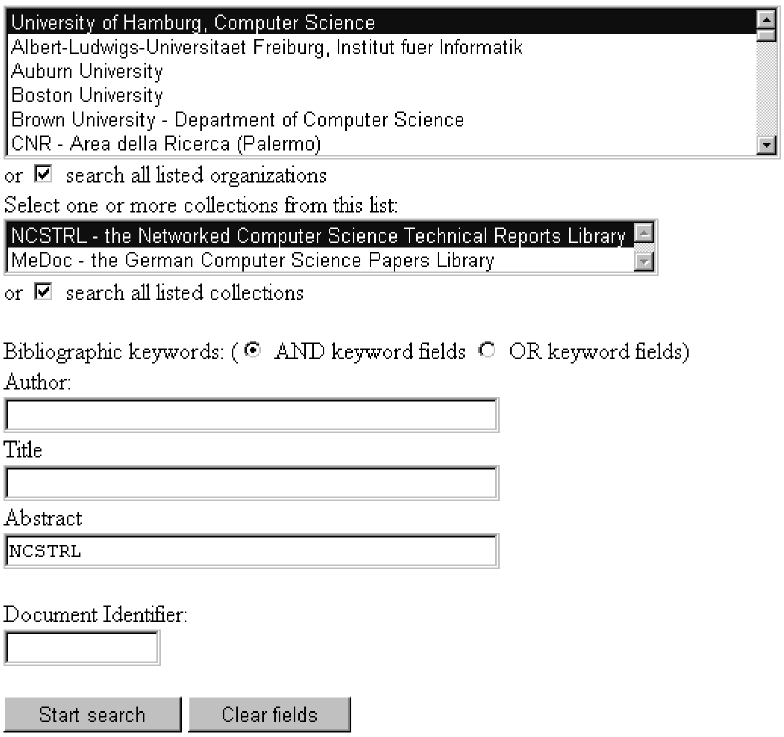

Abbildung 44: Suchformular NCSTRL

NCSTRL verwendet bekannte Internettools zur Suche, zum Browsern, Lesen und Herunterladen von Dokumenten. Ein Zugriff auf das System ist über jeden WWW-Client möglich. Die Suchanfrage läuft über ein HTML-Formular.

Search Results

Search fields:

organizations selected:

- *University of Hamburg, Computer Science (NCSTRL collection)*
- *CNR, Pisa, Italy (NCSTRL collection)*
- *Chinese University of Hong Kong (NCSTRL collection)*
- *Cornell University (NCSTRL collection)*
- *Massachusetts Institute of Technology, Laboratory for Computer Science (NCSTRL collection)*
- *Queensland University of Technology. Faculty of Information Technology (NCSTRL collection)*
- *Stanford University (NCSTRL collection)*
- *Universitaet Freiburg (MeDoc collection)*
- *University of California, Berkeley (NCSTRL collection)*

abstract = *NCSTRL*

Search Summary:

Organizations you selected are listed below by number of titles found.

- *(2)University of Hamburg, Computer Science (NCSTRL collection)*
- *(1)Cornell University (NCSTRL collection)*

Search Results:

- University of Hamburg, Computer Science (NCSTRL collection)
  - *Zugang zu Dokumenten in verteilten elektronischen Bibliotheken*. Stephan Rücker. (Doc-002)
  - *Grey literature and multiple collections in NCSTRL*. Steven Adler, Uwe Berger, Anne Brüggemann-Klein, Cornelia Haber, Winfried Lamersdorf, Malte Münke, Stephan Rücker and Hendrik Spahn. (Doc-001)
- Cornell University (NCSTRL collection)
  - *The Networked Computer Science Technical Report Library*. James R. Davis and Carl Lagoze. (TR96-1595)

Status messages:

The search failed for the following organizations:

- **CNR, Pisa, Italy (NCSTRL collection)** - server unavailable at this time
- **Universitaet Freiburg (MeDoc collection)** - server unavailable at this time

Abbildung 45: Suchergebnis einer NCSTRL-Anfrage

Das System basiert auf HTTP als Transportschicht. Daher ist neben dem Dienst-Server noch ein zusätzlicher WWW-Server erforderlich. Für den Dienst-Server existiert ein eigenes Protokoll [Lagoze et al. 95]. Dienst-Anfragen werden in den Pfad der URL hineinkodiert [Davis & Lagoze 94]. Der WWW-Server muß bei einer Dienst-URL einen CGI-Stub aufrufen, der die Anfrage extrahiert und an den Dienst-Server weiterleitet. Da NCSTRL ein offenes System ist, können auch Softwareagenten darauf zugreifen oder einzelne Komponenten der Architektur z.B. zu Forschungszwecken ausgetauscht werden

5.5.2 Eignung für Simulationsmodelle

NCSTRL wird nun im Hinblick auf die Indizierungsmechanismen, Suchfunktionen und die Serverarchitektur auf eine Eignung für Komponenten untersucht.

Das in NCSTRL verwendete Namensschema ermöglicht keine hierarchischen Namensräume. Für einen Server gibt es keine Möglichkeit, eigene Unterräume anzulegen. Ein Verleger kann jeweils nur auf einem Server publizieren. Generell werden die Namensräume von einer zentralen Stelle verwaltet. Die Zuordnungen sind dabei hardcodiert. Dies ist sehr unflexibel. Ein anderer Mechanismus, wie der Domain Name Service wäre sinnvoll. Denkbar ist hier auch, die Komponenten direkt über deren URLs zu verwalten.

NCSTRL nimmt eine Indizierung der Dokumente durch den RFC 1357, einem Standard für Bibliothekssysteme vor. Die dort spezifizierten Felder sind für Softwarekomponenten wenig geeignet, wir werden jedoch ein ähnliches Format für Softwarekomponenten verwenden.

Bei der Suche werden in NCSTRL alle dem System bekannten Server kontaktiert. Dies ermöglicht jedoch nur eine eingeschränkte Erweiterbarkeit des Systems, zumal der langsamste Server jeweils ausschlaggebend für die Antwortzeit des Systems ist. Für den Komponentenbrowser dürfte dies zunächst eher unkritisch sein, da die Anzahl der Server sich im Rahmen halten wird. Trotzdem sollten hier Erweiterungsmöglichkeiten bedacht werden. Bei NCSTRL werden alle Server von einer zentralen Stelle aus verwaltet. Dies hat Nachteile, wenn es zu einem Ausfall dieses Servers kommt. Auch hier sollten alternative Verwaltungsmöglichkeiten evaluiert werden.

Eine differenzierte Anzeige von Dokumenten nur für einen bestimmten Personenkreis oder authorisierte Benutzer ist mit NCSTRL nicht möglich. Gerade für die Verwaltung von Komponenten wäre dies jedoch wünschenswert. So sollten normale Benutzer auf einem Server nur lesende Recht haben, während der Administrator auch schreibend darauf zugreifen darf.

NCSTRL verwendet HTTP als Grundlage für die Serverkommunikation. Dies hat den Vorteil, daß auch Rechner hinter einer Firewall über einen Proxy auf das System zugreifen können (siehe auch Kapitel 7). Die Anfragen an den Dienst-Server werden in die URL hineincodiert. Diesen Mechanismus werden wir später auch zur Serverkommunikation verwenden. Bei Dienst gibt es neben dem eigentlichen Dienst-Server noch einen WWW-Server, der so den Dienst-Server vom HTTP-Protokoll entlastet.. Für den Komponentenserver werden wir einen eigenen Server implementieren, der das HTTP-Protokoll mit implementiert, da dieser dann vollständig in Java geschrieben sein wird und nicht von der CGI-Schnittstelle abhängt. Zum einen ist dies schneller als die Kommunikation über zwei Server. Zum anderen benötigen wir den WWW-Server nicht, da der Zugriff auf den Server nicht über einen Browser erfolgt.

6 Spezifikation / Konzept & Entwurf

In diesem Kapitel sollen die in Kapitel 1 genannten Anforderungen an den Browser konzeptionell dargelegt werden. Die verschiedenen Ansätze müssen dabei auf ihre Realisierungsfähigkeit und ihre Praxistauglichkeit hin untersucht werden. Die Detailfragen und die Implementation des COLIBRI werden in Kapitel 7 vorgestellt.

6.1 Java

Die Implementation von HADES in Java anstatt anderer objektorientierter Programmiersprachen (C++, Smalltalk, ...), ergibt sich aus den Anforderungen der Portabilität und Erweiterbarkeit. Die Plattformunabhängigkeit ermöglicht es den Studenten, HADES auf praktisch jedem System unverändert laufen zu lassen. Damit in Zusammenhang steht die problemlose Erweiterbarkeit, die ebenfalls ohne weitere Anpassungen oder Neu-Compilationen plattformübergreifend funktioniert. Positiver Nebeneffekt ist die minimale Installation und einfache Aktualisierbarkeit durch Updates. Ideal wäre die Verwendung als Java-Applet im World-Wide-Web. Leider ist dies angesichts instabiler und teilweise unvollständiger Java-Implementationen der derzeitigen Browser aber noch problematisch. Ein erst vor kurzem von Sun vorgestellter Interpreter „Internet Activator“ in Form eines Browser-PlugIns, könnte diese Probleme in Zukunft jedoch beseitigen.

Die Robustheit von Java gegenüber Programmfehlern kommt dem Entwicklungsaufwand entgegen. Anstatt mit nichts sagenden Fehlermeldungen abzustürzen, meldet der Java-Interpreter Fehler (Exceptions) im allgemeinen recht detailliert inklusive einer Zeilennummer, an der der Fehler auftrat. Dies verkürzt die Fehlersuche und bringt auch Fehler zum Vorschein, die sonst mangels Auswirkungen, wie z.B. Feldüberläufe, vielleicht sehr spät entdeckt worden wären. Wechselwirkungen zwischen unterschiedlichen Fehlern können so mit wenig Aufwand vermieden werden und führen zu insgesamt stabileren Systemen. Einzig relevanter Nachteil von Java ist die mäßige Performanz insbesondere im graphischen Bereich. Das Hauptanwendungsgebiet von HADES liegt jedoch in kleinen bis mittelgroßen Designs mit kurzen Simulationszeiträumen. Die Geschwindigkeit von Java liegt dabei durchaus im akzeptablen Bereich und ist in der Vergangenheit durch optimierte Interpreter und Just-In-Time-Compiler laufend gestiegen.

Grundlage jeder Java-Programmierung ist zur Zeit immer noch Suns Java-Development-Kit (JDK), das allen anderen Compiler- und Interpreter-Herstellern als Referenz dient. Das JDK bietet jedoch keine integrierte Entwicklungsumgebung, sondern besteht nur aus einzelnen Shell-basierten Programmen, von denen der Compiler durch lange Übersetzungszeiten besonders negativ hervorsticht. Dennoch garantiert die Verwendung des JDK die größtmögliche Kompatibilität. Da Java bei Sun immer noch intensiver Weiterentwicklung unterliegt, erscheinen momentan regelmäßig neue Versionen, die nicht immer nur Erweiterungen, sondern zum Teil auch Umstrukturierungen und veränderte APIs mit sich bringen. Als Literatur bei der Programmierung dienten uns neben den mitgelieferten Online-Dokumentationen und dem Standardwerk [Flanagan 96] auch [Cornell & Horstmann 96], [Arnold & Gosling 96], [Chan & Lee 96] und [Hendrich 97].

6.2 Aufbau

Um eine intuitive Bedienbarkeit zu gewährleisten, sollten in Colibri möglichst nur gängige GUI-Elemente verwendet werden. Als Grundlage der Strukturdarstellung bietet sich besonders der aus Windows bekannte Tree-View an, der eine übersichtliche und optisch ansprechende Oberfläche mit einer intuitiven Benutzbarkeit vereint. Kurz nach Beginn dieser Arbeit erschien dazu die erste Version der Java-Foundation-Classes, die mit dem `JTree` ein solches GUI-Element enthalten. Die Verwendung der JFC hat, wie in Kapitel 2 dargestellt, weiterhin den Vorteil der unterschiedlichen „Look and Feels", wodurch sich für den Benutzer die gewohnten Darstellungsweisen ergeben. Die flexible Darstellung der Elemente im Baum ermöglicht es, verschiedenartige Komponenten auch optisch voneinander trennen zu können. Komponenten die nicht lokal, sondern auf Servern liegen, können als ein eigener Teilbaum einfach integriert werden. Neben der Baumstruktur werden zu der jeweils gerade ausgewählten Komponente genauere Informationen angezeigt werden. Dazu zählen neben der graphischen Repräsentation auch eine Kurzbeschreibung, der Autor und die verfügbaren Versionen. Als Vorgriff auf Kapitel 7 zeigt die folgende Abbildung einen Screenshot des COLIBRI:

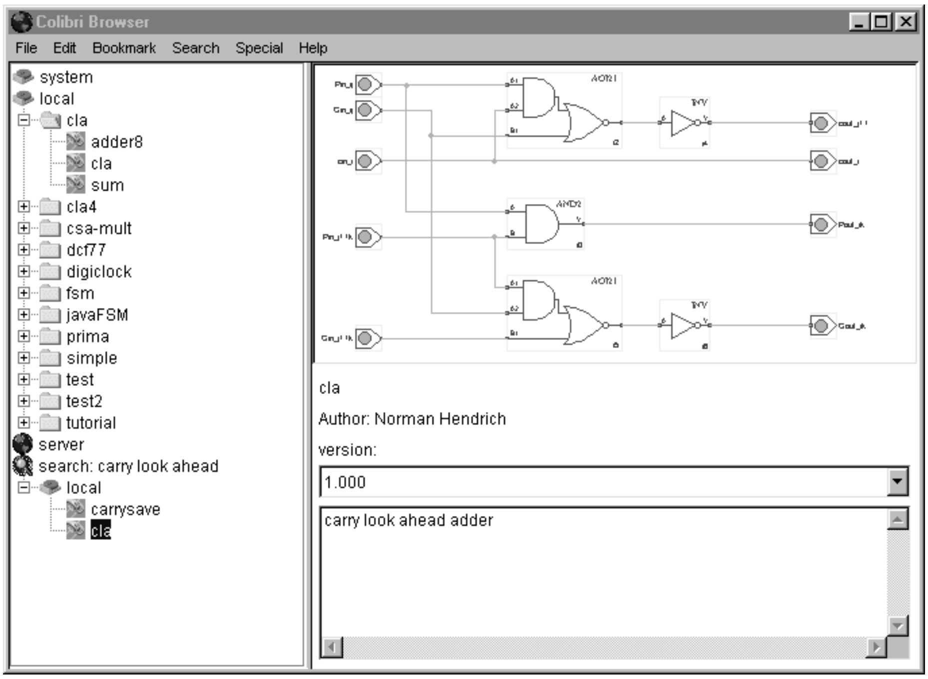

Abbildung 46: COLIBRI Fenster

Die JavaBeans Spezifikation sieht auch optionale HTML-Dateien vor, die zusätzliche Dokumentation für ein Bean bereitstellen. Daher sollte von COLIBRI auch die Möglichkeit zur Anzeige von HTML-Dateien zur Verfügung gestellt werden.

6.3 Laden von Komponenten

6.3.1 Übersicht

Sollen Komponenten aus dem Browser in einen Editor eingefügt werden, so bietet es sich an, dies durch einen einfachen doppelten Mausklick auszuführen. Während dies bei normalen Komponenten eindeutig ist, kann die Unterstützung hierarchischer Designs von HADES zu Unklarheiten führen. Designs können dabei nämlich nicht nur geladen, sondern wie andere Komponenten auch, in ein Design eingefügt werden. Bei einem Doppelklick auf ein Design ist deshalb nicht mehr klar, was mit diesem geschehen soll. Weiterhin läßt sich die Benutzung von COLIBRI in zwei Bereiche aufteilen. Zum einen soll mit dem Browser allgemein „gestöbert“ werden können, um z.B. neue Bibliotheken zu erkunden. Andererseits kann der Browser aber auch eine Dialog-Funktion bekommen, wenn vom System erwartet wird, etwas auszuwählen.

6.3.2 Einfügen oder Öffnen ?

Wie bereits in der Einleitung erwähnt, müssen beim Laden von Designs grundsätzlich zwei Szenarien unterschieden werden: Entweder das gewählte Design wie eine Komponente in ein bereits geöffnetes Design eingefügt, oder es wird als zu bearbeitender Entwurf geöffnet. Die sicherlich häufiger auszuführende Aktion ist das Einfügen; sie sollte daher besonders einfach auszulösen sein. Das Laden eines Designs ist demgegenüber zwar ebenfalls häufig, aber dennoch der seltenere Fall. Deshalb wird der auf allen Plattformen bekannte doppelte Mausklick für das Einfügen von Komponenten benutzt. Da es keine vergleichbar gängige Mausaktion gibt, muß für das Laden von Designs auf das Kontextmenü ausgewichen werden. Speziallösungen, die z.B. auf einer dritten Maustaste beruhen, sollten angesichts der angestrebten Plattformunabhängigkeit und Benutzerfreundlichkeit nicht verwendet werden.

Um verschiedene Benutzergewohnheiten zu unterstützen, sollen nach Möglichkeit auch die gängigen Mechanismen wie „Drag and Drop“ und „Copy and Paste“, sowie eine Tastaturbedienung ermöglicht werden. Eine integrierte „Copy and Paste“-Unterstützung bietet Java für Texte (Strings) und serialisierte Objekte an, wobei der Austausch von Texten am problemlosesten funktioniert. Um Komponenten über das Clipboard einzufügen, bieten sich die Objekt-Serialisierung und Referenzen in Form von URLs an. „Drag and Drop“ wird derzeit in Java implementiert, erfordert aber eine native Unterstützung und ist deshalb noch nicht durchgängig verfügbar. Trotzdem macht es aber mehr Sinn, auf die endgültige Implementation zu warten und diese zu verwenden, anstatt eine eigene proprietäre Lösung zu entwickeln.

6.3.3 Allgemeines Browsen vs. spezielle Auswahl

Die Benutzung von COLIBRI läßt sich in zwei verschiedene Funktionalitäten aufteilen. Der erste Anwendungsfall ist das allgemeine Suchen von Komponenten. Dabei wird nach ganz konkreten oder auch nur interessant erscheinenden Komponenten gesucht. Bei Bedarf werden diese dann eingefügt bzw. geladen. Im Gegensatz dazu steht die Dialog-Funktionalität des Browsers, bei der eine ganz spezielle Auswahl getroffen werden soll. Vergleichbar ist dies mit einem File-Requester, in dem der Benutzer eine konkrete Auswahl treffen muß. Während einer solchen Auswahl bleibt üblicherweise die gesamte Applikation gesperrt (modaler Dialog).

Da COLIBRI von Natur aus zum Laden von Komponenten dient, ist eine Dialog-Funktionalität nur für das Speichern notwendig. COLIBRI bietet deshalb keinen separaten Dialog für das Laden an. Die Konsequenz aus dieser Vorgehensweise ist, daß bei der Verwendung des COLIBRI als Dialog lediglich das Fenster in den Vordergrund springt;

die für Dialog-Fenster üblichen Buttons zum Bestätigen oder Abbrechen gibt es jedoch nicht. Auch die auslösende Applikation wird deshalb nicht gesperrt.

6.3.4 Globaler Browser vs. verschiedene Instanzen

Von einem HADES-Editor kann es mehrere verschiedene Instanzen geben. Für COLIBRI stellt sich daher die Frage, ob es von ihm ebenfalls mehrere Instanzen (also mehrere Fenster) geben soll oder ob es statt dessen nur einen einzigen Browser gibt. Ordnet man jedem Editor genau einen Browser zu, ergibt sich für das System eine klare Zuordnung. Der Benutzer verliert dabei aber unter Umständen schnell die Übersicht. Je mehr Fenster geöffnet sind, desto gravierender wird dieses Problem. Werden in einem Browser Einstellungen verändert oder Komponenten gesucht, tritt bei mehreren Browsern außerdem das Problem von Inkonsistenzen auf.

Existiert nur ein einziger „globaler“ Browser, vermeidet man zwar die Inkonsistenzen, dafür gibt es bei mehreren geöffneten Editoren aber keine eindeutige Zuordnung mehr. Wird in COLIBRI dann eine Komponente durch einen Doppelklick ausgewählt, gibt es keine sichere Methode zu entscheiden, in welchen Editor diese eingefügt werden soll. Eine akzeptable Lösung dafür ist es, den zuletzt aktiv gewesenen Editor auszuwählen. Bei der Verwendung von „Drag and Drop“ oder „Copy and Paste“ tritt dieses Problem nicht auf.

6.3.5 ClassLoader vs. CLASSPATH

Soll eine Komponente geladen werden, müssen die zugehörigen Klassen irgendwie in die Virtual Machine geladen werden. Normalerweise geschieht dies über einen im System integrierten ClassLoader. Dieser sucht die benötigten Klassendefinitionen in allen Verzeichnissen, die in der Umgebungsvariable CLASSPATH spezifiziert sind. Da COLIBRI eine flexible Verwaltung ermöglichen soll, ist es aber fast unmöglich, alle Komponenten-Pfade dort anzugeben. Java bietet für diesen Fall die Möglichkeit, einen eigenen ClassLoader zu verwenden. Dazu gibt es die Klasse `java.lang.ClassLoader`, die als Basis dient. Subklassen müssen dann eine Methode `loadClass()` implementieren, die anhand des Namens die gesuchten Klassendefinitionen ausfindig machen.

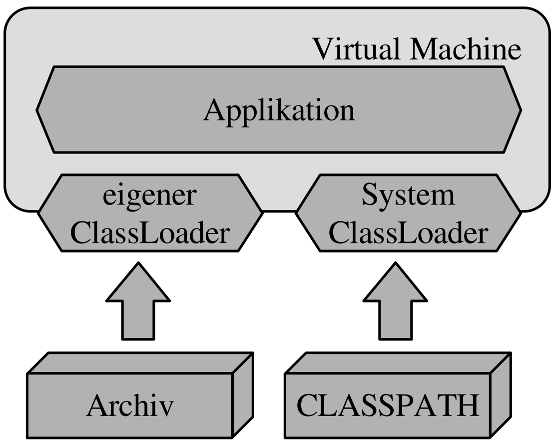

Abbildung 47: Integration eines eigenen ClassLoaders

Die Beanbox von Sun benutzt zwar einen eigenen ClassLoader, erspart sich jedoch jeglichen Verwaltungsaufwand, indem sich alle Beans in einem fest vorgegebenen Verzeichnis befinden müssen. Dieses wird nur beim Programmstart analysiert und alle darin enthalten Komponenten eingelesen. Zusätzliche Installationen zur Laufzeit müssen entweder explizit eingelesen werden oder bleiben bis zu einem Neustart unberücksichtigt. Diese Lösung ist sehr unflexibel und für eine größere Anzahl von Komponenten nicht akzeptabel. Ein anderer Lösungsansatz wäre es, ein temporäres Verzeichnis in den CLASSPATH aufzunehmen und die benötigten Komponenten dort hineinzukopieren. Dies verursacht jedoch einen hohen Kopieraufwand und vergrößert das Risiko von Namenskonflikten. Insbesondere ist es dabei nicht möglich, verschiedene Versionen einer Komponente gleichzeitig zu verwenden. Da Jar-Dateien im CLASSPATH wie eigene Verzeichnisse behandelt werden, müßten sie wahrscheinlich bei jeder Benutzung entpackt werden, was diese Lösung unzumutbar werden läßt.

