Diplomarbeit

Komponentenorientierte Werkzeugkonstruktion

Entwurf und Implementierung eines Werkzeugkomponentenmodells

vorgelegt von

Frank Fröse
Bornbruch 5 b
21521 Wohltorf
Matrikel-Nr. 4561975

November 1999

Diese Diplomarbeit wurde am Fachbereich Informatik der Universität Hamburg zur teilweisen Erfüllung der Anforderungen zur Erlangung des Titels Diplom-Informatiker eingereicht.

Erklärung
Hiermit versichere ich, diese Arbeit selbständig und unter ausschließlicher Zuhilfenahme der in der Arbeit aufgeführten Hilfsmittel erstellt zu haben.

Hamburg, den 11.11.1999

Frank Fröse

Bornbruch 5 b
21521 Wohltorf
Matrikel-Nr. 4561975

E-Mail: Frank.Froese@bigfoot.com

Betreuung
Erstbetreuer: Prof. Dr. Heinz Züllighoven
Zweitbetreuer: Dr. Walter Bischofberger

Prof. Dr. Heinz Züllighoven
Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Straße 30
D-22527 Hamburg
Deutschland

Dr. Walter Bischofberger
TakeFive Software AG
Eidmattstrasse 51
CH-8052 Zürich
Schweiz

Inhaltsverzeichnis

Abbildungsverzeichnis

Danksagung

Ich möchte mich bei Prof. Heinz Züllighoven und Dr. Walter Bischofberger für die Betreuung dieser Diplomarbeit bedanken. Mein besonderer Dank gilt darüber hinaus Dirk Riehle, der den Entstehungsprozeß die ganze lange Zeit über begleitet hat und dessen Kritik und Diskussionsbeiträge für das Gelingen der Arbeit sehr wichtig waren.

1 Einleitung

In vielen Ingenieursbereichen erfolgt heute die Produktentwicklung auf der Basis von Komponenten. Komponenten mit definierten oder genormten Eigenschaften bilden die Grundbausteine bei der Konstruktion komplexer Systeme. Beispiele hierzu finden sich in der Automobilbranche, im Bereich elektronischer Systeme oder auch beim Hausbau. Komponentenbasierte Architekturen helfen den Herstellungsvorgang zu beschleunigen und sind teilweise sogar unabdingbar, um auch die Konstruktion komplexer Produkte überhaupt sinnvoll handhaben zu können. Durch komponentenbasierte Entwicklungstechniken läßt sich grundsätzlich ein hoher Grad an Arbeitsteilung erreichen, da die Herstellung einzelner Komponenten im allgemein von der Herstellung des Gesamtprodukts, in das sie eingehen, getrennt werden kann.

Ein weiterer Vorteil der komponentenbasierten Entwicklung ist die Möglichkeit einzelne Komponenten - innerhalb gewisser Spezifikationsgrenzen - auch nachträglich noch auszutauschen, ohne daß die Stabilität oder Gebrauchsqualität des Gesamtprodukts davon negativ beeinflußt würde. Beispielsweise kann Verschleiß oder veränderte Anforderungen an ein Produkt den Austausch von Komponenten erforderlich machen.

Der Wunsch Komponententechniken auch auf den Bereich der Softwareentwicklung zu übertragen ist nicht neu. Schon 1987 prägte B. Cox den Begriff des Software-IC [Cox87]. Die Konstruktion von Softwaresystemen aus existierenden, vorgefertigten Softwarekomponenten weckt große Hoffnung im Softwareentwicklungsumfeld. Vor allem unter dem Aspekt der Wiederverwendbarkeit werden Softwarekomponenten in einigen Veröffentlichungen als die allmächtige Lösung heutiger Softwareprobleme und als zukünftiges Paradigma der Softwareentwicklung angesehen. Provokant prophezeite J. Udell 1994 in der Zeitschrift Byte gar das Ende der Objektorientierung [Udell94], da Beispiele wie Microsofts Entwicklungsumgebung VisualBasic bewiesen hätten, daß die Komponentenorientierung einen Grad an Wiederverwendung erlaubt, den die Objektorientierung nie erreicht habe.