Für COLIBRI ist es entscheidend, daß der Benutzer eine eigene Ordnung aufbauen kann. Um z.B. Namenskonflikte zu vermeiden, muß es daher möglich sein, Komponenten aus beliebigen Unterverzeichnissen laden zu können. Dies ist insbesondere für ein Versionsmanagement unumgänglich, da hierbei im Normalfall mehrere Komponenten mit gleichen Dateinamen unterschieden werden müssen. Ein eigene Laderoutine für Klassen und damit ein eigener ClassLoader scheint in diesem Fall daher die beste Lösung zu sein. Ein ClassLoader ist mittels der seit Java 1.1 integrierten `getResource()` und `getResourceAsStream()` Methoden nicht nur für das Laden von Klassendefinitionen, sondern auch für Ressourcen wie z.B. Bilder zuständig.

Für das Laden von Klassen ist im ClassLoader immer die Methode `loadClass()` zuständig. Diese sucht normalerweise zuerst in einem Cache und erst danach im Dateisystem, auf Servern, etc. Wurde die angeforderte Klasse gefunden, so muß sie geladen und mit der Methode `defineClass()` der Virtual Machine zugänglich gemacht werden. Diese Methode ruft dann solange rekursiv `loadClass()` auf, bis die gesamte Klassenhierarchie (Super-Klassen und Interfaces) geladen ist.

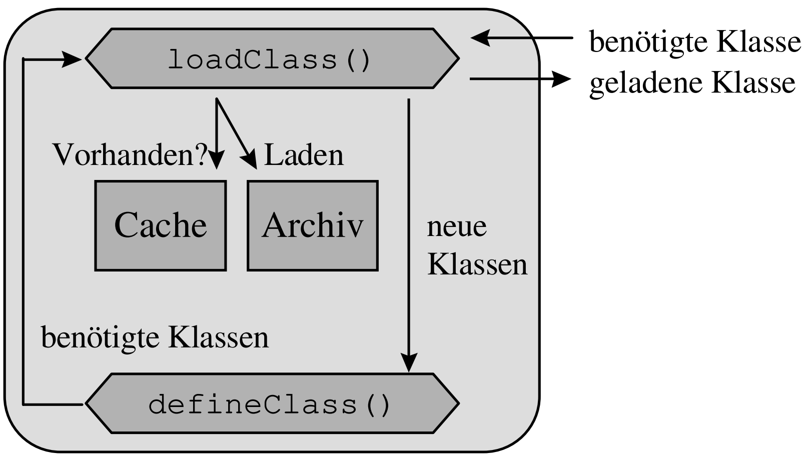

Abbildung 48: Funktionsweise eines ClassLoaders

Ein eigener ClassLoader ersetzt aber nie den System-ClassLoader, sondern ist nur für einen separaten Teil der Applikation zuständig. Da die HADES Simulationsobjekte keine 100%-igen JavaBeans (Kapitel 4.3) sind, ergibt sich aus dieser ClassLoader „Mischung“

eine besondere Problematik, auf die im nächsten Kapitel noch genau eingegangen wird. Weiterhin macht es Sinn, nicht alle Komponenten von nur einem einzigen ClassLoader laden zu lassen. Entsprechend den Paradigmen der objektorientierten Programmierung ist es sinnvoller, für jede Komponente einen eigenen ClassLoader zu verwenden. Auf diese Weise muß bei einem Aufruf nicht erst aufwendig die jeweilige Komponente identifiziert und ihre Quelle ermittelt werden. Da sich die verschiedenen Instanzen einer Komponente einen ClassLoader teilen können, ist der entstehende Speicher-Overhead durchaus vertretbar.

Um COLIBRI während des Betriebs möglichst weit von der restlichen Applikation (HADES) zu entkoppeln, macht es Sinn, beim Laden von Komponenten bereits zu Anfang alle zugehörigen Dateien einzulesen. Dadurch ergibt sich für jede Komponente eine eigene „Sandbox“, aus der im Prinzip keine weiteren Datei- bzw. Netzwerkzugriffe mehr stattzufinden brauchen. Dadurch ergibt sich weiterhin die Möglichkeit, Komponenten von Servern mittels eines serialisierten ClassLoaders zu übertragen. Auf diese Art werden dauernde kleine Netzübertragungen (wie bei HTTP 1.0) vermieden, was die Performanz verbessern sollte. Wichtige Voraussetzung ist allerdings das Bekanntsein aller zugehörigen Dateien. COLIBRI speichert diese Liste und weitere Informationen in speziellen Dateien, die die Endung „.clb“ tragen. Auf diese Dateien wird später (Kapitel 6.5) genauer eingegangen.

Um Komponenten nicht nur lokal, sondern auch im Netzwerk eindeutig identifizieren zu können, bietet sich die Verwendung von URLs an. Die in Java enthaltene Netzwerkfunktionalität läßt sich dazu um eigene Protokolle erweitern. Wie dieser Mechanismus genau funktioniert, wird in Kapitel 7 beschrieben. Ein eigenes Protokoll wird auch in der Methode `getResource()` des ClassLoaders benötigt. Sie liefert eine URL zurück, bei der mittels `getContent()` das Zielobjekt automatisch geladen werden kann. Für COLIBRI muß sich dazu die URL-Quelle (Komponente) rekonstruieren lassen, wofür ein eigenes Protokoll unumgänglich ist.

6.3.6 Editierbare Komponenten

Eine besonders komplizierte Struktur ergibt sich bei editierbaren Komponenten wie z.B. endlichen Automaten (FSM - Finite State Machine). Diese legen ihre Daten in einer separaten Datei ab, mit der sie beim Laden wieder initialisiert werden müssen. Der Problempunkt liegt nun darin, daß sich diese Dateien an verschiedenen Orten befinden können. Sie können sowohl als ein Bestandteil (Ressource) eines Designs abgespeichert sein, als auch eine eigene externe „Pseudo“-Komponente darstellen. Dazu ein Beispiel:

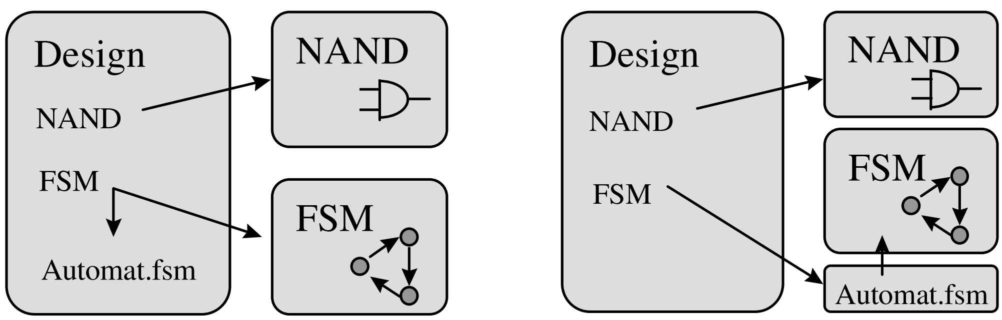

Abbildung 49: FSM als Ressource

Abbildung 50: FSM als Komponente

Ein Design enthält einen endlichen Automaten und daher auch eine Referenz auf die FSM-Komponente. Der gespeicherte Automat befindet sich in einer Datei „Automat.fsm“, die im ersten Fall als Ressource im Design integriert ist. Die Referenz auf die FSM muß deshalb auch den Namen dieser Datei enthalten, damit der Automat wieder korrekt initialisiert werden kann. Im zweiten Fall stellt der Automat eine externe Komponente dar. Da diese aber nur aus einer Ressource-Datei ohne Programm-Code besteht, muß sie eine Referenz auf die eigentliche Komponente, also die FSM, enthalten. Das Design zeigt dabei nur die externe „Pseudo“-Komponente.

Letztlich sind auch die von HADES erzeugten Designs solche „Pseudo“-Komponenten. Sie enthalten keinen Code, sondern nur eine Definitionsdatei, mit der die Komponente „Design“ initialisiert wird. Die Klasse `hades.models.Design` unterscheidet sich jedoch dahingehend, daß sie fester Bestandteil von HADES, und damit keine eigenständige Komponente ist.

6.4 Speichern von Komponenten

Zunächst einmal kann HADES keine wirklichen Komponenten erzeugen, sondern nur die eben erwähnten „Pseudo“-Komponenten. Da diese aber auch aus mehreren Dateien bestehen können, unterscheidet sich das Speichern dabei nicht vom Speichern „echter“ Komponenten. Eine automatische Speicherung in Archiven ist angesichts der relativ kleinen Dateien nicht dringend notwendig und wird deshalb vorerst nicht unterstützt. Insbesondere für ältere Versionen von Komponenten bietet COLIBRI dem Benutzer jedoch die Möglichkeit einer späteren Archivierung.

Für das Speichern stellt COLIBRI einen eigenen Dialog zur Verfügung, da im Vergleich zum allgemeinen Browser einige Veränderungen vorgenommen werden müssen. Zunächst einmal sollen Komponenten nicht überall gespeichert werden können, sondern nur im lokalen Benutzerverzeichnis und eventuell auf bestimmten Servern. Der Save-Dialog zeigt daher auch nur diese Verzeichnisse an. Weiterhin sollen Informationen wie die Name, Versionsnummer, Autor, etc. nicht nur angezeigt, sondern auch veränderbar sein. Hinzu kommen die für Dialoge typischen Buttons zum Bestätigen oder Abbrechen:

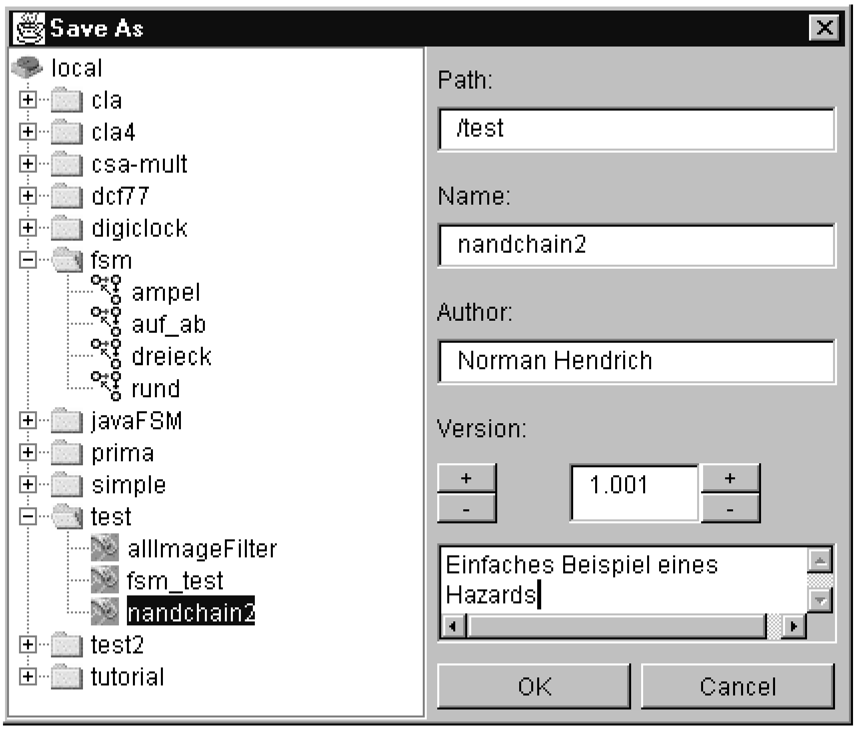

Abbildung 51: Save-Dialog

Wird eine neue Komponente generiert, so muß COLIBRI eine neue CLB-Datei (siehe Abschnitt 6.6) erzeugen, die alle zum Laden wichtigen Informationen enthält. Schwierig wird es hierbei, die Liste der zugehörigen Dateien zu ermitteln, da diese nicht von Anfang an feststeht, sondern nach und nach für jede Ressource ergänzt werden muß. Jedesmal wenn eine Komponente eine Ressource-Datei abspeichern möchte, muß das umgebene Design und der zugehörige Archiv-Eintrag mit Versionsnummer von COLIBRI ermittelt werden. Erst dann kann die neue Ressource dem Design hinzugefügt werden.

Beim Speichern läßt COLIBRI von HADES ein kleines Übersichtsbild (Thumbnail) erzeugen, das im Browser zur besseren Übersicht und Wiedererkennung angezeigt wird. Da dies für HADES etwas Aufwand bedeutet (2x Zoomen und Neuzeichnen) kann das Speichern von größeren Designs einen kleinen Moment dauern. Verschiedene Versionen werden getrennt voneinander gespeichert und können daher nach Belieben erzeugt werden. Alle Versionen haben aber die im Browser angezeigten Informationen gemeinsam. Unterschiedliche Beschreibungen, Autoren, Bilder, etc. sind deshalb nicht möglich. Da die Komponentennamen nicht nur zum Speichern, sondern auch zum Referenzieren von Komponenten verwendet werden, ist eine spätere Namensänderung aus Konsistenzgründen nicht mehr möglich. Statt dessen kann eine Komponente jedoch jederzeit unter einem anderen Namen abgespeichert werden.

Beim Speichern eines Designs wird zuerst die Design-Datei (Endung „.hds“) gesichert, die bereits alle Referenzen auf enthaltene Komponenten besitzt. Danach werden rekursiv alle Komponenten aufgefordert, ihre Daten ebenfalls zu sichern. Dazu stellt bei Bedarf jede Komponente eine Anfrage im `DesignManager`, der das Bindeglied zwischen HADES und COLIBRI darstellt. Dieser liefert daraufhin einen OutputStream zurück, in den die Komponente ihre Daten schreibt. Um zukünftig Daten auch auf Servern sichern zu können, ist diese Methode jedoch nicht ausreichend. Bei Netzwerkverbindungen werden die Daten im allgemeinen in Protokolle eingebettet, von denen die Komponenten aber keine Kenntnis haben sollen. Diese Protokolle (z.B. HTTP) beschreiben unter anderem auch die zu übertragenden Daten näher. Intern verwendet COLIBRI deshalb Byte-Arrays und kann so bereits vor der Übertragung wichtige Informationen wie z.B. die Länge ermitteln. Als Bindeglied zu HADES fungiert deshalb die Klasse `ColibriOutputStream`, die einen gepufferten OutputStream darstellt. Alle in diesen Stream geschriebenen Daten werden in einem Byte-Array abgelegt und erst beim Schließen als fertiges Datenpaket an COLIBRI übergeben. Aus diesem Grunde ist es wichtig, daß alle Komponenten ihre erhaltenen Streams wieder ordnungsgemäß schließen.

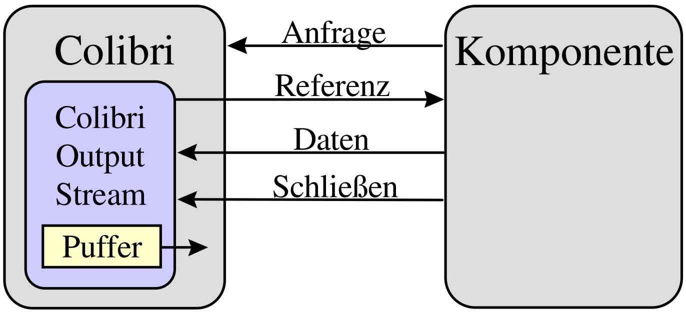

Abbildung 52: ColibriOutputStream

6.5 Verwaltung von Komponenten

6.5.1 Einzelkomponenten

Wie bereits mehrfach erwähnt, liegt eine der Hauptaufgaben von COLIBRI in der flexiblen Verwaltung von Komponenten. Dazu sollen Benutzer beliebige Strukturen („Bibliotheken“) anlegen können, in die die Komponenten einsortiert werden. Neben einzelnen Komponenten, sollen auch ganze Bibliotheken installiert werden können, worauf später genauer eingegangen wird. Insgesamt zählen folgende Punkte zur Komponenten-Verwaltung:

- Verschieben von Komponenten / Verzeichnissen
- Umbenennen von Komponenten / Verzeichnissen
- Löschen von Komponenten / Verzeichnissen
- Anlegen neuer Verzeichnisse
- Import von Komponenten / Verzeichnissen (Bibliotheken)
- Export von Komponenten / Verzeichnissen (Bibliotheken)

Beim Verschieben von Komponenten werden alle zugehörigen Dateien und alle verfügbaren Versionen in ein anderes Verzeichnis verschoben. Dabei ist zu beachten, daß Referenzen auf diese Komponenten anschließend ins Leere zeigen. Anstatt nun alle Designs nach solchen Referenzen zu durchsuchen, ist es effizienter, erst beim Laden von Designs die nicht mehr gefundenen Komponenten anhand ihres Namens wieder aufzufinden. Beim Speichern werden dann alle Referenzen automatisch aktualisiert. Beim Umbenennen von Komponenten kommt man am aktiven Durchsuchen und „Verbiegen“ von Referenzen nicht mehr vorbei. Ausweg wäre hier die Verwendung der eindeutigen Package-Namen von Komponenten. Da Designs aber nur ihren Namen besitzen, ist dies nicht durchgängig praktikabel. Aus Konsistenzgründen können Komponenten daher in COLIBRI nicht umbenannt werden! Designs können aber natürlich jederzeit manuell unter einem anderen Namen abgespeichert und dadurch auch umbenannt werden. Beim Umbenennen von Verzeichnissen werden die darin enthaltenen Komponenten praktisch in ein neues Verzeichnis verschoben, weshalb für diesen Fall das weiter oben gesagte gilt.

Der Import von neuen Komponenten kann sowohl aus dem lokalen Dateisystem als auch von Servern geschehen. Der Import von Servern gestaltet sich recht einfach, da die Komponenten dort bereits unter COLIBRI installiert sind und deshalb nur sämtliche Dateien übertragen werden müssen. Das Importieren von Komponenten aus dem lokalen Dateisystem ist dagegen deutlich aufwendiger, tritt dafür aber eigentlich auch nur bei neu entwikkelten Komponenten auf. Das Problem dabei ist, alle benötigten Informationen einer Komponente zu erhalten, was insbesondere für die Liste aller zugehörigen Dateien schwierig ist. Hier erweist sich die lose Spezifikation von JavaBeans als hinderlich. Beim Export sollen Komponenten oder Designs anderen Nutzern zugänglich gemacht werden, indem man sie auf einen Server kopiert. Da hierbei wie beim Import alle nötigen Informationen der Komponenten bereits bekannt sind, muß lediglich eine Schreiberechtigung überprüft werden.

Für COLIBRI stellt sich nun die Frage, wie die Verwaltungsstruktur im Dateisystem abgelegt wird. Um die Verwaltung für manuelle Eingriffe nachvollziehbar zu halten und die Entwicklung neuer Komponenten mit externen Entwicklungswerkzeugen zu ermöglichen, sollte die Struktur nach Möglichkeit auf das Dateisystem abgebildet werden. Dies gilt auch für die in Java übliche Abbildung der Package-Strukturen. Die in COLIBRI angezeigte Benutzer-Struktur wird 1:1 auf das Dateisystem abgebildet. An den Stellen, an denen Komponenten angezeigt werden sollen, befinden sich deren CLB-Dateien. Ausgehend von diesem Punkt im Verzeichnisbaum beginnt die Dateistruktur der Java Packages, die für den Benutzer jedoch verborgen bleibt. Die folgende Abbildung verdeutlicht den Zusammenhang dieser beiden Strukturen.

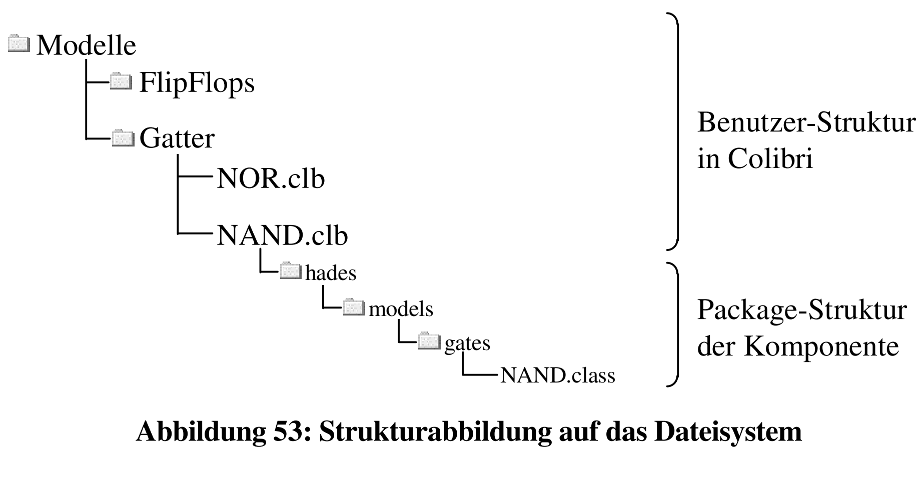

Abbildung 53: Strukturabbildung auf das Dateisystem

Durch den intensiven Gebrauch von Unterverzeichnissen ergeben sich sehr schnell lange Datei-Pfade, was auf manchen Plattformen problematisch sein könnte. Versuche, die Anzahl der Verzeichnisse zu verringern, würden durch erhöhte Undurchsichtigkeit und Inkompatibilitäten jedoch noch größere Probleme nach sich ziehen. Außerdem senken viele

Unterverzeichnisse das Risiko doppelter Dateinamen. Dieses ist besonders wichtig, wenn ähnliche Komponenten in einem Verzeichnis gesammelt werden sollen. Während es für den Benutzer noch nachvollziehbar ist, daß es in einem Verzeichnis keine zwei gleichnamigen Komponenten geben darf, ist es nicht nachvollziehbar, wenn in einem für den Benutzer unsichtbaren Teil Konflikte auftreten.

Für Komponenten, die in Jar-Dateien verpackt sind, braucht die Package-Struktur nicht mehr berücksichtigt werden, da sie bereits innerhalb des Archivs bestehen muß. Jar-Dateien liegen daher im gleichen Verzeichnis wie die CLB-Dateien der Komponenten. Bis jetzt völlig unberücksichtigt geblieben ist das Versionsmanagement, das COLIBRI ebenfalls auf das Dateisystem abbildet. Der genaue Aufbau des Versionsmanagements wird im Abschnitt 6.7 beschrieben.

6.5.2 Bibliotheken

Wie oben erwähnt sollen nicht nur einzelne Komponenten, sondern auch zusammengestellte Bibliotheken installiert werden können. Als eine solche Bibliothek können z.B. die bei HADES mitgelieferten Komponenten angesehen werden. Im Browser kann man sie sich als eigenen Teilbaum vorstellen. Für Designs innerhalb einer Bibliothek ist wichtig, daß die verwendeten Komponenten wiedergefunden werden müssen. Hierfür sollte es möglicherweise nicht nur absolute, sondern auch relative Referenzen geben, die die Position in der Bibliothek wiedergeben. Ansonsten könnte es notwendig werden, in sämtlichen Designs die enthaltenen Referenzen anzupassen. Eine weitere Möglichkeit wäre es, Bibliotheken immer auf der ersten Verzeichnisebene zu installieren, um ungültige Referenzen zu vermeiden. In diesem Zusammenhang könnte man für eine bessere Ordnung auch ein gemeinsames Unterverzeichnis „Libraries" vorschlagen, das sich dann alle Bibliotheken automatisch teilen. Eine mögliche Struktur könnte dann folgendermaßen aussehen:

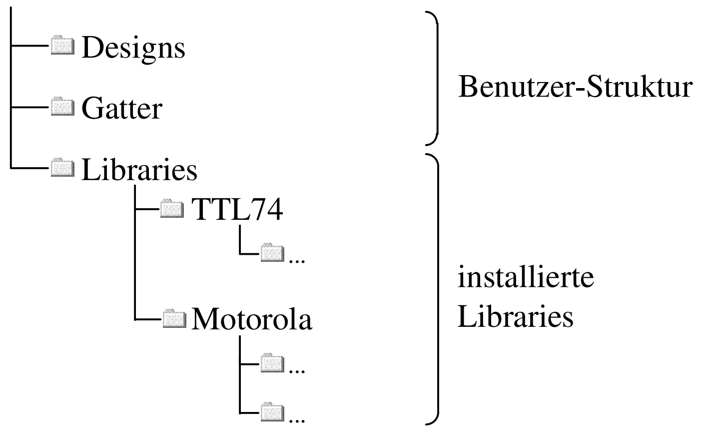

Abbildung 54: Gemeinsames Basisverzeichnis für Bibliotheken

Um z.B. die Version einer installierten Bibliothek herausfinden zu können, könnte es sinnvoll sein, Informationen über die gesamte Bibliothek zu speichern. Hierfür wäre ein Eingriff in die bisher geschilderte Struktur notwendig. Als Ersatz könnte statt dessen einfach eine leere Komponente dienen, die die gewünschten Informationen inklusive Versionsnummer enthält. Das Installieren von Bibliotheken ist damit mit dem Kopieren ganzer Teilbäume vergleichbar und muß nicht mehr gesondert berücksichtigt werden.

6.5.3 Verzeichnisse und Server

Grundsätzlich verwaltet COLIBRI alle Komponenten innerhalb eines einzigen Basisverzeichnisses. Insbesondere auf Unix-Systemen mit mehreren Benutzern macht es jedoch Sinn, zusätzlich ein gemeinsames Basisverzeichnis angeben zu können. In dieses Verzeichnis sollten dann allgemeine Komponenten und Bibliotheken installiert werden. Dieses „Systemverzeichnis" muß allen Benutzern zwar zugänglich, aber auch schreibgeschützt sein.

Server sollten als eigenständiger Hauptast im Komponenten-Baum erscheinen und auch als solcher erkennbar sein. Da Server nach Belieben hinzugefügt und entfernt werden müssen, muß eine flexible Liste verwaltet werden. Um das Eintragen neuer Server einfach zu gestalten, bietet sich das Konzept der in WWW-Browsern Verwendung findenden Lesezeichen („Bookmarks") an. Trifft man durch eine Suche auf einen interessanten Server, sollte für diesen durch ein Kontextmenü oder ähnliches ein neues Lesezeichen zu erzeugen sein.

6.6 CLB-Datei

Für die Verwaltung von Komponenten wird eine Vielzahl unterschiedlicher Informationen benötigt. Zum Teil werden diese von den Komponenten selbst bereitgestellt, zum Teil sind sie aber auch erst durch ein genaueres Untersuchen zu ermitteln. Manche Informationen lassen sich unter Umständen sogar überhaupt nicht automatisch ermitteln, weshalb eine zusätzliche Informationsspeicherung unvermeidbar ist.

Von besonderer Bedeutung für COLIBRI ist eine Liste der zu einem JavaBean gehörigen Dateien. Leider ist für die Bestimmung dieser Liste in der Bean-Spezifikation keine Möglichkeit vorgesehen, und auch das Untersuchen der Vererbungshierarchie kann nur die zugehörigen Klassen-Definitionen bestimmen. Andere wichtige Dateien (wie z.B. Icons) können von diesem Mechanismus nicht erfaßt werden. Hier zeigt sich ein Nachteil der losen Spezifikation von JavaBeans. Es ist noch nicht einmal vorgeschrieben, daß Beans sich in einem Jar-Archiv befinden müssen. Aber selbst wenn doch, ist aus der enthalten „Manifest"-Datei gerade einmal der Name der Startklasse ersichtlich. Eine zusätzliche BeanInfo-Klasse ist nur optional vorhanden, enthält aber auch kaum für die Verwaltung relevante Informationen. Sie bietet hauptsächlich für Programmierer gedachte Zusatzinformationen über das bereitgestellte Interface.

Hat man erst einmal alle relevanten Informationen über eine Komponente gesammelt, stellt sich die Frage, wo und wie diese gespeichert werden sollen. In Java bietet sich hierfür insbesondere der enthaltene Serialisierungsmechanismus an, durch den kein weiterer Verwaltungsaufwand entsteht. Allerdings lassen sich die damit gespeicherten Daten nur in Grenzen wieder zurücklesen, wenn sich die zugrunde liegenden Klassendefinitionen geändert haben. Da dies im Zuge der Weiterentwicklung nicht immer sichergestellt werden kann, wurde auf diese Möglichkeit verzichtet und statt dessen auf normale Textdateien zurückgegriffen, die die Endung „.clb" für COLIBRI tragen. Der genaue Inhalt und Aufbau dieser Dateien wird in Abschnitt 7.2.2 beschrieben. Um die in den CLB-Dateien enthaltenen Informationen zu kapseln und damit flexibel zu halten, enthält COLIBRI die Klasse `ComponentInfo`. Diese „Wrapper"-Klasse liest und schreibt die CLB-Dateien und wird intern als Komponenten-Repräsentation verwendet.

6.7 Versionsmanagement

Ein Versionsmanagement dient dazu, verschiedene Versionen eines Objektes parallel zu verwalten, um jederzeit auf die vorhergehenden Versionen zurückgreifen zu können. Das einfachste und immer funktionierende Versionsmanagement ist das manuelle Abspeichern unter verschiedenen Namen. Wesentlich anwenderfreundlicher ist jedoch ein integriertes Versionsmanagement, bei dem nicht der Benutzer, sondern das System die Verwaltung der unterschiedlichen Versionen übernimmt. Als Identifikation dient dabei idealerweise ein für alle Versionen gleicher Bezeichner. Zum Generieren neuer Versionen kommen neben der manuellen auch automatische Erhöhungen in Frage. Eine automatische Versionserhöhung (wie z.B. im Betriebssystem VMS integriert) hat den Vorteil der besseren Rekonstruktionsmöglichkeiten, da im allgemeinen deutlich mehr Versionen erzeugt werden. Durch die mangelnde Einflußnahmemöglichkeit durch den Anwender ergeben sich jedoch schlechte Versionsstrukturierungen. Vor der Verwendung des COLIBRI sicherte HADES bei jedem „Save" die alte Version durch Anhängen einer laufenden Nummer.

Im Zusammenhang mit Komponenten hat ein integriertes Versionsmanagement den Vorteil, daß laufend Updates der installierten Komponenten stattfinden können und diese ohne weiteres Zutun automatisch verwendet werden. Im Falle von auftretenden Inkompatibilitäten kann jederzeit auf die noch vorhandenen älteren Versionen zurückgegriffen werden Da Simulationsmodelle einer ständigen Evolution unterworfen sind, ist ein integriertes Versionsmanagement für HADES von besonderer Bedeutung und gehörte deshalb von vornherein zu den Hauptaufgaben von COLIBRI. Ein intelligentes Versionsmanagement, das neue Komponenten auch auf ihre Kompatibilität hin überprüft, erfordert für HADES jedoch intime Kenntnisse des Aufbaus von Simulationsobjekten. HADES Komponenten exportieren ihre Anschlüsse und Funktionalität nicht über individuelle Interfaces, sondern basieren statt dessen auf einem einzigen gemeinsamen Interface, mit dem alle Anschlußpunkte ermittelt und verwendet werden können. Eine Kompatibilitätsprüfung mit Hilfe der Introspektion würde daher zwar bei JavaBeans, nicht aber bei HADES Komponenten ausreichen.

Ein JavaBean besteht von außen betrachtet aus einer Menge von Dateien, die üblicherweise (jedoch nicht zwingend) in einem Jar-Archiv zusammengepackt sind. Ein Versionsmanagement für Beans besteht deshalb hauptsächlich darin, die üblicherweise gleichnamigen Dateien der verschiedenen Versionen getrennt voneinander zu verwalten, ohne die Gesamtbeziehung aus den Augen zu verlieren. Ein erster Ansatz für ein Versionsmanagement wäre es, alle Dateinamen um Versionsnummern zu erweitern.

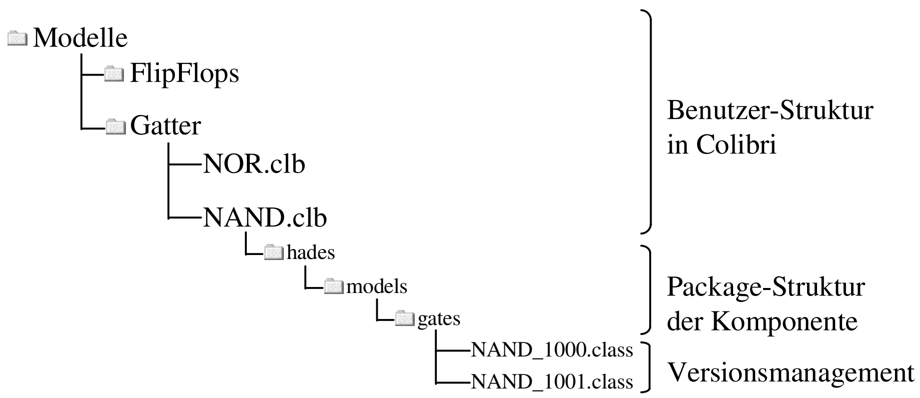

Abbildung 55: Versionsmanagement über Dateinamen

Die Veränderung der Dateinamen ist jedoch inkompatibel zu der von Java benutzten Abbildung der Package-Strukturen, wodurch die Entwicklung neuer Komponenten deutlich erschwert wird. Die von Compilern erzeugten Klassen müssen deshalb nach jedem Übersetzungsvorgang manuell oder durch Shell-Skripte erst umkopiert und umbenannt werden. Ein anderer Ansatz ist das Einfügen von Unterverzeichnissen, um verschiedene Versionen voneinander zu trennen. Eine optisch schöne und durchschaubare Kapselung ergebe sich durch das Einfügen auf letzter Ebene (im Beispiel in das „gates“ Verzeichnis). Allerdings wird auch hierbei die Package Struktur inkompatibel verändert. Viel gravierender ist jedoch, daß diese Variante nicht mehr mit in Jar-Dateien verpackten Beans funktioniert. Deutlich bessere Lösung ist deshalb das Einfügen der Versionsverzeichnisse zwischen der Benutzer- und der Package-Struktur. Neue Komponenten können dadurch direkt innerhalb der COLIBRI-Struktur ohne Kopieraufwand entwickelt werden. Nachteil an dieser Art der Implementierung ist die etwas undurchsichtigere Struktur. Alle Komponenten eines Verzeichnisses teilen sich dabei die Versionsverzeichnisse, wodurch nicht mehr direkt erkennbar ist, welche Versionen einer bestimmten Komponente existieren.

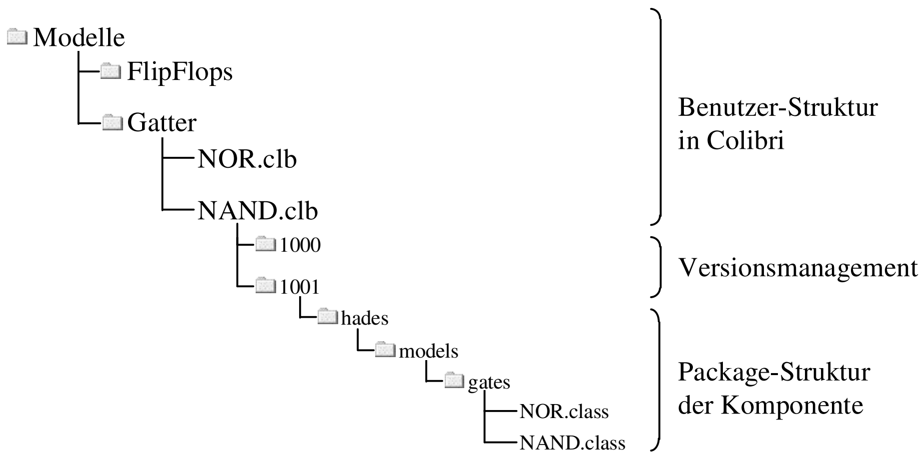

Abbildung 56: Versionsmanagement mit Verzeichnissen

Hat ein Benutzer eine Reihe von Komponenten aus dem Internet installiert, wäre es angenehm, wenn COLIBRI regelmäßig nach neuen Versionen dieser Komponenten sucht und sie ggf. auch gleich hinzu installiert. Voraussetzung dafür ist, daß COLIBRI für alle diese Komponenten den Herkunftsort kennt. Diese Information müßte dazu bei Bedarf in die CLB-Dateien aufgenommen werden. Eine andere Möglichkeit eines automatischen Updates sind sogenannte „Push“-Techniken, bei denen der Server die Clients von neuen Versionen in Kenntnis setzt. Alle interessierten Clients müssen sich dafür beim Server registrieren und eine Möglichkeit zur Kontaktierung bieten. Davon ausgehend, daß dem „Normal“-Benutzer jedoch keine dauerhafte Internetverbindung mit festen Rechner-Adressen (IP-Nummern) zur Verfügung steht, ist diese Variante eher schwierig zu realisieren und wurde deshalb nicht betrachtet.

Um gezielt ältere Versionen in Designs benutzen zu können, müssen die von COLIBRI und HADES verwendeten Komponenten-Referenzen (URLs) auch die gewünschte Versionsnummer enthalten können. Hierzu bieten sich die von URLs verwendete Syntax für Parameter an. Eine genaue Beschreibung dieser URLs findet sich in Kapitel 7.

6.8 COLIBRI Server

6.8.1 Funktionsumfang

Um Komponenten und Designs anderen Anwendern zugänglich machen zu können, war für COLIBRI von Anfang an ein Server vorgesehen. Dieser stellt die Objekte jedoch nicht nur zum Installieren bereit, sondern ermöglicht es auch, Komponenten und Designs direkt vom Server zu laden, ohne sie vorher lokal installiert haben zu müssen. Dies ist insbesondere für umfangreiche Bibliotheken oder selten benötigte Komponenten sinnvoll. Weiterhin läßt sich durch strukturierte Server eine effektive Suche realisieren. Da COLIBRI normalerweise nicht als eigenständige Applikation, sondern als ein Bestandteil einer anderen Applikation (HADES) läuft, bietet sich die Implementierung als unabhängiger Server an, der nach Belieben gestartet und gestoppt werden kann. Beim Starten sollte ihm ein Basisverzeichnis übergeben werden, dessen Inhalt er dann verwalten soll. Um auch verändernd auf den Server zugreifen zu können, ist unbedingt eine Benutzeranmeldung mit Rechtsvergabe zu implementieren. Da dies nicht nur viel Aufwand, sondern auch ein erhöhtes Sicherheitsrisiko bedeutet, kann der Server als Alternative auch ausschließlich lesende Zugriffe erlauben. Zur Administration muß in diesem Fall das Serververzeichnis als (lokales) Basisverzeichnis für COLIBRI eingesetzt werden. Da dies keine wesentliche Einschränkung bedeutet, sollte vorerst dieser Weg beschritten werden.

6.8.2 Serverarchitektur

Wie oben erwähnt, müssen für eine effektive Suche einzelne Server zu einem Verbund gekoppelt werden können. Ein guter Ansatz ist dabei das Konzept der im Internet verwendeten DNS-Server (*Domain Name Service*). Diese Server dienen ähnlich einem Telefonbuch dazu, Rechnernamen in IP-Nummern umzusetzen. Entscheidend dabei ist, daß jeder DNS-Server nicht alle Rechner weltweit kennen muß, sondern nur die innerhalb seine Domäne. Erhält er eine Anfrage für einen Rechner außerhalb seines Bereiches, leitet er diese an den zuständigen Server weiter. Jeder DNS-Server muß deshalb nur wissen, wie er den jeweils zuständigen Server ermittelt. Verwendet man dieses Konzept für COLIBRI, müssen für eine netzweite Suche die Clients nicht unzählige Rechner kennen, sondern beauftragen statt dessen einen Server, die Anfrage an mehrere andere Server weiterzuleiten. Die erhaltenen Teilresultate müssen dann von jedem Server zu einem Gesamtergebnis zusammengesetzt werden.

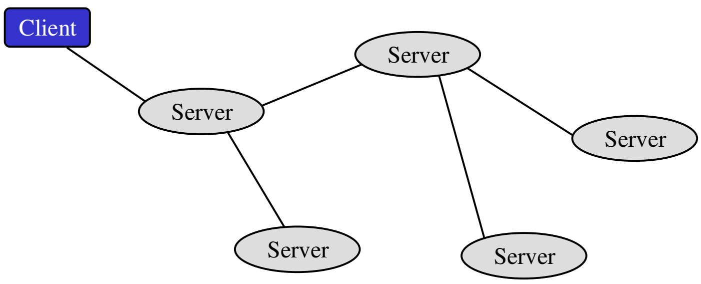

Abbildung 57: gekoppelte Serverarchitektur

Da jeder Server gleichzeitig auch immer Vermittlungstätigkeit übernehmen muß, ist diese Architektur für eine große Anzahl von Servern zu aufwendig. Deshalb sollte es wie beim Vorbild DNS eigene Server für die Vermittlung geben. Diese „Lookup“-Server bestehen eigentlich nur aus einer Liste von anderen bekannten Servern, wodurch sie relativ klein ausfallen sollten. Bei dieser Struktur können sich mehrere Server einen Lookup-Server teilen, wodurch eine Art Gruppenbildung entsteht.

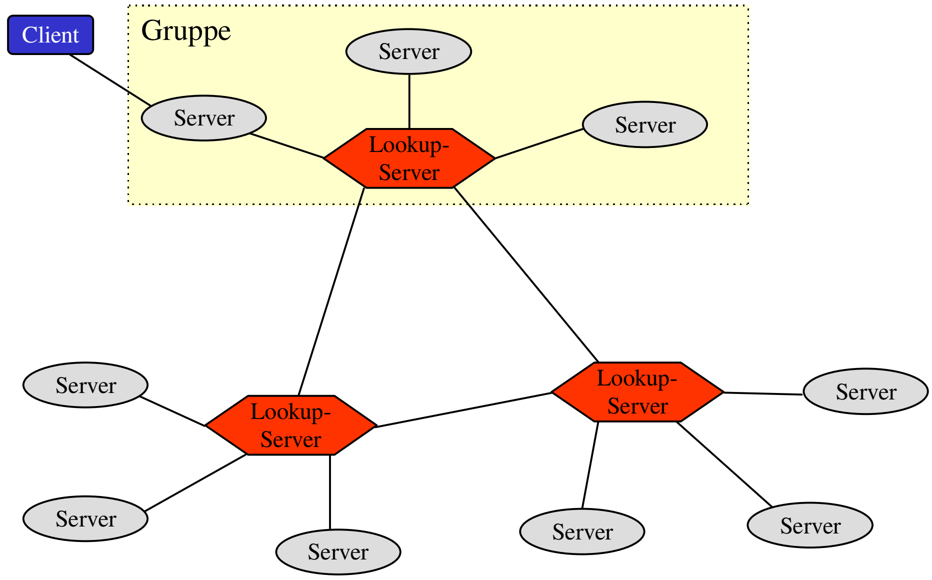

Abbildung 58: Serverarchitektur mit Lookup-Servern

Die gesamte Serverstruktur dient nur zur netzweiten Suche. Sind die einzelnen Server erst einmal bekannt, werden sie vom Client direkt kontaktiert. Dazu besteht in COLIBRI die Möglichkeit, Server nicht nur manuell, sondern auch halbautomatisch aus Suchergebnissen als „Bookmarks" einzutragen. Die Komponenten dieser Server können dann direkt ohne vorherige Installation genutzt werden. Im Falle einer langsamen Netzwerkverbindung können sie jedoch auch einfach in das lokale Dateisystem installiert werden. Inhalte von Servern werden so übertragen, wie sie auch lokal angezeigt werden würden. Dies beinhaltet also auch sämtliche Zusatzinformationen wie Icon, Versionen, Beschreibung, Autor, etc.