Bei näherer Betrachtung zeigt sich jedoch, daß die Entwicklung komponentenbasierter Architekturen nicht so unproblematisch ist, wie es auf den ersten Blick erscheint. Die einfache Wiederverwendung und Integration von Komponenten ist nicht automatisch eine den komponentenorientierten Techniken inhärente Eigenschaft. Garlan et al. haben aufgezeigt, daß die Hauptprobleme bei der Entwicklung komponentenbasierter Systeme weniger auf technischen Implementierungsaspekten beruhen, als auf abstrakten Architekturunverträglichkeiten (Architectural Mismatch) zwischen den beteiligten Komponenten [GAO95]. Neben der Zergliederung eines Systems in Komponenten ist immer auch ein klares Komponentenmodell erforderlich, das die Art und Weise des Zusammenspiels der verschiedenen Komponenten definiert. Grundlage für eine komponentenorientierte Softwareentwicklung bilden anwendungs- und produktspezifische Rahmenwerke, die die prinzipiellen Anforderungen an verwendbare Komponententypen spezifiziert sind und die generellen Architekturprinzipien eines Systems festschreiben. Der von Udell vorgenommene schlichte Vergleich Komponentenorientierung versus Objektorientierung vermischt Aspekte der

Systemarchitektur mit Fragen der Systemimplementation. Eine differenziertere Betrachtung der Aspekte der Komponentenorientierung ist deshalb erforderlich.

Im Vergleich zu gängigen objektorientierten Ansätzen betonen komponentenorientierte Techniken stärker die Außensicht von Softwarebausteinen. Wesentlich in komponentenorientierten Ansätzen ist eine strenge Trennung der Schnittstellendefinition von der Komponentenimplementation. Das Geheimnisprinzip (vgl. [Parnas72]) ist bei Komponenten im allgemeinen noch stärker ausgeprägt als bei Objekten in objektorientierten Modellen. Die starke Abstraktion von der Implementation ist der Schlüssel zur Wiederverwendung und zum Plug-and-Play bei der Entwicklung komponentenorientierter Systeme.

1.1 Zielsetzung der Arbeit

Im Rahmen des WAM[1]-Ansatzes werden Softwarewerkzeuge bisher ausschließlich mit objektorientierten Modellen entworfen und konstruiert (vgl. [Züllighoven98]). Die objektorientierte Abstraktionen Klasse und Objekt erweisen sich jedoch häufig als zu feingranular für die Beschreibung von Softwarearchitekturen. Der reguläre Aufbau von Softwarewerkzeugen legt hier einen komponentenorientierten Ansatz nahe. Softwarewerkzeuge lassen sich als Hierarchien von Werkzeugkomponenten modellieren und konstruieren, deren Kommunikationsstrukturen denen einer Bürokratie gleichen (vgl. [Riehle98]).

Ziel dieser Diplomarbeit ist es zu beschreiben, wie ein Komponentenmodell für die Entwicklung von Softwarewerkzeugen aussehen könnte und wie sich solch ein Modell in der Programmiersprache Java implementieren läßt.

Es wird eine von mir entwickelte Modellarchitektur für die komponentenorientierten Konstruktion von Softwarewerkzeugen vorgestellt, die Werkzeugarchitekturen in Form von Werkzeugkomponenten, Komponentenschnittstellen, Verknüpfungsbeziehungen, Interaktions- und Kompositionsregeln beschreibt. Das Modell wurde im Rahmen der Diplomarbeit von mir in Java implementiert. Eine Diskussion verschiedener Implementierungsvarianten zeigt den Spielraum, der bei Abbildung komponentenorientierter Modelle auf objektorientierte Programmiersprachen vorhanden ist. Es wird deutlich, daß mit einem komponentenorientierten Ansatz die Frage nach der am besten geeigneten Komponentenimplementation unabhängig von der Architektur eines Werkzeugs beantwortet werden kann.

1.2 Aufbau der Arbeit

In Kapitel 2 werden die grundlegenden Begriffe *Komponente* und *Komponentenklasse* wie sie in der Literatur verwendet werden erörtert und hieraus Definitionen und graphische Notationen für die weitere Verwendung in der Diplomarbeit entwickelt.

In Kapitel 3 werden die zwei zur Zeit wichtigsten verfügbaren Komponentenstandards vorgestellt: Microsofts Component Object Model (COM) und Javas Komponentenmodell JavaBeans.

[1] WAM ist ein Akronym für **W**erkzeug-**A**utomat-**M**aterial und bezeichnet die drei Hauptentwurfsmetaphern des methodischen Ansatzes.

Kapitel 4 diskutiert verschieden Arten des Werkzeugentwurfs. Neben der *klassischen* Form des WAM-Ansatzes wird ein komponentenorientierter Ansatz des Werkzeugentwurfs vorgestellt. Für den komponentenorientierten Werkzeugentwurf wird eine Modellarchitektur spezifiziert, die Werkzeugarchitekturen als Hierarchien miteinander interagierender Werkzeugkomponenten modelliert. Zum Vergleich mit anderen komponentenorientierten Ansätzen wird die Modellarchitektur dem C2-Architekturstil gegenübergestellt.