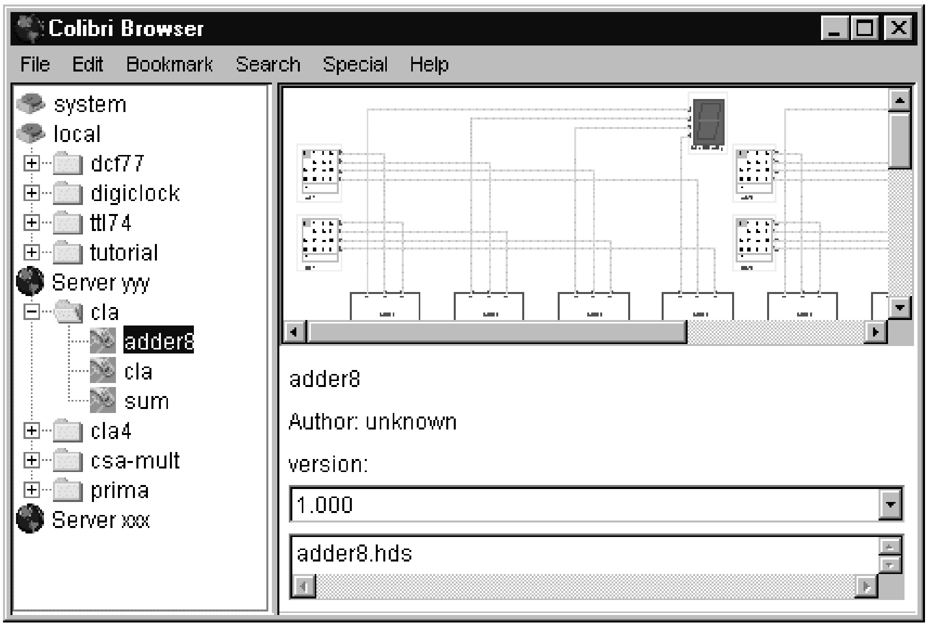

Abbildung 59: Anzeige von Servern in COLIBRI

Für eine spätere Erweiterung wäre eine optionale Benutzer-Anmeldung mit unterschiedlichen Rechten wünschenswert, so daß Komponenten bei entsprechender Berechtigung auch auf einen Server installiert werden können. Diese Möglichkeit würde die gesamte Administration des Servers erheblich vereinfachen, birgt aber natürlich auch immer ein größeres Sicherheitsrisiko. Die genaue Implementation des Servers und die Wahl des Kommunikationsprotokolles wird in Abschnitt 7.4 erläutert.

6.9 Namespaces

Mit der Ausweitung des Internets werden weltweit eindeutige Bezeichner zunehmend wichtiger. Für Java-Klassen wird dies über eine Konvention erreicht, nach der die Klassen- und Package-Namen mit den umgekehrten Domain-Namen der jeweiligen Anbieter beginnen. Da die Vergabe der Domain-Namen behördlich geregelt wird, ist eine zufällige Mehrdeutigkeit nahezu ausgeschlossen. Java-Programme der Firma Sun beginnen so z.B. immer mit dem Package `com.sun.java`. Der Arbeitsbereich TECH am Fachbereich Informatik der Universität Hamburg hingegen müßte seine Programme immer in einem Package `de.uni-hamburg.informatik.tech` herausgeben.

Mit der Einführung der JavaBeans wird die Einhaltung dieser Konvention von entscheidender Bedeutung, da Beans sonst miteinander kollidieren könnten. Die von Java verwendeten Namespaces sind daher auch verbindlich für COLIBRI. Designs als Pseudo-Komponenten ohne Programmcode sind hiervon ausgenommen, da sie keine Package-Struktur besitzen. COLIBRI benutzt deshalb generell die Benutzer-Struktur zur Identifikation und hält die Package-Struktur im Verborgenen. Für den Benutzer hat dies den Nachteil, daß in einem Verzeichnis keine gleichnamigen Komponenten liegen dürfen. Dies ist jedoch eine übliche und nachvollziehbare Einschränkung. Viel gravierender ist dagegen der Nachteil, daß Komponenten nur mit Problemen verschoben oder umbenannt werden können (siehe hierzu auch Kapitel 6.4). Würde man die Java-Packages zur Identifikation verwenden, könnte die Benutzer-Struktur völlig entkoppelt und beliebig verändert werden.

Dieser Ansatz ist jedoch aufgrund der Package-losen Entwürfe schwerlich konsequent durchzuhalten.

6.10 Suche

Eine der Anforderungen an COLIBRI war es, von vornherein Komponenten auch im Internet suchen zu können. Die heute gängige Suche auf Suchmaschinen mit Tausenden von Ergebnissen ist jedoch als ein schlechtes Vorbild zu bewerten. Vielmehr sollte eine qualitativ hochwertige Suche ermöglicht werden, wozu COLIBRI die eigene Serverarchitektur (siehe Kapitel 6.7) verwendet. Ideal wäre sicherlich eine integrierte „Trader“-Funktionalität, die nicht nur passiv die Suchworte oder Kategorien vergleicht, sondern aktiv verwandte Begriffe bzw. Bereiche miteinbezieht.

Ziel eines Tradings ist generell die Vermittlung von Diensten (oder Objekten). Ein solcher Dienst wird dabei nicht anhand seines Namens, sondern anhand seiner Eigenschaften gesucht. Bei einer erfolgreichen Vermittlung erhält der Client die nötigen Bindungsinformationen des Dienstes übergeben, um direkt mit dem Diensterbringer kommunizieren zu können [Griffel 98], [Müller-Jones et al. 95]. Zwar existieren mittlerweile einige CORBA-Implementationen, die einen Trading-Dienst zur Verfügung stellen, diese sind jedoch zumeist kommerziell (siehe Kapitel 5.1.3). Aufgrund dessen und der Komplexität von Tradern verzichten wir im Rahmen dieser Arbeit auf einen Einsatz.

6.10.1 Suchkriterien

Wichtigstes Suchkriterium sind sicherlich immer Stichworte. Um nicht auf die kurzen und oftmals wenig aussagekräftigen Komponentennamen angewiesen zu sein, verwaltet COLIBRI auch Kurzbeschreibungen von Komponenten. Da diese Kurzbeschreibungen in die Stichwortsuche miteinbezogen werden, bietet es sich an, eine Zeile mit Stichworten darin aufzunehmen. Der Mehraufwand der getrennten Stichwortverwaltung lohnt sich nur, wenn Stichwortlisten unbedingt benötigt werden.

6.10.2 Kategorisierung

Neben einer Stichwortsuche wäre auch eine Einteilung der Komponenten in Kategorien möglich. Bei Bedarf können dann gezielt die Inhalte einer bestimmten Kategorie auf anderen Servern angezeigt werden. Die Benutzung einer Stichwortsuche könnte dann dazu genutzt werden, die Ergebnisse noch weiter einschränken zu können.

Auch wenn dieser Ansatz sehr vielversprechend scheint, ist eine mögliche Kategorisierung doch fraglich. Um sinnvoll angewendet werden zu können, reicht es nicht, nur eine grobe Einteilung vorzunehmen. Sie muß statt dessen so ausführlich sein, daß die meisten Komponenten einer spezielleren Unterkategorie zuzuordnen sind. Meist dürfte es jedoch schon schwierig sein, eine Komponente überhaupt einer Kategorie klar zuordnen zu können. Im Zusammenhang mit HADES mag es noch einfach sein, ein simples Gatter einzuordnen. Bei einem komplexen Design, wie z.B. einer DCF-77 Uhr wird dies schon deutlich schwerer. Zweite Schwierigkeit ist die Bildung der Kategorien selbst. Je tiefer man die Kategorisierung durchführt, desto schwieriger wird es. Mag dies für die derzeit auf dem Markt befindlichen Komponenten noch möglich sein, ist es für die Zukunft eher fraglich. Angesichts des Potentials an Möglichkeiten müßte eine Kategorisierung ständig aktualisiert und erweitert werden, was einen erheblichen Verwaltungsaufwand bedeutet. Trotzdem ist damit nicht sichergestellt, daß die Kategorisierung eindeutig ist und sich ein deutlicher Vorteil ergibt. Hier kann der Ansatz des Typmanagements, wie er in [Christiansen et al. 97] und [Griffel 98] vorgestellt wird, Einsatz finden. Im Rahmen diese Arbeit sehen wir jedoch davon ab.

6.10.3 Suchvorgang und Ergebnisse

Erhält ein Server eine Suchanfrage, müssen die darauf zutreffenden Komponenten zusammengestellt werden. COLIBRI durchsucht dazu alle verfügbaren Informationen der vorhandenen Komponenten nach den gegebenen Stichworten. Die gefundenen Komponenten sollen dann in einer möglichst übersichtlichen und intuitiv verständlichen Form von COLIBRI angezeigt werden. Wünschenswert wäre es dabei, gefundene Komponenten direkt aus den Suchergebnissen installieren bzw. benutzen zu können. Die Ergebnisse sollten weiterhin nicht nur kurz, sondern auch dauerhafter integrierbar sein.

COLIBRI integriert die Suchergebnisse deshalb als neuen Hauptast in den Komponenten-Baum, wo sie solange verfügbar bleiben, bis sie explizit vom Benutzer gelöscht werden. Die Generierung der Teilbäume wird bereits von den Servern übernommen. Diese werden dann in serialisierter Form als Suchergebnis zurückgeliefert und brauchen von COLIBRI nur noch in den Hauptbaum integriert werden. Die Ergebnisse verschiedener Server werden dabei noch unter einem gemeinsamen Sucheintrag zusammengefaßt.

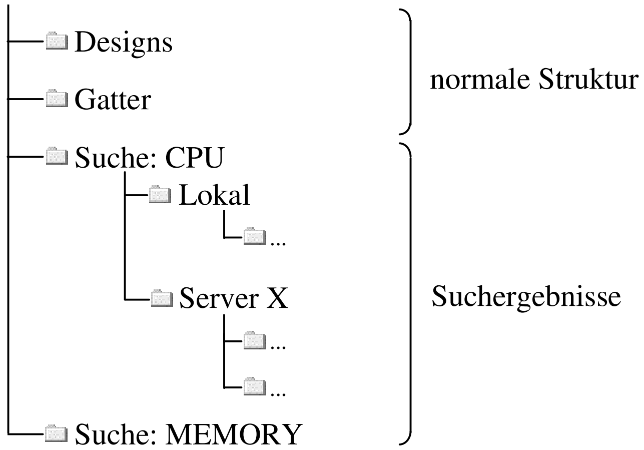

Abbildung 60: Darstellung von Suchergebnissen

6.11 Sicherheitskonzept

Beim Umgang mit Komponenten ist es in jedem Fall wünschenswert, unterschiedliche Zugriffsrechte vergeben zu können. Jede neue Komponente stellt ein potentielles Risiko dar und sollte daher in ihrem Zugriff auf sicherheitsrelevante Ressourcen beschränkt werden. Java bietet dafür seit jeher die Möglichkeit, einen eigenen SecurityManager (siehe Kapitel 2) zu integrieren, der jeglichen Ressourcenzugriff steuert. Um jedoch verschiedenen Komponenten unterschiedliche Rechte vergeben zu können, muß der SecurityManager immer die eigentlich anfragende Komponente ermitteln. Java 1.2 besitzt ein neues Sicherheitskonzept, das diese Möglichkeit bereits fertig zur Verfügung stellt. Dazu können jedem JavaBean eigene Domänen (siehe Abschnitt 2.2) und damit individuelle Rechte zugeordnet werden. Die eigentliche Applikation (HADES & COLIBRI) läuft dabei unter der Systemdomäne und behält volle Zugriffsrechte.

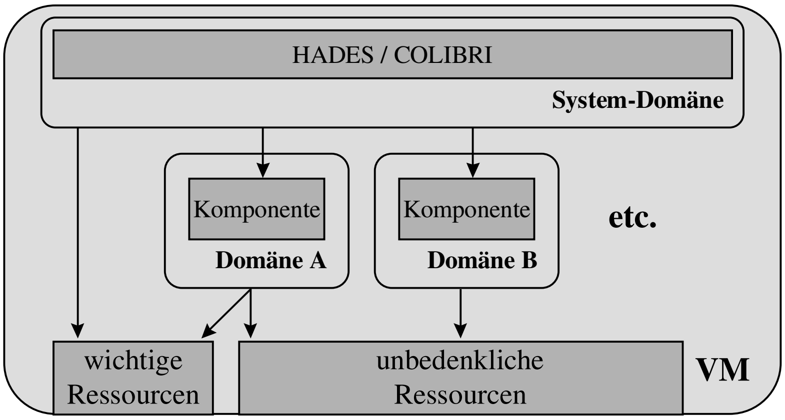

Abbildung 61: Mögliche Domänen in HADES

Die Verwaltung von Domänen und Rechten übernimmt generell das System, die erste Zuordnung jedoch der ClassLoader. Java 1.2 stellt deshalb einen neuen `java.security.SecureClassLoader` zur Verfügung, der als Basis für eigene Anpassungen dient [Gong et al. 97]. Dieser `SecureClassLoader` führt einen zusätzlichen CLASSPATH (Property `java.app.class.path`) ein, der als Unterscheidung zwischen vertrauenswürdigem System- und nicht vertrauenswürdigem Applikationscode dient.

Um diesem Sicherheitskonzept zu folgen, müßte Colibri lediglich den eigenen ClassLoader auf den neuen `SecureClassLoader` aufbauen und die Erzeugung neuer Domänen inklusive der Rechtsvergabe übernehmen. Da HADES-Simulationsobjekte ihre Daten nicht direkt, sondern nur über Colibri laden bzw. speichern sollen und die graphische Repräsentation von HADES übernommen wird, benötigen sie momentan eigentlich keine sicherheitsrelevanten Rechte. Man könnte daher derzeit alle Komponenten in einer einzigen Domäne ablaufen lassen, was den Verwaltungsaufwand noch verringern würde.

Neben den Zugriffsrechten von Komponenten besitzt Colibri aber noch einen weiteren sicherheitsrelevanten Aspekt, nämlich die Serverkommunikation. Insbesondere für spätere Benutzeranmeldungen wäre eine verschlüsselte Kommunikation wünschenswert. Bereits Java 1.1 stellt dafür grundlegende Algorithmen zur Verfügung. Die bereits angekündigte Unterstützung der SSL (*Secure Socket Layer*) Architektur [Gong 98] eignet sich dafür jedoch noch besser, da sie ein etabliertes Protokoll darstellt, das z.B. auch von Proxy-Servern unterstützt wird. Für Benutzeranmeldungen enthält Java 1.2 bereits jetzt eine integrierte Infrastruktur.

Da Java 1.2 erst gegen Ende dieser Arbeit veröffentlicht wurde und das leider auch nur als Betaversion, haben wir von der Integration eines Sicherheitskonzeptes abgesehen. Die Möglichkeit ist jedoch vorhanden und sollte sich wie oben beschrieben relativ leicht nachträglich implementieren lassen. Grundlage sollte aber in jedem Fall das neue Sicherheitskonzept von Java 1.2 sein. Es verbindet vielfältige Möglichkeiten mit einer einfachen Handhabung und wird obendrein von Sun gepflegt, wodurch Sicherheitslücken durch Programmfehler leichter vermieden werden können.

6.12 Kommerzieller Aspekt

Angesichts des zu erwartenden Booms von Komponenten sollte auch in dieser Arbeit der kommerzielle Aspekt zumindest angedacht werden. Relevant sind dabei hauptsächlich dienstorientierte Komponenten, bei denen nicht nur ein einmaliger Kaufpreis, sondern auch laufende benutzungsabhängige Kosten anfallen. Das derartige Komponenten in Zukunft eine wichtige Rolle spielen werden, zeigt z.B. die Entwicklung der Enterprise-JavaBeans [Thomas 97]. Diese spezielle Art von Beans ist Server-basiert und stellt damit einen normalerweise sicherlich kostenpflichtigen Dienst zur Verfügung. Wichtiger Punkt solcher Architekturen sind immer integrierte Transaktionen, die die Grundlage für Abrechnungsverfahren bilden.

Während für allgemeine Komponenten viele Kosten verursachende Szenarien (Benutzung umfangreicher Datenbanken, besonders schneller Rechner, etc.) denkbar sind, wird dies für HADES Simulationsobjekte wesentlich schwieriger, da sie normalerweise keinen Zugriff auf Datenbanken oder ähnliches benötigen. Auch die Verwendung besonders leistungsfähiger Rechner für außergewöhnlich komplexe Komponenten erscheint aufgrund langsamer Netzwerkverbindungen in der Simulation nicht sinnvoll.

Dennoch sind natürlich für die Zukunft auch hier solche Szenarien denkbar und sollten deshalb zumindest soweit berücksichtigt werden, daß sie nachträglich integrierbar sind. In COLIBRI sollte deshalb auch die Verwaltung von Enterprise-JavaBeans berücksichtigt werden. Da Komponenten nicht beim Laden oder Speichern, sondern nur bei ihrer Verwendung Kosten verursachen können, braucht dieser Aspekt in COLIBRI nicht betrachtet zu werden. In diesem Zusammenhang ist eher HADES gefordert, Kontrollmöglichkeiten anzubieten. Eine Vermittlung von Komponenten könnte jedoch auch in COLIBRI Kosten verursachen. An dieser Stelle müßte ein Server dem Benutzer für das Verwenden des Suchdienstes einen bestimmten Betrag in Rechnung stellen. Für die in diesem Zusammenhang benötigten Transaktionen, sollte auf den für Enterprise-JavaBeans bereitgestellten Transaktions-Dienst zurückgegriffen werden.

7 Realisierung und Implementation des COLIBRI

Nachdem in Kapitel 6 verschiedene konzeptionelle Ansätze erläutert wurden, soll in diesem Kapitel auf die konkrete Implementation eingegangen werden. Dazu gehört neben der Darstellung verschiedener Implementationsdetails auch eine Erläuterung der während der Realisierung aufgetretener Schwierigkeiten. Kapitel 7.1 gibt zunächst einen groben Überblick über die vorhandenen Klassen und deren Zusammenspiel. Im Anschluß vertiefen wir in Kapitel 7.2 allgemeine Punkte der Implementation. Kapitel 7.3 und 7.4 gehen dann speziell auf den Browser, bzw. die Serverimplementation ein.

7.1 Klassenhierarchie

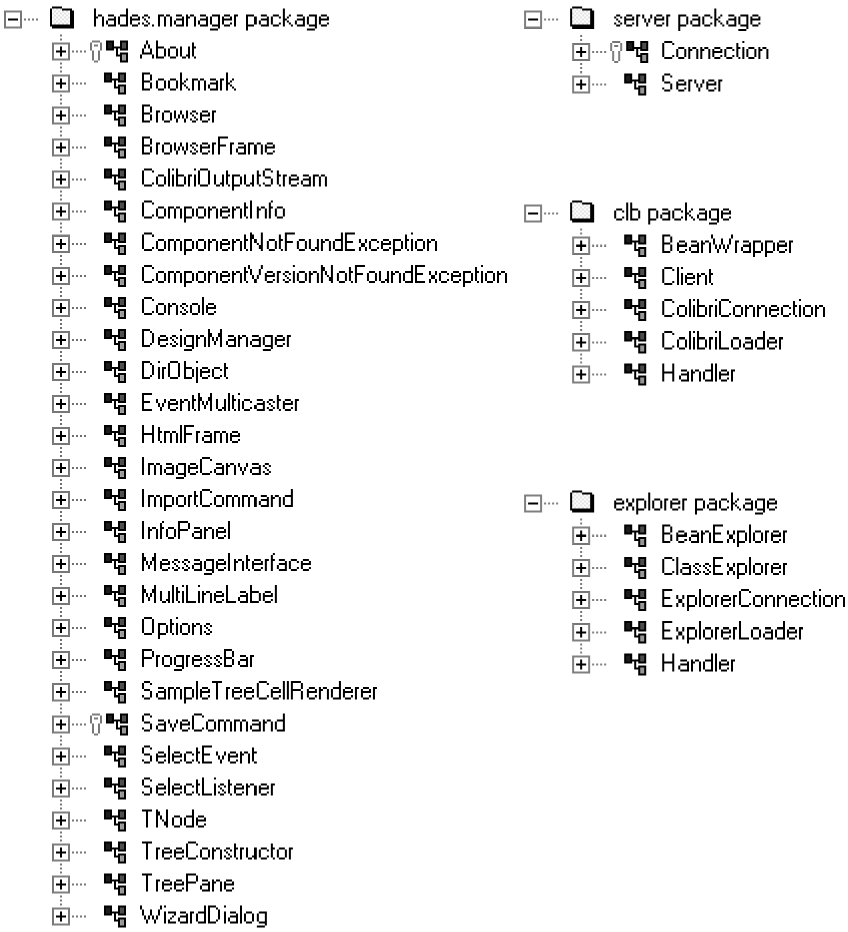

Abbildung 62: COLIBRI Klassenhierarchie

Das gesamte COLIBRI-Package umfaßt mittlerweile über 13000 Zeilen Code in insgesamt 40 Klassen. Im Mittelpunkt der Klassenhierarchie steht die Klasse `Browser`. Diese stellt Methoden zum Laden und Speichern von Komponenten zur Verfügung. Das eigentliche Browserfenster wird durch die Klasse `BrowserFrame` repräsentiert, welches verschiedene `Panel` enthält. Das `TreePane` enthält der `JTree`, der sich aus einzelnen `TNodes` zusammensetzt. Diese `TNodes` enthalten jeweils ein Objekt. Hierbei kann es sich um ein

`Bookmark`, ein `DirObject` oder eine `ComponentInfo` für eine Komponente handeln. In der `ComponentInfo` sind die verfügbaren Informationen zu einer Komponente gespeichert (siehe 7.2.2). Daher greifen mehrere Klassen auf diese zu. Der `TreeConstructor` ist für die Konstruktion der Baumes aus den TNodes zuständig. Je nach Objekt des `TNodes` zeichnet der `SampleTreeCellRenderer` im Baum das entsprechende Icon. Die `Options` speichern die benutzerrelevanten Daten, wie Verzeichnisse, in dem sich die Komponenten befinden, Bookmarks und den Benutzernamen. Beim erstem Programmstart werden die Optionen durch den `WizardDialog` abgefragt. `ImageCanvas` und `InfoPanel` zeigen Informationen über eine Komponente an. Diese erhalten sie aus der `ComponentInfo`. Mit dem `ImportCommand` können bestehende Komponenten importiert werden. Der `ImportCommand` erzeugt dazu die `ComponentInfos`.

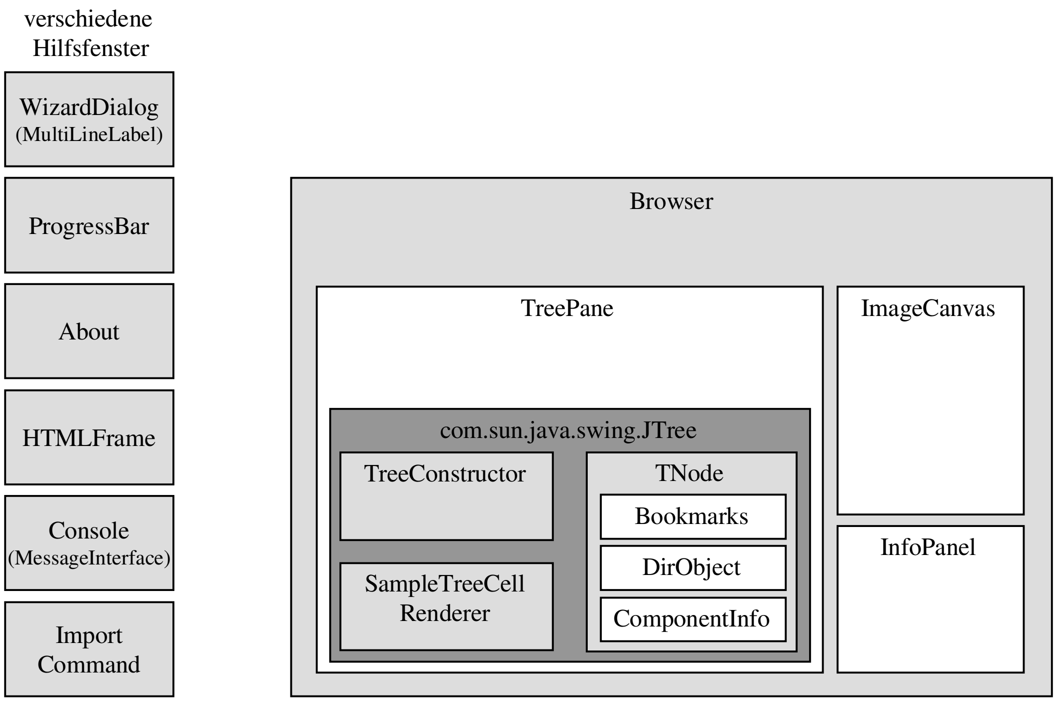

Abbildung 63: GUI-Komponenten des COLIBRI

Das Laden von Komponenten erfolgt durch das `clb package`. Dieses implementiert ein eigenes Protokoll für eine `URL` (siehe unten). Die `ColibriConnection` implementiert dabei die Methode `getContent` der `URL`. Findet sie keine entsprechende Komponente oder Version wirft sie eine `ComponentNotFoundException` bzw. eine `ComponentVersionNotFountException`. Der `ColibriLoader` ist für das eigentliche Laden der Klasse verantwortlich. Dabei wird ein Zwischenschritt üver den BeanWrapper gemacht, damit Komponenten auch über einen Server geladen werden können (siehe 7.1). Der Client ist für den Verbindungsaufbau zu einem Server zuständig. Das Speichern von Komponenten erfolgt durch den `SaveCommand` (siehe 7.1). Als Bindeglied zwischen dem COLIBRI und HADES dient der `DesignManager` und die Hilfsklasse `ColibriOutputStream`. Um über die Auswahl einer Komponente benachrichtigt zu werden, implementiert der `DesignManager` das `SelectListener` Interface. Von dem `EventMulticaster` wird er dann über vom Browser ausgelöste `SelectEvents` informiert (siehe 7.2.4).

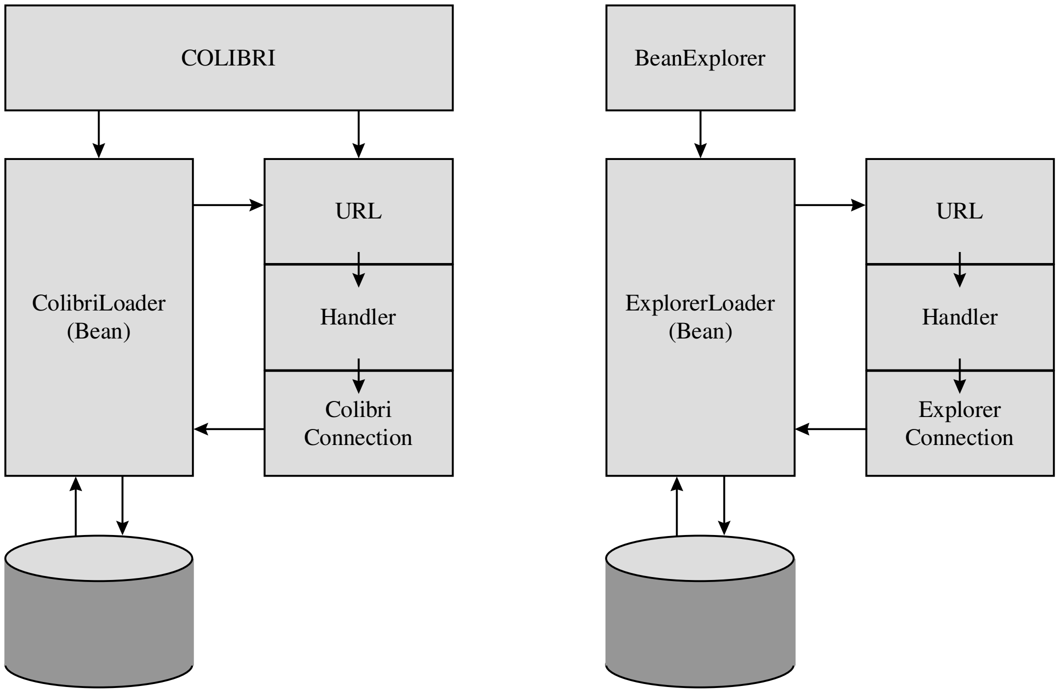

Abbildung 64: Laden von Komponenten über eine URL

Mit dem `explorer package` lassen sich JavaBeans untersuchen. Auch hierzu wird ein eigenes Protokoll für eine `URL` durch die `ExplorerConnection` implementiert. Der `BeanExplorer` ist das eigentliche Fenster, in dem die Informationen über das Bean angezeigt werden (siehe 7.2.5).

Das server package enthält den Server und die Connection. Die Connection implementiert den benötogten Teil der HTTP-Protokolls (siehe 7.3).

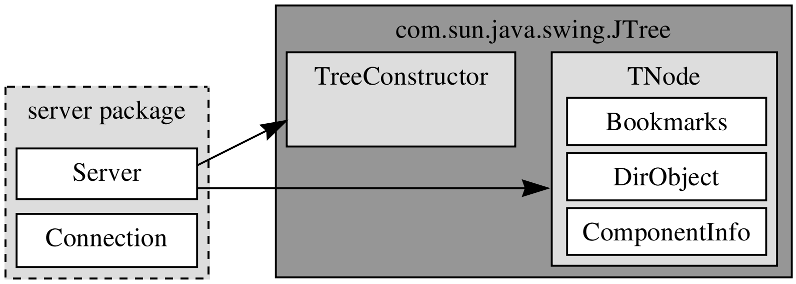

Abbildung 65: Server-Package

7.2 Überblick

7.2.1 ClassLoader-Problematik

Um Komponenten von beliebigen Stellen laden zu können, ist es in jedem Fall sinnvoll einen eigenen ClassLoader zu verwenden (siehe Abschnitt 6.3). Vorrangige Aufgabe eines ClassLoaders ist es, Klassendefinitionen zu lokalisieren, zu laden und der Virtual Machine bekanntzumachen. Seit Java 1.1 dient der ClassLoader zusätzlich auch zum Auffinden beliebiger Ressourcen. Dazu wurden verschiedene `getResource`-Methoden eingeführt, mittels derer Ressourcen direkt über Streams oder indirekt über URLs geladen werden können.

Um jederzeit zugehörige Klassen und Ressourcen nachladen zu können, müssen alle Klassen ihren ClassLoader kennen. Mit dem Aufruf `getClass().getClassLoader()` kann dazu von jedem Objekt der jeweils zuständige ClassLoader ermittelt werden. Der in der Virtual Machine integrierte Loader ist jedoch definitionsgemäß `null`. Das bedeutet, daß vom System geladene Klassen aus Programmsicht keinen ClassLoader besitzen. Um zu vermeiden, daß Klassen mehrfach geladen werden, besitzen ClassLoader intern jeweils einen eigenen Cache.

Im Laufe der Arbeit stellte sich heraus, daß genau hierin die größte Problematik im Umgang mit HADES-Simulationsobjekten besteht: Da der System-ClassLoader `null` ist, gibt es keine Möglichkeit, auf den Cache des System-ClassLoaders zuzugreifen. Greift nun ein Objekt des System-ClassLoaders direkt auf ein Objekt eines anderen ClassLoaders zu, wird die benötigte Klasse im System-Cache gesucht. Da der System-ClassLoader sie dort nicht findet, versucht er sie aus dem CLASSPATH zu laden. Findet er sie dort auch nicht, kommt es zu einer `ClassNotFoundException`. Wird die Klasse gefunden, befinden sich zwei Definitionen im System, die obwohl gleich, vom Interpreter als unterschiedlich angesehen werden. In diesem Fall kommt es deshalb zu einer `ClassCastException`.

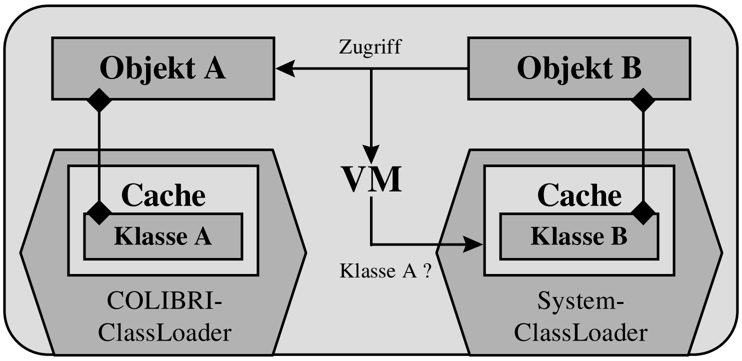

Abbildung 66: Objekt-Zugriff bei zwei ClassLoadern

Bei JavaBeans tritt dieses Problem normalerweise nicht auf, da die Entwicklungswerkzeuge (BeanBox, etc.) die verwendeten Beans nicht direkt, sondern nur über die Reflection-Schnittstelle ansprechen. Zusammengebaute Applikationen werden später komplett vom System-ClassLoader geladen, wodurch auch hier keinerlei Probleme auftreten.

Wie in Abschnitt 4.3 erwähnt, sind HADES-Simulationsobjekte keine 100%igen Beans, da sie alle von der HADES-Klasse `hades.simulator.SimObject` erben müssen. Über diese vom System-ClassLoader geladene Klasse, werden die Simulationsobjekte dann direkt von HADES angesprochen. Problematisch wird dies erst bei Designs. Stellt HADES fest, daß es sich um ein Design handelt, greift es mittels einer Typumwandlung direkt auf diese Klasse zu, weswegen die Klasse `hades.models.Design` in jedem Fall vom System-ClassLoader geladen wird. Der zugehörige ClassLoader ist damit `null`. Damit ein Design aber auf seine Ressource-Dateien zugreifen kann, benötigt es eine Referenz auf den zuständigen COLIBRI-ClassLoader. Automatisch wäre dies nur dann der Fall, wenn jeder COLIBRI-ClassLoader die Klasse `Design` neu definiert. Wie oben geschildert, käme es dadurch jedoch zu einer ClassCastException.

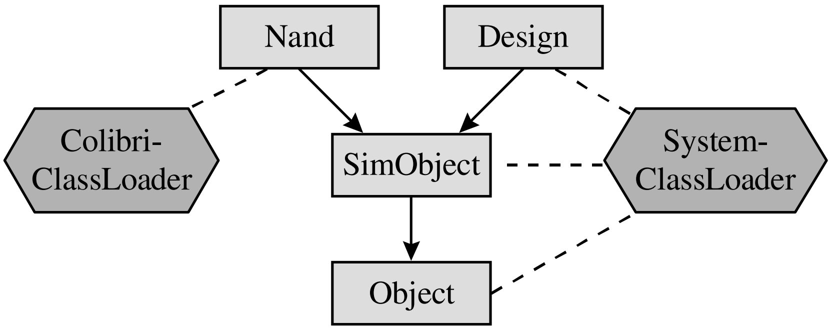

Abbildung 67: Klassenhierarchie bei HADES-Komponenten

Ein möglicher Ausweg wäre es, wenn HADES seine Simulationsobjekte nur noch über die Reflection-Schnittstelle ansprechen würde. Alle COLIBRI-ClassLoader könnten dann ein eigenes `Design` definieren, ohne daß es zu einer ClassCastException kommt. Aus Performanz-Gründen ist diese Lösung für HADES jedoch inakzeptabel. Da diese Problematik bei echten JavaBeans nicht auftritt, ist es besser, statt dessen eine HADES spezifische Lösung zu wählen.

Damit HADES-Komponenten ihre Ressourcen laden können, benötigen sie eine Referenz auf ihren ClassLoader. Da dieser bei Designs zwangsweise `null` ist, bietet es sich an, die Referenz manuell in jeder Instanz zu speichern. Um ein einheitliches Vorgehen zu erreichen, wird die Referenz daher jedoch nicht in der Klasse `Design`, sondern in der allen HADES-Simulationsobjekten gemeinsamen Basisklasse `hades.simulator.SimObject` gespeichert. Da zum Laden von Ressourcen dann immer die „neue“ Referenz auf ihren ClassLoader verwendet werden muß, ist es weiterhin sinnvoll, in der Klasse `SimObject` eigene `getResource()` Methoden zu integrieren, die dies automatisch sicherstellen.

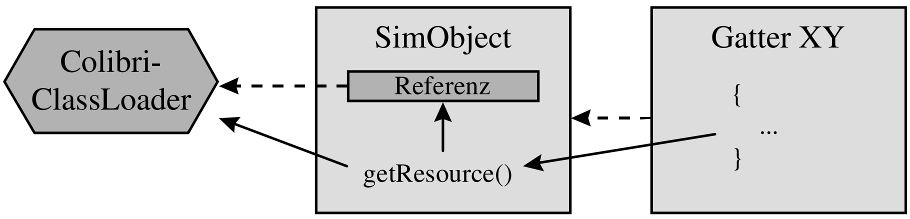

Abbildung 68: Benutzung der ClassLoader Referenz in SimObject

7.2.2 COLIBRI-ClassLoader

Wie im vorigen Abschnitt beschrieben, hat der COLIBRI-ClassLoader die Aufgabe, Klassen und Ressourcen der Komponenten zu laden. Da es kaum möglich ist, aus einer geladenen Komponente ihren Herkunftsort zu ermitteln, verwendet COLIBRI für jede Komponente (nicht Instanzen) einen eigenen ClassLoader. Eine aufwendige Verwaltung der Komponenten und Ursprungsorte entfällt dadurch.

Da COLIBRI zur Verwaltung alle zugehörigen Dateien einer Komponente kennen muß, bietet es sich an, diese Dateien im Voraus zu laden. Die ClassLoader müssen dadurch die benötigten Dateien nicht einzeln laden, sondern können es „en bloc“ am Anfang tun. Dies ist insbesondere bei Server-basierten Komponenten vorteilhaft, die sonst Datei für Datei über oftmals schlechte Netzwerkverbindungen geladen werden müßten.

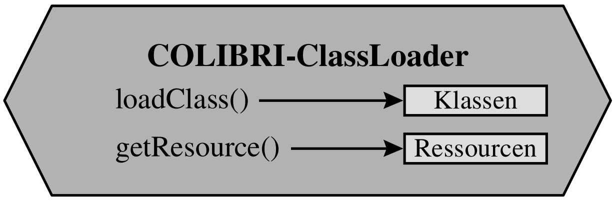

Abbildung 69: COLIBRI-ClassLoader mit allen Dateien

Intern speichert jeder ClassLoader die Dateien zunächst als Byte-Arrays. Klassendateien werden bei Bedarf der Virtual Machine bekanntgemacht („definiert“) und dann durch die neuen „Class“-Objekte ersetzt. Jeder ClassLoader bildet dadurch eine geschlossene Einheit, die bis auf die Klassen der Systembibliothek alles nötige enthalten.

7.2.3 Die Klasse BeanWrapper

Sollen Komponenten von einem Server geladen werden, liegt es durch die gekapselten ClassLoader nahe, bereits auf dem Server einen solchen ClassLoader zu erstellen, und ihn in serialisierter Form über das Netzwerk zu übertragen. Es stellte sich jedoch heraus, daß ein ClassLoader aus Sicherheitsgründen nicht serialisiert bzw. von einer anderen Virtual Machine eingelesen werden darf.

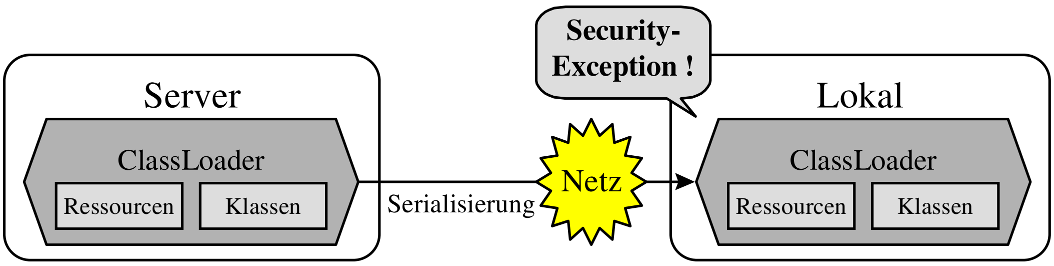

Abbildung 70: Übertragen eines serialisierten ClassLoaders

Aus diesem Grund wurde ein zweistufiger Ansatz gewählt, bei dem die nötigen Daten zunächst in einer Wrapper-Klasse gespeichert werden, die sich serialisieren und übertragen läßt. Danach wird lokal aus dem Wrapper der eigentliche ClassLoader erstellt und die eigentliche Komponente bereitgestellt. Sämtliche Lade-Logik befindet sich dabei in der Klasse `BeanWrapper`. Der Konstruktor des COLIBRI-ClassLoaders bekommt dann nur noch einen BeanWrapper übergeben und übernimmt die darin enthaltenen Daten.

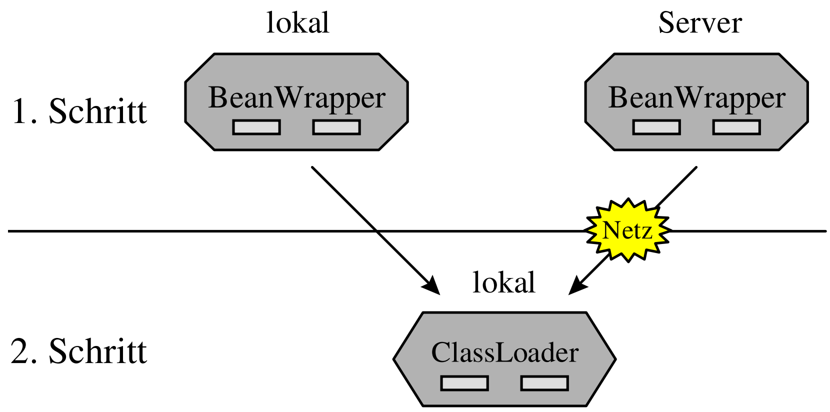

Abbildung 71: BeanWrapper zum Laden/Übertragen von Komponenten

7.2.4 URL

Da Komponenten auch über das Internet bereitgestellt werden können, muß eine eindeutige Referenzierung sichergestellt werden. Hierzu bietet sich das im Internet verwendete Konzept der *Uniform Resource Locators* (URL) besonders an. URLs sind ein standardisiertes Format zur Adressierung von Ressourcen im Internet. Eine URL besteht aus mehreren Angaben: einem Protokoll, einem Server, ggf. ein Port (Protokoll-Standardport kann entfallen) und einer Pfadangabe zu einer Datei oder einem Verzeichnis:

Protokoll://Server:Port/Verzeichnis/Datei

So liefert z.B. die URL *http://tech-www.informatik.uni-hamburg.de/Students/2kroene/ index.html* die HTML-Datei *index.html* im Verzeichnis */Students/2kroene* auf dem Web-Server *tech-www.informatik.uni-hamburg.de*. Die URL-Schreibweise wird auch für andere Internet-Dienste wie Gopher (*gopher://*), FTP (*ftp://*) oder Newsgroups (*news://*) verwendet.

Die in der Java Systembibliothek enthaltene Klasse `java.net.URL` enthält eine automatische Unterstützung der wichtigsten Internet-Protokolle. Um für andere Protokolle auf die bereits bestehende Infrastruktur zurückgreifen zu können, bietet Java die Möglichkeit, eigene Protokolle zu integrieren. Dies ist jedoch nicht nur optional, sondern in Verbindung mit einem eigenen ClassLoader sogar Pflicht. Ein ClassLoader bietet grundsätzlich zwei verschiedene Arten Ressourcen zu laden, zum einen direkt über einen InputStream und zum anderen über URLs. Fordert man bei einem ClassLoader eine Ressource-URL an, muß dieser eine solche dynamisch generieren, anhand derer später der jeweilige ClassLoader wieder ermittelt werden kann (üblicherweise mittels eines Hashcodes). Dieses ist ohne die Einbindung eines eigenen Protokolls kaum möglich.

Um neue Protokolle zu integrieren, verwendet die Klasse `URL` ein System-Property `java.protocol.handler.pkgs`, das ähnlich dem CLASSPATH die Basispfade der zusätzlichen Protokoll-Handler enthält. Trifft die Klasse URL auf ein ihr normalerweise unbekanntes Protokoll, hängt sie den Namen des Protokolls an die in dem Property spezifizierten Pfade und sucht dort eine „`Handler`“-Klasse. Dies geschieht für alle im CLASSPATH angegebenen Klassenpfade.