In Kapitel 5 wird gezeigt, wie komponentenorientiert entworfene Softwarewerkzeuge in der Programmiersprache Java implementiert werden können. Es werden grundsätzliche Aspekte der Implementierung in Java betrachtet und verschiedene Konstruktionsvarianten der Werkzeugkomponentenimplementierung vorgestellt und diskutiert.

Zum Abschluß faßt Kapitel 6 die Ergebnisse dieser Arbeit zusammen und zieht ein Fazit über die komponentenorientierte Werkzeugkonstruktion.

2 Komponentenorientierte Softwareentwicklung

Hinter dem Begriff komponentenbasierte Softwareentwicklung (component-based development) verbirgt sich ein Softwareentwicklungsansatz, bei dem Softwaresysteme aus genormten Einzelkomponenten baukastenartig zusammengesetzt werden. Idealer Weise müssen die einzelnen Komponenten nicht selbst entwickelt werden, sondern liegen bereits vor oder können von Drittherstellern erworben werden.

Komponentenorientierte Ansätze können auf verschiedenen Abstraktionsebenen sinnvoll bei der Entwicklung von Softwaresystemen eingesetzt werden. Shaw und Garlan unterscheiden bei der Softwareentwicklung drei Ebenen, auf denen sich ein Softwaresystem in Form von miteinander interagierenden Komponenten beschreiben lassen. Jede dieser Ebenen betrachtet ein System auf einer anderen Abstraktionsstufe und betrachtet andere Komponentenstrukturen. Die Ebenen im einzelnen sind (vgl. [SG96], S.4):

(1) Architektur

(2) Programmcode

(3) Ausführungsmodul

Die Art der Komponenten ist auf den verschiedenen Ebenen sehr unterschiedlich. Die unterste Ebene (3) behandelt Fragen, die das Speicherabbild eines Programmmoduls, Registerbelegungen, Aufbau des Call-Stack beim Aufruf von Programmroutinen und anderen „Low-Level“-Aspekten betreffen. Komponenten dieser Ebene sind entsprechend Binärcodekomponenten. Die Programmcodeebene (2) betrachtet dagegen die Struktur eines einzelnen Programms. Komponenten dieser Ebene sind die primitiven Sprachelemente der verwendeten Programmiersprache.

In der Vergangenheit setzten sich Softwareentwickler vor allem mit diesen zwei unteren Ebenen auseinander. Komponenten wurden im wesentlichen als ausführbare Codemodule angesehen. Auch in aktuelleren Arbeiten wird Komponentenorientierung teilweise lediglich als eine besondere Implementierungstechnik verstanden (vgl. z.B. [Griffel98], S. 33). Wie Shaw und Garlan in [SG96] zeigen, können komponentenorientierte Techniken aber auch sinnvoll auf der abstrakten Softwarearchitekturebene (3) eingesetzt werden. Gerade komplexe und anspruchsvolle Softwaresysteme lassen sich durch eine komponentenorientierte Modellierung einfacher und übersichtlicher handhaben. Wie DeRemer und Kron schon in den 70er Jahren feststellten, ist bei der *Programmierung-im-Großen* die geeignete Modularisierung eines Softwaresystems sowie die Betrachtung der Gesamtstruktur der miteinander interagierenden Module von essentieller Bedeutung [DK76]. Komponentenorientierte Softwarearchitekturen abstrahieren vom konkreten Programmcode und ermöglichen damit eine klare Trennung von architekturspezifischen und implementierungsspezifischen Aspekten.

2.1 Komponentenbegriff

Der Begriff der Komponente wird in der Literatur in vielfältiger Weise verwendet. Das Verständnis darüber, was eine Komponente ist, variiert jedoch stark. So ist der Begriff in einigen Definitionen sehr eng an Aspekte der Implementierung und Code-Wiederverwendung gebunden. Die Unified Modeling Language (UML) definiert eine Komponente beispielsweise folgendermaßen:

> *"A physical, replaceable part of a system that packages implementation and conforms to and provides the realization of a set of interfaces." ([RJB99], S. 216)*

In UML ist eine Komponente eine physische Einheit eines Computersystems, in der Softwarecode gebündelt wird. Der Begriff Code ist hier sehr allgemein zu verstehen und umfaßt Sourcecode, Binärcode, ausführbare Programmodule, Scripte und Befehlsdateien. Typische Komponenten in UML sind Bibliotheken, wie DLL‘s (Dynamic Link Library) und JAR’s (Java Archive) oder auch einzelne Dateien, wie beispielsweise Java Class-Files. In UML wird darüber hinaus zwischen zustandslosen Komponenten und Komponenten mit eigenem Zustand und eigener Identität (Identity Component) unterschieden.