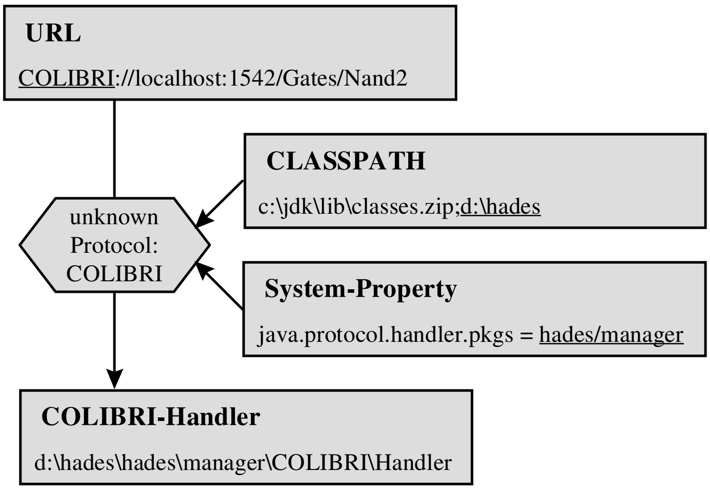

Abbildung 72: Suche nach neuem Protokoll-Handler

Zu Programmbeginn muß der Pfad, in dem sich der eigene Handler befindet, in der Umgebungsvariablen registriert. Damit dies nur einmal geschieht, wird es am Einfachsten in einem `static`-Block ausgeführt, der automatisch beim Laden der Klasse durchlaufen wird:

```
static
{
   // *** Add new protocol type to URL ***
   String protocolPathProp = "java.protocol.handler.pkgs";
   Properties prop = System.getProperties();

   // add path to protocol property
   if (prop.getProperty(protocolPathProp) == null)
   {
      prop.put(protocolPathProp, "HADES.manager");
   }
   else if (prop.getProperty(protocolPathProp).indexOf
                ("HADES.manager") <0 )
   {
      prop.put(protocolPathProp,
               prop.getProperty(protocolPathProp)
             +"|HADES.manager");
   }
   System.setProperties(prop);
}
```

Der Protokoll-Handler selbst übergibt der Klasse `URL` lediglich ein spezifisches `Connection`-Objekt, welches die eigentliche Arbeit leistet. Hat man ein entsprechendes `Connection`-Objekt und den passenden Protokoll-Handler programmiert und registriert, können Komponenten und zugehörige Ressourcen direkt mittels einer normalen URL referenziert und geladen werden. Die von COLIBRI verwendete URL hat folgenden Aufbau:

clb://*<server>*/*<path>*/*<component>*[?version=*<version>*,file=*<file>*]

Die einzelnen Bestandteile haben dabei folgende Bedeutungen:

<server> Der Server, auf dem die Daten liegen. Komponenten, die im lokalen Dateisystem liegen, enthalten als Server „local“ (nicht „localhost“, etc.)

<path> Der Pfad, unter dem sich die Komponente im COLIBRI befindet. Der Pfad ist jedoch nie absolut, sondern relativ im COLIBRI System- oder Benutzerverzeichnis:

URL	clb://local/gatter/simple/Nand
Benutzer-Verz.	e:\hades\modelle
absoluter Pfad	e:\hades\modelle\gatter\simple\Nand

Bei Servern wird ebenfalls nur der relative Pfad angegeben. Der absolute Pfad bleibt hier verborgen und ist nur dem Server selbst bekannt.

<component> Name der referenzierten Komponente

<version> Version der referenzierten Komponente (Optional)

<file> Wenn nicht die Komponente selbst, sondern nur ein dazugehöriges Resource-File (z.B. „sym“ - Datei) referenziert werden soll.

Beispiele für mögliche URLs wären:

clb://local/modelle/gatter/simple/Nand
clb://hades.motorola.com/processors/m680000
clb://hades.tech.informatik.uni-hamburg.de/modelle/gatter/Nand?version=1000
clb://localhost/designs/dcf_clock?file=dcf_clock.sym

7.2.5 Lokale Verzeichnisse

Anfangs bestand die Idee, ähnlich wie bei Servern beliebige lokale Verzeichnisse durch „Bookmarks“ hinzufügen zu können. Dazu hätten jedoch die Namen dieser Bookmarks in die URLs (quasi als Server) aufgenommen werden müssen. Verwendet dann ein Design eine solche Referenz, wäre es auf den entsprechenden Bookmark festgelegt. Dieses ist jedoch sehr unflexibel und läßt ein Gewirr von Abhängigkeiten entstehen.

Aus diesem Grund verwendet COLIBRI eigentlich nur ein lokales Basisverzeichnis für Komponenten. Insbesondere auf Unix-Systemen mit einer zentralen Installation ist es jedoch wünschenswert, zumindest zwischen System- und Benutzerdaten unterscheiden zu können. COLIBRI bietet deshalb die Möglichkeit diese zwei Basisverzeichnisse getrennt angeben zu können.

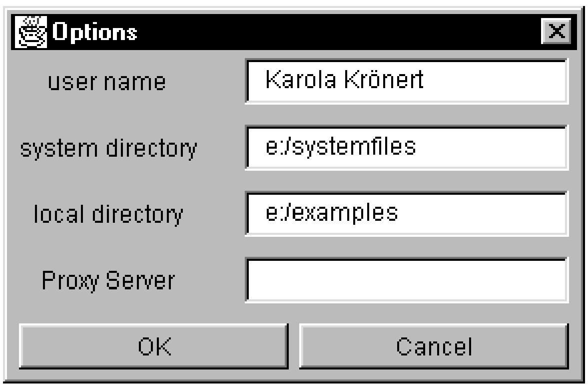

Abbildung 73: Options-Dialog mit getrennten Verzeichnissen

Obwohl intern die beiden Verzeichnisse getrennt verwaltet und im Baum auch separat dargestellt werden, besitzen die Referenzen (URLs) auf Komponenten keinen Verweis auf das jeweilige Verzeichnis. Soll eine Komponente geladen werden, sucht COLIBRI zuerst im System- und danach im (lokalen) Benutzerverzeichnis. Auf diese Weise werden die oben erwähnten Abhängigkeiten vermieden, und Designs lassen sich leichter Verschieben bzw. auf anderen Rechnern installieren.

7.3 Der Browser

Der COLIBRI-Browser wird als Singleton Klasse realisiert [Gamma et al. 94]. Hierdurch wird sichergestellt, daß jeweils nur eine Instanz dieser Klasse existiert. Hierzu wird der Konstruktor für den Browser als `private` deklariert. Eine Methode `getBrowser` wird hinzugefügt, die eine Referenz auf die einzige Instanz zurückliefert. Bei ersten Aufruf dieser Methode ruft diese den Konstruktor für den Browser auf:

```
private static Browser _browser = null;

public static Browser getBrowser()
{
   if (_browser == null)
   {
      _browser = new Browser();
   }
   return _browser;
}

/**
* As Browser is a Singleton class, the constructor is
* private.
* Use getBrowser()' to reference the global
* Browser.
*/
private Browser ()
{
   ...
}
```

7.3.1 Verwendung der Java Foundation Classes

Bei der Implementation des Browsers haben wir für die Benutzeroberfläche die Java Foundation Classes eingesetzt. Da das Browserfenster aus mehreren Teilen besteht, ist eine variable Aufteilung wünschenswert. Swing stellt hierzu das `JSplitPane` zur Verfügung. Diesem werden zwei Panel übergeben, deren Trennlinie dann verschiebbar ist. Der zweite Parameter gibt an, daß das Fenster erst neu gezeichnet werden soll, wenn der Verschiebevorgang beendet ist. Es ist zwar auch ein kontinuierliches Neuzeichen möglich, dies führt jedoch zu einem starken Flackern.

```
splitPane = new JSplitPane
(JSplitPane.HORIZONTAL_SPLIT, false, treePane, rightPanel);
splitPane.setDividerLocation(220);
splitPane.setDoubleBuffered(true);
```

Durch Double Buffering erreichen wir ein Reduzieren des Flackerns. Dabei erfolgen die Zeichenoperationen zunächst in einen nicht sichtbaren Bereich, der dann komplett in das sichtbare Fenster kopiert wird. Dadurch verzögert sich das Zeichnen zwar leicht, aber das Bild wird deutlich ruhiger.

Wie in Kapitel 2 dargestellt verwendet Swing eine modifiziertes Model-View-Controller-Architektur. Im Colibri kommt diese in der Baumdarstellung der Komponenten zum Ausdruck.

```
tree = new JTree();
// do not show root
tree.setRootVisible(false);
// set TreeCellRenderer (used to draw nodes)
tree.setCellRenderer(new SampleTreeCellRenderer());
// set TreeSelectionModel (how many Components
// can be selected at one time)
DefaultTreeSelectionModel dtsm = new DefaultTreeSelectionModel();
dtsm.setSelectionMode(
   DefaultTreeSelectionModel.SINGLE_TREE_SELECTION);
tree.setSelectionModel(dtsm);
tree.setModel(new DefaultTreeModel(new TNode(null, null)));
this.getViewport().add(tree);
```

Hier wird das TreeModel übergeben

Eine Instanz der Klasse `TreeModel` repräsentiert die Daten des Modells. Dabei übergeben wir zunächst einen leeren Knoten, den wir später durch den gesamten Baum ersetzen. Dadurch kann das Laden des Baumes in einem getrennten Thread ablaufen. Die Klasse `JTree` ist für die korrekte Bildschirmrepräsentation des Modells verantwortlich. Dabei verwenden wir zwei verschiedene Views auf das Modell: den Tree View im Hauptfenster und einen weiteren im Save-Dialog, in dem jedoch die Komponenten nicht editiert werden können. Der SampleTreeCellRenderer ist für die Darstellung des Baumes verantwortlich.

Swing stellt einen einfachen Mechanismus für Benutzerdialoge zur Verfügung. Dazu dient die Klasse `JOptionPane`, die unter anderem folgende Methoden zur Verfügung stellt:

- `showConfirmDialog` stellt eine Frage zur Zustimmung (Ja, Nein, Abbruch)
- `showInputDialog` erwartet eine Benutzereingabe
- `showMessageDialog` Benutzerinformation
- `showOptionDialog` eine Vereinigung der oberen drei

Der Aufruf für den Dialog zum Suchen von Komponenten in der Klasse `BrowserFrame` sieht z.B. folgendermaßen aus:

```
String words = JOptionPane.showInputDialog(
   this,
   "Please enter your keywords for the search:",
   "Search Component",
   JOptionPane.QUESTION_MESSAGE);
if (words!=null)
Vector keys = new Vector();
StringTokenizer st = new StringTokenizer(words, " ");
while (st.hasMoreElements()) {keys.addElement(st.nextToken());}
```

Es wird die Methode `showInputDialog` aufgerufen, da an dieser Stelle von Benutzer die Eingabe von Suchstrings verlangt wird. Zunächst wird der Frame als Hauptfenster angegeben, damit dieser gesperrt bleibt, solange der Benutzerdialog geöffnet ist. Der nachfolgende String wird als Text im Dialog angezeigt. Als drittes wird der Fenstertitel angezeigt. Hier liegt allerdings noch ein Fehler in Swing, da dieser nicht konsistent angezeigt wird.

Der letzte Parameter bestimmt das Aussehen des Dialogs, z.B. das verwendete Icon. Dies ist abhängig vom jeweiligen „Look and Feel“. Unter Windows´95 hat die „Question Message“ beispielsweise das folgende Aussehen:

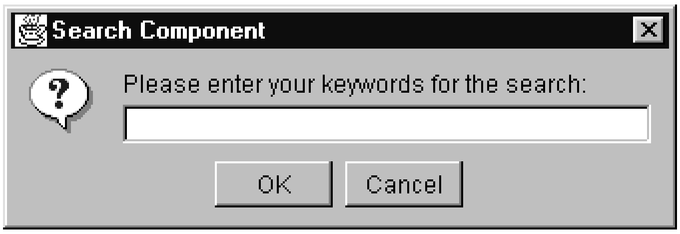

Abbildung 74: Such-Dialog

Die Methode `showInputDialog` liefert den vom Benutzer eingegebenen String zurück, oder `null`, wenn Cancel gedrückt wurde. In diesem Beispiel ermitteln wir mit Hilfe des `StringTokenizers` die einzelnen Suchwörter. Dieser zerteilt einen String in Substrings, wobei ein Zeichen als Trennzeichen gilt.

Die Rückfrage beim Löschen von Komponenten geschieht durch das folgende Codebeispiel:

```
Object[] options = {"Delete", "Cancel"};
int erg = JOptionPane.showOptionDialog(
   this,
   "Are you sure, you want to delete the
        component and all its files ?",
   "Delete Component",
   JOptionPane.DEFAULT_OPTION,
   JOptionPane.WARNING_MESSAGE,
   null,
   options,
   options[0]);
if (erg == 0) deleteNode(selected);
```

Die ersten drei Parameter ergeben sich wie oben. Der vierte Parameter gibt an, welche Buttons im Dialog angezeigt werden. Der fünfte Parameter beschreibt wieder das Aussehen. Hier handelt es sich um eine „Warning Message“, die unter Windows z.B. mit einem Ausrufungszeichen versehen wird. Anstelle von null könnten man als sechsten Parameter auch ein Icon angeben, welches dann alternativ im Fenster angezeigt wird. Die beiden letzten Parameter beschreiben die Buttons noch näher. Wir geben hier ein Array von Strings an. Zu jedem String wird ein Button hinzugefügt. Es ist jedoch auch möglich, andere Objekte, z.B. Icons zu übergeben, die dann als Button angezeigt werden. Der letzte Parameter gibt an, welcher Button ausgewählt ist. Als Rückgabewert erhält man einen Integer, der angibt, welcher Button gedrückt wurde.

Abbildung 75: Rückfrage beim Löschen von Komponenten

Die im COLIBRI verwendete Klasse `ProgressBar` baut nicht auf die entsprechende Klasse aus dem Swing-Package auf, da dann zunächst die Swing-Klassen geladen werden müssen, bevor es überhaupt zu einer Fortschrittsanzeige kommt. Da aber das Laden der Swing-Klassen einen großen Teil der Startzeit des COLIBRI beansprucht, wäre diese Anzeige dann nicht mehr sehr sinnvoll.

Abbildung 76: ProgressBar

Die Klassen aus dem Swing-Package unterlagen während der gesamten Entwicklung ständig Neuerungen. Dadurch wurde die Entwicklung der COLIBRI immer wieder unterbrochen, um Anpassungen an die jeweils neue Version vorzunehmen. Leider läßt auch die Performanz von Swing noch zu wünschen übrig, insbesondere beim Laden der Klassen zum Programmstart. Dies wird sich jedoch ändern, wenn die JFC ab Version 1.2 zu dem JDK gehören, da das Laden dann entfällt. Auch sonst hat sich im Bereich der Geschwindigkeit seit der ersten Version von Swing schon einiges getan.

7.3.2 CLB-Datei

Um Komponenten mit verschiedensten Ressourcen flexibel verwalten zu können, ist es wichtig zu wissen, was zu welcher Komponente gehört. Derartige Informationen werden von COLIBRI in sogenannten CLB-Dateien gespeichert. Dort wird explizit aufgeführt, welche Dateien zu einer Komponente gehören, welche Klasse die Startklasse ist, womit diese gegebenenfalls initialisiert wird und welche Icons und Images verwendet werden sollen.

Die Information in einer CLB-Datei wird im ASCII-Format gespeichert. Sie besteht aus mehreren Blöcken, die jeweils durch ein Schlüsselwort in eckigen Klammern eingeleitet werden. Dabei sind einige Blöcke zwingend, andere optional. Der Aufbau einer CLB-Datei ist offen für Erweiterungen. Momentan unterstütze Blöcke sind folgende:

[files x]	zwingend	die nächsten x Zeilen enthalten alle direkt zur Komponente gehörigen Dateien
[type]	zwingend	Typ der Komponente (beans / AxctiveX / OpenDoc...)
[references x]	zwingend	die nächsten x Zeilen enthalten die URLs der direkten Sub-Komponenten
[archive]	optional	Name des Archives, das alle zur Komponente gehörigen Dateien enthält
[start]	optional	z.B. Name der Startklasse
[init]	optional	String, der der Initialisierungs-Methode übergeben wird (kann z.B. eine ini-Datei sein: .hds / .fsm / .kvd)
[icon]	optional	Name des Icons
[image]	optional	Name des Images
[description]	optional	Beschreibung der Komponente (auf eine Zeile kodiert)

[html]	optional	HTML-Seite zur Beschreibung der Komponente

Tabelle 2: Aufbau einer CLB-Datei

Ein Beispiel für ein einfaches Gatter könnte folgendermaßen aussehen:

```
[files 4]
hades/models/gates/And2.class
hades/models/gates/And2.sym
hades/models/gates/And2.gif
hades/models/gates/And2.html
[versions 1]
1000
[references 0]
[type]
beans
[start]
hades.models.gates.And2
[icon]
simobj.gif
[image]
hades/models/gates/And2.gif
[description]
AND Gate with 2 Inputs
[html]
hades/models/gates/And2.html
```

Hierbei wird als Startklasse jeweils direkt das Simulationsobjekt übergeben. Bei einem Design wird als Startklasse die Klasse `hades.models.Design` angegeben und die eigentliche Design-Datei als Initialisierungsparameter:

```
[files 3]
sum.hds
sum.sym
sum.gif
[versions 1]
1000
[references 0]
[type]
beans
[start]
hades.models.Design
[init]
sum.hds
[icon]
hds.gif
[image]
sum.gif
[description]
sum.hds (automatisch eingelesen)
```

Ähnlich sieht es bei einem endlichen Automaten (FSM: Finite State Machine) aus. Dies ist gleichzeitig ein Beispiel für eine Komponente mit mehreren verfügbaren Versionen.

```
[files 1]
ampel.fsm
[references 0]
[versions 3]
1000
1001
2000
[type]
beans
[start]
hades.models.fsm.FsmWrapper
[init]
ampel.fsm
[icon]
```

```
fsm.gif
[description]
FSM
```

Eine Entsprechung findet die CLB-Datei in der Klasse `ComponentInfo`, die den Inhalt einer CLB-Datei einliest bzw. abspeichert. Sie enthält die entsprechenden `get`- und `set`-Methoden für die einzelnen Blöcke. Ferner enthält sie Methoden, um die Icons und Images zu den Komponenten einzulesen. Die Methode `search` bekommt einen Vektor übergeben, der Suchstrings enthält. Name, Autor und Beschreibung der Komponente werden nun daraufhin untersucht, ob sie diese String enthalten. Zur Zeit ist nur eine Oder-Suche möglich. Denkbar ist aber auch eine Suche mit regulären Ausdrücken.

7.3.2.1 Icons für Komponenten im Tree:

In der CLB-Datei wird das Icon angegeben, mit dem die Komponente im Baum erscheint. Der COLIBRI stellt hierzu einige Standardicons zur Verfügung. Es ist jedoch auch möglich eigene Icons zu erstellen und der Komponente zuzuordnen. Dabei kann dies entweder direkt bei der Komponente liegen, oder sich im Verzeichnis `hades/manager/icons` befinden. Wird ein nicht vorhandenes Icon oder kein Icon angegeben, so wird das Default-Icon ? angezeigt. Wird ein Tree von einem Server angefordert, sendet dieser alle Icons mit, so daß auch dort zusätzlich angegebene Icons angezeigt werden können.

simobj.gif	Simulationsobjekte
hds.gif	Designs
fsm.gif	endliche Automaten
kvd.gif	KV-Diagramme
ic.gif	IC-Baustein
bean.gif	Beans
activex.gif	ActiveX Komponenten
opendoc.gif	OpenDoc Komponenten
default.gif	Default-Icon

7.3.3 Änderungen an der HADES-Datei

In einem HADES-Design enthielten Verweise auf Subkomponenten zunächst den Klassennamen und gegebenenfalls einen Verweis auf Ressourcen relativ zum CLASSPATH. Dies war möglich, da sich alle Komponenten im HADES-Package befanden. Mit Einführung der COLIBRI zur Verwaltung der Komponenten können diese Verweise so jedoch nicht funktionieren. Daher muß statt dessen hier die URL der entsprechenden Komponente eingetragen werden. Dabei kann es sich um eine relative URL handeln, wenn die Komponente auf dem selben Server liegt, oder eine vollständige URL, wenn die Komponente auf einem anderen Server liegt.

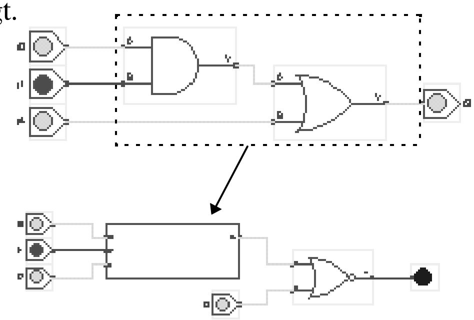

Abbildung 77: HADES-Design und -Subdesign

Beispiel für ein altes HADES-Design mit Klassennamen:

```
# hades.models.Design file
#
[name] unnamed
[components]
hades.models.io.LED i5 28200 12000 @N 1001 3
hades.models.gates.Nor2 i4 22800 10800 @N 1001 5.0E-9
hades.models.io.Ipin i3 20400 13200 @N 1001 U
hades.models.io.Ipin i2 12000 12000 @N 1001 U
hades.models.io.Ipin i1 12000 10800 @N 1001 U
hades.models.io.Ipin i0 12000 9600 @N 1001 U
hades.models.Design unnamed 14400 9600 @N 1001
    /hades/examples/example/sub.hds
[end components]
[signals]
hades.signals.SignalStdLogic1164 n5 2 i4 Y i5 A 1 2 26400 12000
28200 12000 0
hades.signals.SignalStdLogic1164 n4 2 i3 Y i4 B 3 2 20400 13200
21600 13200 2 21600 13200 21600 12600 2 21600 12600 22800 12600 0
hades.signals.SignalStdLogic1164 n3 2 unnamed i3 i4 A 3 2 20400
10200 21600 10200 2 21600 10200 21600 11400 2 21600 11400 22800
11400 0
hades.signals.SignalStdLogic1164 n2 2 i2 Y unnamed i4 3 2 12000
12000 13800 12000 2 13800 12000 13800 11400 2 13800 11400 14400
11400 0
hades.signals.SignalStdLogic1164 n1 2 i1 Y unnamed i1 1 2 12000
10800 14400 10800 0
hades.signals.SignalStdLogic1164 n0 2 i0 Y unnamed i0 3 2 12000 9600
13800 9600 2 13800 9600 13800 10200 2 13800 10200 14400 10200 0
[end signals]
[end]
```

Simulations-Objekt

Subdesign mit Angabe der Ressource

Das selbe Design mit URLs als Verweisen auf die Subkomponenten sieht jetzt so aus:

```
# hades.models.Design file
#
[name] unnamed
[components]
clb://local/io/LED?version=1000 i5 28200 12000 @N 1001 3
clb://local/gates/Nor2?version=1000 i4 22800 10800 @N 1001 5.0E-9
clb://local/io/Ipin?version=1000 i3 20400 13200 @N 1001 U
clb://local/io/Ipin?version=1000 i2 12000 12000 @N 1001 U
clb://local/io/Ipin?version=1000 i1 12000 10800 @N 1001 U
clb://local/io/Ipin?version=1000 i0 12000 9600 @N 1001 U
clb://local/example/sub?version=1000 unnamed 14400 9600 @N 1001
    /hades/examples/example/sub.hds
[end components]
[signals]
hades.signals.SignalStdLogic1164 n5 2 i4 Y i5 A 1 2 26400 12000
28200 12000 0
hades.signals.SignalStdLogic1164 n4 2 i3 Y i4 B 3 2 20400 13200
21600 13200 2 21600 13200 21600 12600 2 21600 12600 22800 12600 0
hades.signals.SignalStdLogic1164 n3 2 unnamed i3 i4 A 3 2 20400
10200 21600 10200 2 21600 10200 21600 11400 2 21600 11400 22800
11400 0
hades.signals.SignalStdLogic1164 n2 2 i2 Y unnamed i4 3 2 12000
12000 13800 12000 2 13800 12000 13800 11400 2 13800 11400 14400
11400 0
hades.signals.SignalStdLogic1164 n1 2 i1 Y unnamed i1 1 2 12000
10800 14400 10800 0
hades.signals.SignalStdLogic1164 n0 2 i0 Y unnamed i0 3 2 12000 9600
13800 9600 2 13800 9600 13800 10200 2 13800 10200 14400 10200 0
[end signals]
[end]
```

Simulations-Objekt

Subdesign

Um eine Komponente vom alten in das neue Formate zu konvertieren, muß diese einmal geöffnet und dann gespeichert werden. Wir werden hierzu noch einen automatischen Mechanismus zur Verfügung stellen.

Bei Verweisen auf Subkomponenten sind im Prinzip zwei Verweise nötig, der eine auf der Startklasse und der andere auf die Initialisierungsdatei. Bei Designs liegt die Startklasse im HADES-Package und auch bei FSMs und KVDs kann man annehmen, daß sich diese dort befinden. Daher könnte der Verweis direkt auf die Initialisierungsdatei zeigen. Ein Problem ergibt sich aber, wenn fremde Bibliotheken eigenen Komponenten entwerfen, die dann nicht mehr im CLASSPATH liegen.

Die Subkomponente selbst kennt zwar ihre zugehörige Klasse und könnte diese rekursiv selbst laden und sich dann selbst initialisieren. Dies ist jedoch in HADES nicht möglich, da zum Initialisieren noch bestimmte Informationen benötigt werden, die an diese Stelle nicht zur Verfügung stehen. So wird vor der Initialisierung festgelegt, ob es sich um ein Toplevel-Design handelt oder nicht, da bei Subdesigns die graphische Repräsentation aus Performanzgründen nicht mitgeladen wird. Nun könnte man diese Informationen der Komponente mit übergeben. Dies würde jedoch im COLIBRI zu eine Abhängigkeit von HADES führen. Eine Eigenständigkeit von Komponenten ist daher in HADES nicht gegeben.

7.3.4 Verbindung HADES und COLIBRI:

Ziel der Diplomarbeit ist es, ein universelles Werkzeug zur Verwaltung von Komponenten zu schaffen. Der Einsatz in HADES erfordert jedoch einige spezielle Anpassungen, unter anderem weil es sich wie in Kapitel 4 erläutert nicht um 100-prozentige Beans handelt. Um jedoch den COLIBRI unabhängig von HADES implementieren zu können, wurde die Klasse DesignManager eingeführt. Der DesignManager dient als Adapter [Gamma et al. 94] zwischen dem COLIBRI und HADES. Dabei konvertiert er das Interface, das von COLIBRI zur Verfügung gestellt wird, in das Interface, das von HADES erwartet wird. Da ihm die HADES Klassen bekannt sind, kann er Anpassungen, z.B. Initialisierung von Komponenten vornehmen.

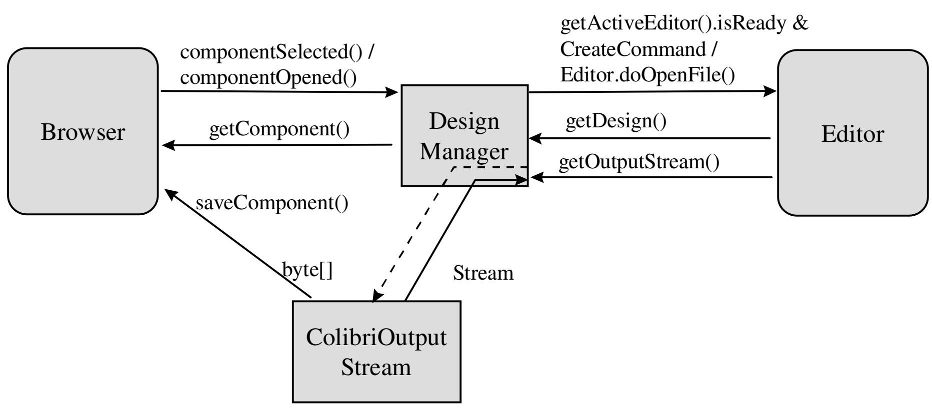

Abbildung 78: DesignManager als Adapter zwischen COLIBRI und HADES

HADES ruft Methoden im DesignManager auf, die dieser dann modifiziert an den COLIBRI weiterleitet. Ein Spezialfall tritt dann auf, wenn Komponenten gespeichert werden sollen. HADES erwartet einen OutputStream, um in diesen dann die zu speichernde Komponente schreiben zu können. Der COLIBRI hingegen erwartet ein fertiges Byte-Array mit den zu speichernden Daten. Dies geschieht im Hinblick auf ein Speichern von Komponenten auf Servern. Hier kann die Stream-Technik nicht angewendet werden. Der DesignManager gibt nun einen `ColibriOutputStream` an HADES zurück. Wenn HADES die Komponente fertig geschrieben hat, wird `close()` aufgerufen. An dieser Stelle leitet der `ColibriOutputStream` die kompletten Daten an den COLIBRI zum Speichern weiter. Damit dient auch der `ColibriOutputStream` als Adapter.

Um vom COLIBRI über ausgewählte Komponenten benachrichtigt zu werden, erfüllt der DesignManager das `SelectListenerInterface`. Der DesignManager wird hierüber vom COLIBRI aufgerufen, wenn Komponenten ausgewählt wurden. Er ruft dann seinerseits die entsprechenden Methoden in HADES auf.

SelectListenerInterface

```
/**
* The listener interface for receiving select events.
*
*/
public interface SelectListener extends ActionListener{
   /**
   * Invoked when a component has been selected.
   */
   public void componentSelected(SelectEvent e);

   /**
   * Invoked when a component has been opened.
   */
   public void componentOpened(SelectEvent e);
}
```

Dabei ermittelt der DesignManager sich zunächst den aktuellen Editor und prüft, ob dieser bereit ist. Dann erzeugt er einen neuen `ColibriCreateCommand` und übergibt diesem die ausgewählte Komponente. Der Editor springt nach vorne, und die Komponente hängt an der Maus, um plaziert zu werden

7.3.5 BeanExplorer

Zu dem COLIBRI-Package gehört auch ein `BeanExplorer`, mit dem genauere Informationen über JavaBeans angezeigt werden können. Der BeanExplorer ist dabei ein eigenständiges Package, das nicht von den in COLIBRI installierten Komponenten ausgeht. Statt dessen wird manuell oder per Datei-Requester ein Bean-Archiv angegeben, das der BeanExplorer öffnet. Es werden dann alle darin enthaltenen JavaBeans in einer Liste angezeigt. Wählt der Benutzer ein Bean aus, erscheinen im unteren Teil des Fensters

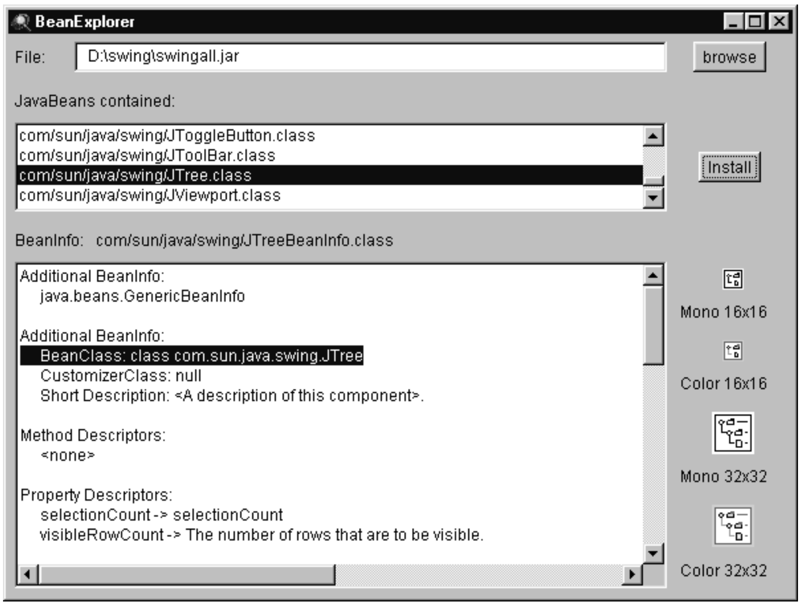

Abbildung 79: BeanExplorer

deren Eigenschaften, Methoden und Events angezeigt werden und die Beans auf Wunsch in den Browser installiert werden können.

7.3.6 Auswahlmechanismen

Der Browser soll, um möglichst intuitiv bedienbar zu sein, verschiedene Varianten einer Auswahl implementieren.

Der Doppelklick mit der Maus

Doppelklick
Da einfügen von Komponenten in Superkomponenten häufiger vorkommt als das Öffnen bestehender Superkomponenten haben die hierfür den Doppelklick verwendet.

DND
Wird zur Zeit noch nicht vollständig von Java implementiert. Es existiert zwar ein Entwurf und eine Teilimplementierung, dies ist jedoch noch nicht sinnvoll einsetzbar. Daher ...

Baut auf Uniform Data Transfer Mechanismus auf. Austausch von Daten Cut, Copy, Paste, „Drag and Drop"...
Framework für Kapselung und Austausch von Daten
Verschiedene Datentypen
Austausch zwischen Java und plattformabhängigen Anwendungen ermöglichen.

Interface Transferable
DataFlavour Datentypen, die ausgetauscht werden können Mime Types sollen verwendet werden.

Clipboard:
ClipboardOwner clipboard state changed

DragSource, DragContext, DropTarget, DropContext

Cut and Paste
Da der Java Cut and Paste Mechanismus zur Zeit nur sinnvoll mit Strings funktioniert, haben wir zur Realisierung ein eigenes Clipboard verwendet, auf das über die Methoden ... auch von außen zugegriffen werden kann.

Kontextmenü ist für Open implementiert außerdem für Select

Die Variante einer Task-Leiste ähnlich wie in Windows´95, die alle offenen Editor-Fenster enthält und auf die Komponenten per Drag and Drop gezogen werden können, haben wir nicht implementiert, da dies zu einer Abhängigkeit von Hades geführt hätte.

7.4 Der Server

Ein COLIBRI-Server stellt eine Menge von Komponenten zur Verfügung. Auf Anfrage schickt er Inhaltsverzeichnisse oder serialisierte Komponenten (s.u.) an die Clients. Außerdem stellt er einen Suchmechanismus zur Verfügung.

7.4.1 Netzwerk

Wahl des Protokolls -> Firewall

7.4.2 Firewalls

Da heute aus Sicherheitsgründen fast alle Intranets durch Firewalls vom Internet abgeschottet sind, ist eine direkte Socket-Kommunikation, d.h. eine direkte Netzwerk-Verbindung zwischen zwei Rechnern, nur in seltenen Fällen möglich. Üblich ist deshalb die Benutzung von sogenannten Proxy-Servern, die als Verbindung zwischen LAN und Internet fungieren. Anstatt einen Server direkt zu kontaktieren, beauftragt man einen solchen Proxy und erhält von diesem dann die Daten.

Da ein Proxy das benutzte Protokoll kennen muß, ist man an dessen Funktionsumfang gebunden. Für HTTP als derzeit wichtigstes Internetprotokoll sollte ein entsprechender Proxy überall anzutreffen sein. Durch die Verwendung von HTTP und der Möglichkeit einen Proxy anzugeben, können auch Anwender hinter einer Firewall mit COLIBRI arbeiten und Komponenten aus dem Internet benutzen. Eventuelle Vorteile eines eigenen anderen Protokolls können aus Anwendersicht nicht mithalten, da hierbei eine Kommunikation hinter Firewalls entweder nicht möglich ist oder ein eigener Proxy-Server implementiert und installiert werden müßte. Während eine Implementation nur zusätzlichen Aufwand bedeutet, ist eine Installation aus Sicherheitsgründen aber meistens gar nicht möglich.

7.4.3 HTTP

Für die Implementation des Servers gab es die Möglichkeit, entweder ein eigenes Protokoll zu entwickeln oder auf bestehende Protokolle zurückzugreifen. Die Wahl fiel hierbei auf das bestehende und erprobte HTTP-Protokoll. Dieses ermöglicht insbesondere eine Nutzung auch bei Verwendung von bestehenden Proxy-Servern.

Hypertext Transfer Protocol (HTTP):

> „The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems."
>
> (Berners-Lee, T. u.a.)

HTTP wird seit 1990 für das World Wide Web als Kommunikationsprotokoll zwischen Web-Browser und -Server verwendet. Es basiert auf einem Anfrage/Antwort Paradigma („request/response paradigm"). Ein Client (im WWW i.d.R. ein Browser) etabliert eine Verbindung mit einem Server und sendet eine Anfrage in Form eines URL mit weiteren Informationen wie Protokollversion, Anfragemethode und Client-Informationen. Der Server antwortet mit einem Erfolgs- oder Fehlercode, weiteren Informationen wie Protokollversion und Server-Informationen und ggf. den Daten der angeforderten Datei.

Http-Protokoll legt Schnittstelle zwischen Web-Server und Client fest Grundlage der Kommunikation ist die Socket-Schnittstelle

Das HTTP-Protokoll ist ein sogenanntes „zustandsloses" Protokoll, bei dem nach jeder Kommunikation die Verbindung wieder abgebaut wird. Eine fortlaufende Assoziation eines Benutzers mit einer Netzwerkverbindung und eine Auswertung der bisherigen Kommunikation (Zustand) ist daher nicht möglich. Eine HTTP-Kommunikation besteht dadurch immer aus einer Client-Anfrage und einer Server-Antwort mit abschließendem Verbindungsabbau.

Die Client-Anfrage besteht aus einer Kommandozeile und mehreren Informationszeilen (dem sogenannten „Header"), einer Leerzeile als Trennung und bei Bedarf einem Datenblock. Der Server schickt daraufhin eine Responsezeile (enthält Erfolgs- bzw. Fehlercode und Klartext), die wiederum von einem Header und optionalen Datenblock gefolgt wird.

Soll z.B. die Startseite vom Server „www.microsoft.com" geladen werden, dann wird die Kommadozeile folgendermaßen aussehen:

GET /index.html HTTP/1.0

Das eigentliche Kommando „GET" besagt, daß man ein Dokument vom Server anfordert. Der darauf folgende Teil („/index.html") spezifizert das gewünschte Dokument. Dahinter folgt die benutzte HTTP-Version.

Der Server könnte daraufhin folgendermaßen antworten:

HTTP/1.0 200 OK

Server: Microsoft-PWS-95/2.0
Date: Sun, 08 Feb 1998 14:18:12 GMT
Content-type: text/html
Content-length: 1234
Last-Modified: Sun, 08 Feb 1998 14:04:46 GMT

<HTML> ....</HTML

Die oberste Zeile enthält neben der HTTP-Version einen Ergebnis-Code (200) und eine textuelle Kurzbeschreibung dazu (OK). Die folgenden Headerzeilen besagen, daß der Server ein Microsoft Personal-Web-Server ist und ein 1234 Bytes langer Datenblock vom Typ text/html folgt, der zuletzt am 8. Februar um 14:04 Uhr geändert wurde.

Die Headerzeilen sind beliebig erweiterbar und enthalten meist viele Informationen über die benutzen Browser, Server, Systeme, etc.
Weitere Informationen über das HTTP-Protokoll und die genaue Spezifikation finden sich unter [] http://www.w3.org.

Ein Proxy-Server wird grundsätzlich genauso kontaktiert, nur daß dieser Informationen über den zu kontaktierenden Server benötigt (Name, Port, Protokoll, etc.), die bei einer direkten Kommunikation überflüssig sind. Dem GET-Kommando wird deshalb die gesamte URL übergeben:

GET http://www.sun.com/index.html HTTP/1.0

Der Proxy-Server kontaktiert daraufhin den in der URL spezifierten Server und leitet die Ergebnisdaten HTTP-konform an seinen Client weiter.

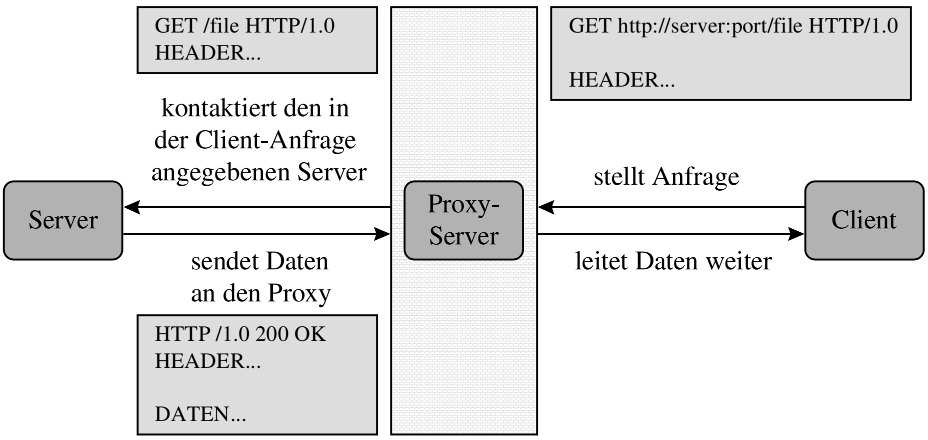

Abbildung 80: Zugriff über einen Proxy-Server

7.4.4 Verwendung von HTTP im COLIBRI

Die vom COLIBRI-Server bereitgestellten Daten sind grundsätzlich serialisierte Objekte. Hiefür gibt es den Datentyp „application/java-serialized-object". Das Inhaltsverzeichnis z.B. ist ein serialisierter Tree (TNodes), den man mit „GET / HTTP/1.0" bekommt. Mit „GET /models HTTP/1.0" bekommt man nur den entsprechenden Teil des Trees. Eine

Komponente referenziert man z.B. mit „GET /models/gates/Nand2 HTTP/1.0“. Der Server liefert dann alle Komponenten-Bestandteile in einem serialisierten BeanWrapper zurück. Wurde nur eine einzelne Datei einer Komponente angefordert (siehe clb-URL), dann bekommt man vom Server einen serialisierten ByteArrayInputStream.

Außer dem GET-Kommando werden vom COLIBRI-Server derzeit noch keine weiteren HTTP- Kommandos (HEAD, POST, PUT, etc.) unterstützt. „POST“ oder „PUT“ können jedoch beim Speichern von Komponenten und „HEAD“ beim Versionsmanagement Einsatz finden. Insbesondere beim Speichern, aber auch für private Komponenten auf einem Server, kann die in HTTP vorgesehene Authentisierung nützlich sein.

Interessiert man sich bei einem Server nur für einen bestimmten Teil (Subtree) der Komponenten, so kann man der Teilpfad zu diesen Komponenten im Bookmark mit angeben und im Tree wird nur der entsprechende Subtree eingeblendet. Werden dann Komponenten geladen, die auf andere Komponenten außerhalb dieses Subtrees verweisen kann der Browser sie trotzdem laden, da er ja den Server kennt und den Pfad innerhalb des Servers aus der Designdatei entnehmen kann.

7.4.5 Technischer Aufbau

Das Serverprotokoll wird im COLIBRI über die Klassen Client, Server und Connection abgewickelt. Der Browser stellt über den Client eine Verbindung zum Server her und stellt eine Anfrage. Der Server wartet auf seinem Port auf Client-Anfragen und kreiert bei einer Anfrage einen Connection-Thread, der dann über Streams direkt mit dem Client kommuniziert.

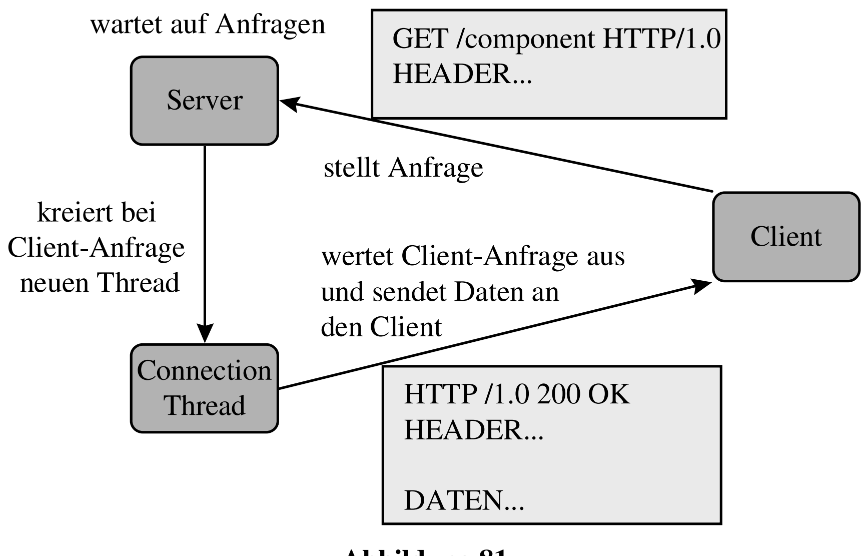

Abbildung 81

Der Server erzeugt zunächst einen Socket auf dem entsprechenden Port. Dann startet er den Server-Thread. Dieser versucht in einer Endlosschleife Client-Anfragen entgegenzunehmen. Die Methode `listen_socket.accept()` kehrt erst dann zurück, wenn eine Anfrage eingetroffen ist. Der Server erzeugt dann ein `Connection`-Objekt und springt sofort wieder in die `listen_socket.accept()`-Methode, um auf weitere Anfragen zu warten.

```
listen_socket = new ServerSocket(port);
this.start();

public void run()
{
   try
   {
      while(true)
      {
         Socket client_socket = listen_socket.accept();
         Connection c = new Connection(client_socket, top, base-
dir);
         if (debug) message("new connection accepted");
      }
   }
   catch (Exception e)
   {
      message("Exception while listening for connections: "+e);
   }
}
```

Die Connection öffnet nun zwei Streams, über die sie mit dem Client kommuniziert.

```
// open streams to communicate with the client
in = new DataInputStream(client.getInputStream());
out = new DataOutputStream(client.getOutputStream());
```

Dann liest sie aus dem DataInputStream die Anfrage des Client aus und liefert das Ergebnis zurück. Hierbei kann es sich um einen Baum oder Teilbaum, eine Komponente oder ein Suchergebnis handeln.

8 Zusammenfassung

8.1 Bewertung der Ergebnisse

Für die HADES-Umgebung wurde ein Werkzeug benötigt, mit dem Komponenten komfortabel ausgewählt und in den Editor geladen werden können. Dabei sollen dem Benutzer möglichst viele Informationen über die Komponenten zur Verfügung gestellt werden, um eine qualifizierte Auswahl zu ermöglichen.

Der COLIBRI ist ein sehr nützliches Werkzeug zur Verwaltung von Komponenten. Er stellt ausführliche Informationen über die Komponenten zur Verfügung und ermöglicht so eine qualifizierte Auswahl.

Als eines der größten Hindernisse stellte sich wie beschrieben das Laden von Klassen in den Java-Interpreter heraus. Allein die Ursachenforschung nahm dabei sehr viel Zeit in Anspruch. Da wie beschrieben die HADES-Simulationskomponenten keine echten Java-Beans sind, mußten einige spezielle Anpassungen vorgenommen werden, wobei jedoch die Allgemeingültigkeit des COLIBRI nicht verlorengehen sollte. Dies ist generell gelungen, so daß der Browser auch für normale JavaBeans und theoretisch auch für andere Komponenten einsetzbar ist.

Ein sehr nützliches Werkzeug ist der BeanExplorer, der auch außerhalb des COLIBRI einsetzbar ist. Mit ihm können allgemeine Beans analysiert werden.

Bei der Entwicklung des COLIBRI wurden von uns jeweils die neusten verfügbaren Versionen das JDK und der Java Foundation Classes (Swing) verwendet. Da diese noch nicht völlig ausgereift und fehlerhaft waren, traten in diesem Bereich einige Probleme auf. So stand am Anfang noch keine Dokumentation zu den von uns verwendeten Komponenten zur Verfügung. Außerdem änderte sich die API noch mehrmals. Ein Großteil der Zeit mußte deshalb zur ständigen Fehlersuche, anstatt zur Konzeptumsetzung aufgewendet werden. Im Rahmen einer Forschungsarbeit wollten wir jedoch auf den Einsatz der jeweils neusten Technologie nicht verzichten.

8.2 Ausblick

Um den allgemeinen Einsatz zu demonstrieren, könnte der COLBRI zusammen mit der BeanBox eingesetzt werden.

Serverarchitektur
Ab Java Version 1.2 wird ein ausgefeiltes Sicherheitskonzept mit verschiedenen Sicherheitsstufen vom System zur Verfügung gestellt. Dieses könnte im COLIBRI für die einzelnen Komponenen Anwendung finden.

Drag and Drop wird in zukünftigen Java Versionen integriert sein. Dann bietet es sich an, eine Komponentenauswahl auch mit Hilfe dieses Mechanismuses zu ermöglichen.

Benutzerprofile

Die Suche nach Komponenten kann auch über boolesche Operatoren erfolgen.
Trader zur Suche von Komponenten [Müller-Jones et al. 95], Typmanagement [Christiansen et al. 97]

Ein Einsatz von CORBA oder RMI erscheint im Rahmen von HADES nicht sinnvoll. Im Hinblick auf einen allgemeinen Einsatz des COLIBRI auch für andere Komponenten läßt sich jedoch das Ausführen von Komponenten auf entfernten Rechner in den COLIBRI integrieren.

9 Literaturverzeichnis

[Andrews 96]	*IBM's San Francisco Project, Java Frameworks for Application Development*, D.H. Andrews Group, Inc., 1996, http://www.dhagroup.com/r_sanfra.html
[Andrews 98]	Mark Andrews, *Introducing Swing,* Sun Microsystems, Inc., Mountain View CA, Mai 1998, http://web3.javasoft.com/products/jfc/swindoc-static/what_is_swing.html
[Arnold & Gosling 96]	Ken Arnold und James Gosling, *The Java Programming Language,* Addison Wesley Publishing Company, Inc., 1996
[Box 95]	Don Box, *The Component Object Model and Some Other Model*, 1995 http://www.develop.com/dbox/COM_vs_SOM_Summ.htm
[Brando 95]	Thomas J. Brando, *Comparing DCE and CORBA*, MITRE Document, The Mitre Corporation, 1995 http://www.mitre.org/research/domis/reports/DCEvCORBA.html
[Cadence 97]	Cadence Design Framework II 4.4.1, Cadence Design Systems, Inc., San Jose, CA, 1997
[Campione & Walrath]	Mary Campione, Kathy Walrath, *The Java Tutorial, Online Edition*, Sun Microsystems, Inc., http://java.sun.com/docs/books/tutorial/index.html
[Caribou 97]	*RMI versus CORBA*, Caribou Lake Software Inc., 1997, http://www.cariboulake.com/techinfo/rmi_corba.html
[Chan & Lee 96]	Patrick Chan, Rosanna Lee, *The Java Class Libraries*, Addison Wesley Publishing Company, Inc., 1996
[Chen 97]	Weiying Chen, *ActiveX Programming unleashed*, Sams net, 1997
[Christiansen et al. 97]	Bernd Christiansen, Malte Münke, Kai Müller-Jones, Winfried Lamersdorf, *A Key to Software Reuse in Open Distributed Systems*, in Proceedings of the First International Workshop on Enterprise Distributed Object Computing, EDOC´97, Australia, S. 78-89. IEEE, Oktober 1997
[COM 95]	*The Component Model Specification*, Microsoft Corporation, 1995 http://www.microsoft.com/com/com1598B.zip
[COM 97]	*COM Reality Tour*, Microsoft Corporation, 1997 http://www.microsoft.com/com/slides/realtour.zip

[Components 97] *What are Components?,* Komponenten-Seminar, VSYS, 1997, http://vsys-www.informatik.uni-hamburg.de/teaching/ws-97.98/ComponentSeminar/links/WhatComponents.htm

[CORBA 97] *CORBA for Java Programmers*, Sun Microsystems, Inc., 1997, http://java.sun.com/products/jdk/idl/docs/CORBAJava.html

[CORBA ORB 98] *The Common Object Request Broker: Architecture and Specification*, OMG, Februar 1998 http://www.omg.org/library/specindex.htm

[CORBA Overview] *Object Management Architecture Overview*, The Object Management Group, http://www.omg.org/about/omaov.htm

[CORBA Services 97] *CORBAservices: Common Object Services Specification*, OMG, November 1997, http://www.omg.org/library/specindex.htm

[CORBA] *What IS CORBA????,* The Object Management Group, http://www.omg.org/about/wicorba.htm

[Cornell & Horstmann 96] Garry Cornell, Cay S Horstmann, *Core Java,* Prentice Hall PTR, 1996

[Curtis 97] David Curtis, *Java, RMI and CORBA, A White Paper*, Object Management Group, 1997, http://www.omg.org/news/wpjava.htm

[Cyberdog 96] *Cyberdog General Information*, Apple Computer, Inc., 1996 http://www.cyberdog.apple.com/geninfo.html

[Dalheimer 98] Kalle Dalheimer, *Klassenkampf in Java*, c´t Magazin für Computertechnik, S.198 ff., 2/98

[Davis & Lagoze 94] James R. Davis, Carl Lagoze*, A protocol and server for a distributed digital technical report library*, 1994, http://cs-tr.cs.cornell.edu/Dienst/Repository/2.0/Body/ncstrl.cornell/TR94-1418/postscript

[Davis & Lagoze 96] James R. Davis, Carl Lagoze*, The Networked Computer Science Technical Report Library*, 1996

[Davis 95] James R. Davis, *Creating a Networked Computer Science Technical Report Library*, D-Lib Magazine, September 1995, Xerox Corporation, http://www.dlib.org/dlib/september95/09davis.html

[DeSoto 97] Alden DeSoto, *Using the Beans Development Kit 1.0, A Tutorial*, Sun Microsystems, Inc., 1997, http://java.sun.com/beans

[Dienst 95] *Dienst protocol version 4.0 DRAFT*, NCSTRL Documentation, 1995, http://www.ncstrl.org/Dienst/htdocs/Info/protocol4.html

[DigSim 96] *DigSim Version 1.00b3*, Iwan van Rienen, 1996, http://www.lookup.com/Homepages/96457/home.html

[Eng 98] Ben Eng, *Corba Feature Matrix*, Juni 98, http://www.vex.net/~ben/orbmatrix.html

[EnterprBean 97] *Sun releases Draft Enterprise JavaBeans Specification for public Review, Press Release,* Sun Microsystems, Inc., http://web3.javasoft.com/pr/1997/dec/pr971210-01.html

[Flanagan 96] David Flanagan, *Java 1.1 in a Nutshell,* O´Reilly & Associates, Inc, 1997

[Fowler 98] Amy Fowler, An Overview Of Swing Architekture, Sun Microsystems, Inc., Mountain View CA, Mai 1998, http://web3.javasoft.com/products/jfc/swindoc-static/swing_arch.html

[Frey 97] Anthony Frey, *Into Orbit*, Network Computing Online, Februar 97, http://techweb.cmp.com/nc/804/804f1.html

[Fritzinger & Mueller 96] J. Steven Fritzinger, Marianne Mueller, Java Security, Sun Microsystems, Inc., 1996, http://java.sun.com/security/whitepaper.ps

[FUHagen RMI 97] *Java Remote Method Invocation*, Seminar SS 1997: Programmierung im Großen, FernUni Hagen, 1997, http://www.stud.fernuni-hagen.de/q3283267/

[Gamma et al. 94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns,* Addison Wesley Publishing Company, Inc., 1994

[Gong 98] Li Gong, *Security on the Java Platform: Recent Developments and Future Directions*, Java One, Sun Microsystems, Inc., 1998, http://java.sun.com/javaone/javaone98/sessions/T101

[Gong et al. 97] Li Gong, Marianne Mueller, Hemma Prafullchandra, Roland Schemers, *Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2*, in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, JavaSoft, Sun Microsystems, Inc., Dezember 1997, http://java.sun.com/security/usenix-jdk12security.ps

[Gosling et al. 96] James Gosling, Bill Joy, Guy Steel, *The Java Language Specification,* Sun Microsystems, Inc., Addison Wesley Publishing Company, Inc. 1996, http://java.sun.com/docs/books/jls/index.html

[Griffel 96] Frank Griffel, *Komponentenbasierte Entwicklung interoperabler Software in heterogenen verteilten Systemen,* Diplomarbeit, Arbeitsgruppe Verteilte Systeme, Fachbereich Informatik, Universität Hamburg, 1996

[Griffel 98] Frank Griffel, *Componentware,* dpunkt.verlag, 1998

[HADES 97] Norman Hendrich, *HADES Tutorial,* Version 0.43, 1997

[Hendrich 97 b] Norman Hendrich, *A Java-based visual simulation environment,* Fachbereich Informatik, Universität Hamburg, 1997

[Hendrich 97] Norman Hendrich, *Java für Fortgeschrittene,* Springer Verlag, Heidelberg, 1997

[Java Enterprise 98] *Enterprise JavaBeans,* Sun Microsystems, Inc., Mountain View CA, 1998 http://web3.javasoft.com/products/ejb/docs.html

[Java Language 95] *The Java Language, An Overview*, Sun Microsystems, Inc., Mountain View CA, 1995, http://java.sun.com/docs/overviews/java/java-overview-1.html

[Java OSS 96] *Java Object Serialization Specification,* Sun Microsystems, Inc., Mountain View CA, 1997, http://java.sun.com/products/jdk/rmi/serial/index.html

[Java Reflect 97] *Java Core Reflection API and Specification,* Sun Microsystems, Inc., Mountain View CA, 1997, http://java.sun.com/products/jdk/1.1/docs/guide/reflection

[Java RMI 97] *Java RMI Specification,* Sun Microsystems, Inc., Mountain View CA, 1997, http://web3.javasoft.com/products/jdk/rmi/index.html

[Java RMI Tut 96] *Java RMI Tutorial,* Sun Microsystems, Inc., Mountain View CA, 1997 http://web3.javasoft.com/products/jdk/rmi/index.html

[Java Studio RevGuide 97] *Java Studio Reviewer´s Guide,* Sun Microsystems, Inc., Mountain View CA, 1997, http://www.sun.com/studio/

[Java Studio 97] *Java Studio, Product Overview,* Sun Microsystems, Inc., Mountain View CA, 1997, http://www.sun.com/studio/

[Java Swing 97] *Swing The Preliminary Specification,* Sun Microsystems, Inc., Mountain View CA, 1997

[JavaBeans 96] *JavaBeans Specification,* Sun Microsystems, Inc., Addison Wesley Publishing Compoany, Inc. 1996 http://java.sun.com/beans

[Keahey] Kate Keahey, *A Brief Tutorial on CORBA*, http://www.cs.indiana.edu/hyplan/kksiazek/tuto.html

[Kindel 97] Charlie Kindel, *ActiveX Overview*, Microsoft Corporation, 1997, http://www.microsoft.com/com/slides/kindel1.zip

[Lagoze et al. 95] Carl Lagoze, Erin Shaw, James R. Davis, Dean B. Krafft, *Dienst: Implementation Reference Manual*, 1995, http://cs-tr.cs.cornell.edu/ Dienst/Repository/2.0/Body/ncstrl.cornell/ TR95-1514/postscript

[Loviscach 97] Dr. Jörn Loviscach, *Paßgenau, Mit Komponenten zur persönlichen Software,* c´t Magazin für Computertechnik, S.224 ff., 3/97

[Maslo 97] Andreas Maslo, *Verbinden statt erfinden,* c´t Magazin für Computertechnik, S.230 ff., 3/97

[Meyer 88] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988

[Minton 96] Gabriel Minton , Programming With CORBA, UNIX Review, 1996, http://www.blackmagic.com/people/gabe/prog-with-corba.html

[Moeller 98] Michael Moeller, *COM+ services add up*, PC Week Online, 1998, http://www.zdnet.com/pcweek/news/0413/13com.html

[Mössenböck 96] Hanspeter Mössenböck, *Trends in Object-Oriented Programming,* Johannes Kepler University Linz, Institute of Computer, 1996, http://www.ssw.uni-linz.ac.at/pub/Papers/SDCR.HM.html

[Müller-Jones et al. 95] Kai Müller-Jones, Michael Merz, Winfried Lamersdorf*, Realisierung von Kooperationsanwendungen auf der Basis erweiterter Diensttypbeschreibungen*, in H. Krumm, Hrsg. Entwicklung und Management verteilter Anwendungssysteme - Tagungsband des 2. Arbeitstreffens der GI / ITG Fachgruppe „Kommunikation und verteilte Systeme“ und der GI Fachgruppe“ Betriessysteme“, S. 20-30. Universität Dortmund, Oktober 1995

[NCSTRL] *NCSTRL Technical Overview,* NCSTRL Documentation, http://www.ncstrl.org/Dienst/htdocs/Info/architecture.html

[Orfali & Harkey 97] Robert Orfali, Dan Harkey, *Client/Server Programming with Java and CORBA*, Wiley, 1997

[Orfali et al. 96] Robert Orfali, Dan Harkey, Jeri Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, Inc., 1996

[Page 91] Bernd Page, *Diskrete Simulation*, Springer Verlag, 1991

[Pfister 95] Cuno Pfister, *Componentware Standards*, The Oberon Tribune, Vol 1, No 2, Oberon Microsystems, 1995, http://www.oberon.ch/services/odf/tribune/trib2/standards.html

[Pountain & Montgomery 97] Dick Pountain and John Montgomery, *Components and the Web are a match made in developer heaven,* 1997, http://www.byte.com/art/9708/sec5/art1.htm

[Pure RMI 98] *JAVA REMOTE METHOD INVOCATION, Pure Java Distributed Programming*, Sun Microsystems, Inc., Mountain View CA, Mai 1998, http://java.sun.com/marketing/collateral/javarmi.html

[RMI Whitepaper 98] *JAVA REMOTE METHOD DISTRIBUTED COMPUTING FOR JAVA*, Sun Microsystems, Inc., Mountain View CA, Mai 1998, http://java.sun.com/marketing/collateral/javarmi.html

[Rumbaugh et al. 91] J. Rumbaugh, M. Blahe, W. Premerlani, F. Eddy, W. Lorensen, Object-oriented Modeling and Design, Prentice-Hall, 1991

[Schmidt 97] Douglas C. Schmidt, *Overview of CORBA*, 1997, http://www.cs.wustl.edu/~schmidt/corba-overview.html

[Schwarz 97] Martin Schwarz, *Moment, ich verbinde...,* c´t Magazin für Computertechnik, S.256 ff., 3/97

[SOM 94] *The System Object Model (SOM) and the Component Object Model (COM)*, IBM Corporation Austin, 1994 http://www.software.ibm.com/ad/somobjects/library/somvscom.html

[Tanenbaum 95] Andrew S. Tanenbaum, *Verteilte Betriebssysteme*, Prentice Hall, 1995

[Thomas 97] Anne Thomas, *Enterprise JavaBeans, Server Component Model for Java,* Patricia Seybold Group, 1997, http://web3.javasoft.com/products/ejb/docs.html

[Udell 94] Jon Udell, *Componentware,* Byte Magazine, Mai 1994, http://www.byte.com/art/9405/sec5/art1.htm

[Voyager 97a] *ObjectSpace Voyager Core Technology User Guide*, ObjectSpace, Dezember 1997, http://www.objectspace.com

[Voyager 97b] *ObjectSpace Voyager Core Package Technical Overview*, ObjectSpace, Dezember 1997, http://www.objectspace.com

[Weber 95] Joseph Weber, *Special Edition Using Java, 2nd Edition,* Que Corporation, 1996

[Weyerhäuser 97] Markus Weyerhäuser, *Echte Bohne,* c´t Magazin für Computertechnik, S.300 ff., 9/97

[Yang & Duddy 96] Zhonhua Yang, Keith Duddy, *CORBA: A Platform for Distributed Object Computing*, in *AVM Operating Systems Review*, Vol. 30, No. 2, Seite 4-31, April 1996, http://www.dstc.edu.au/AU/research_news/omg/report/OSP.ps

10 Anhang A: Tutorial

Contents

11 Anhang B: Format der CLB-Datei

Eine CLB-Datei enthält alle für den COLIBRI benötigten Informationen über die dazugehörige installierte Komponente. Die enthalten Daten werden als ASCII-Text gespeichert. Eine CLB-Datei besteht aus mehreren Blöcken, die jeweils durch ein Schlüsselwort in eckigen Klammern eingeleitet werden. Dabei sind einige Blöcke zwingend, andere optional. Der Aufbau einer CLB-Datei ist offen für Erweiterungen.

Momentan werde folgende Blöcke unterstützt:

[files x]	zwingend	die nächsten x Zeilen enthalten alle direkt zur Komponente gehörigen Dateien
[type]	optional	Typ der Komponente (beans / AxctiveX / OpenDoc...)
[references x]	optional	die nächsten x Zeilen enthalten die URLs der direkten Sub-Komponenten (zur Zeit nicht verwendet)
[archive]	optional	Name des Archives, das alle zur Komponente gehörigen Dateien enthält
[start]	optional	z.B. Name der Startklasse
[init]	optional	String, der der Initialisierungs-Methode übergeben wird (kann z.B. eine ini-Datei sein: .hds / .fsm / .kvd)
[icon]	optional	Name des Icons
[image]	optional	Name des Images
[description]	optional	Beschreibung der Komponente (auf eine Zeile kodiert)
[html]	optional	HTML-Seite zur Beschreibung der Komponente

Ein Beispiel für ein einfaches Gatter könnte folgendermaßen aussehen:

```
[files 4]
hades/models/gates/And2.class
hades/models/gates/And2.sym
hades/models/gates/And2.gif
hades/models/gates/And2.html
[versions 1]
1000
[references 0]
[type]
beans
[start]
hades.models.gates.And2
[icon]
simobj.gif
[image]
hades/models/gates/And2.gif
[description]
AND Gate with 2 Inputs
[html]
hades/models/gates/And2.html
```

12 Anhang C: Stichwortverzeichnis

API	*Application Programming Interface*, Programmierschnittstelle
ClassLoader	ist für das Laden von Klassen in den Java-Interpreter zuständig
COLIBRI	*Component- and Library-Browser*, Inhalt dieser Arbeit
COM	Komponentenarchitektur von Mirosoft
Compiler	Übersetzungsprogramm von Quellcode, etc.
Copy and Paste	Kopieren und Einfügen von Objekten über ein Clipboard
CORBA	*Common Object Request Broker Architecture*, eine von der OMG spezifizierte Architektur zur verteilten Objektkommunikation
Drag and Drop	Einbetten graphischer Objekte durch Ziehen und Fallenlassen mit der Maus
GUI	*Graphical User Interface*, graphische Benutzeroberfläche
HADES	*Hamburger Designsystem*, Simulationssystem für das der Browser entwickelt wurde
HTTP	*Hypertext Transfer Protocol*, Übertragungsprotokoll im Internet
Interpreter	Ausführendes Programm z.B. von Skripten oder Java-Bytcode
Java	Objektorientierte und plattformunabhängige Programmiersprache der Firma Sun Microsystems
Java Foundation Classes / Swing	Sammlung neuer graphischer Benutzerelemente (GUI-Elemente) komplett in Java geschrieben
JavaBeans	Komponentenarchitektur in Java
Komponente	eigenständiger, wiederverwendbarer Softwarebaustein
Marshaling	Verpacken der Parameter einer Methode zum Senden über das Netz
NCSTRL	*Networked Computer Science Technical Report Library*, System zur Verwaltung von Forschungspapieren
Object Serialization	Mechanismus zum Abspeichern, Übertragen, etc. von Objekten (Persistenz)
OMG	*Object Management Group*, Standardisierungsgremium (siehe CORBA)
Open Doc	Komponentenarchitektur der CI-Labs
Reflection	Mechanismus zum dynamischen Inspizieren von Objekten
RMI	*Remote Method Invocation*, Architektur für verteilte Java-Objekte
Skeleton	Stellvertreterobjekt auf Serverseite

SOM	Komponentenarchitektur von IBM
Stub	lokales Stellvertreterobjekt für eine entfernte Komponente

13 Erklärung

Hiermit erklären wir, die vorstehende Arbeit selbständig und ohne fremde Hilfe unter ausschließlicher Nutzung der angegebenen Quellen und Hilfsmittel angefertigt zu haben.

Kapitel 1.5-1.7, 2, 3, 5, 7.1, 7.2 und das Tutorial wurden von Karola Krönert angefertigt. Die Kapitel 1.1-1.4, 2.2, 3.2, 4, 6 und 7.1 wurden von Ulrich Dallmann erstellt. Kapitel 8 und die Anhänge sind eine Gemeinschaftsarbeit.

Hamburg, den 30.06.1998