Auch Nierstrasz und Dami haben eine vor allem technisch ausgerichtete Komponentensicht. Sie fassen den Komponentenbegriff jedoch allgemeiner. Konzeptionell ist für Nierstrasz und Dami eine Komponente eine Abstraktionseinheit, die bewußt konzipiert wurde um mit anderen Komponenten zusammenzuarbeiten. Sie definieren eine Komponente wie folgt:

> *"In short, we say a component is a „static abstraction with plugs“. By „static“, we mean that a software component is a long-lived entity that can be stored in a software base, independently of the application in which it has been used. By „abstraction“, we mean that a component puts a more or less opaque boundary around the software it encapsulates. „With plugs“ means that there are well-defined ways to interact and communicate with the component (parameters, ports, messages, etc.)" ([NT95], S. 5)*

In der Komponentendefinition des Catalysis-Ansatz führen D'Souza und Wills an, daß Komponenten unabhängige Einheiten darstellen und daß nicht nur die Schnittstellen explizit spezifiziert sind, über die eine Komponente ihre eigenen Dienste anbietet, sondern auch die Schnittstellen zu Diensten anderer Komponenten, von denen die Komponente selbst abhängig ist:

> *"A coherent package of software implementation that (a) can be independently developed and delivered, (b) has explicit and well-specified interfaces for the service it provides, (c) has explicit and well-specified interfaces for services it expects from others, and (d) can be composed with other components, perhaps customizing some of their properties, without modifying the components themselves." (DW98], S. 387)*

Ein weiterer Aspekt in dieser Definition ist, daß Komponenten in Grenzen konfigurierbar sind, ohne daß die Komponente selbst hierfür modifiziert werden muß.

Shaw und Garlan fassen den Komponentenbegriff weiter. Sie begrenzen den Komponentenbegriff nicht auf Code-Komponenten. Shaw und Garlan verwenden Komponenten als konzeptionelle Modellelemente unabhängig von der Art ihrer Implementierung.

> *"Components are the loci of computation and state. Each component has an interface specification that defines its properties, which include the component's type or subtype (e.g., filter, process, server, data storage), functionality, guarantees about global invariants, performance characteristics, and so on. The specific named entities visible in a component's interface are its players. The interface includes the signature, functionality, and interaction properties of its players." ([SG96], S. 185)*

Allgemeiner Konsens herrscht darüber, daß eine Komponente eine (wieder)verwendbare Einheit eines Softwaresystems darstellt, die ihre Implementierungsdetails verbirgt und mit ihrer Umgebung ausschließlich über explizite Schnittstellen kommuniziert. [Bäumer98] betont eine weitere Eigenschaft von Komponenten: neben ihrer technischen Interpretation besitzt eine Komponente immer auch eine fachliche. In einem anwendungsfachlichen Kontext erbringt eine Komponente eine definierte fachliche Dienstleistung.

Im Rahmen dieser Arbeit soll der Komponentenbegriff ebenfalls nicht auf Code-Komponenten beschränkt bleiben. Komponenten werden als Grundbausteine zur Modellierung von Systemarchitekturen eingesetzt, die eine von Implementierungsdetails abstrahierende Sicht auf komplexe Softwaresysteme fördern. Im weiteren Kontext dieser Diplomarbeit soll eine Komponente wie folgt verstanden werden:

Komponente

Eine Komponente ist eine benannte, (wieder)verwendbare Einheit des Architekturmodells, die innerhalb eines Softwaresystems definierte fachliche Dienstleistungen erbringt. Eine Komponente besitzt eine Menge explizit definierter und benannter Schnittstellen, über die sie mit ihrer Umwelt interagiert und hinter denen Details der Komponentenimplementation verborgen bleiben. Es lassen sich zwei Schnittstellenarten unterscheiden: Schnittstellen, über die eine Komponente ihre Dienstleistungen anderen zur Verfügung stellt (Call-In-Schnittstellen) und Schnittstellen, die Abhängigkeiten zu externen Diensten, auf welche die Komponente selbst angewiesen ist, definieren (Call-Out-Schnittstellen). Eine Komponente besitzt innerhalb eines Systems immer eine eindeutige Identität. Jede Komponente ist Exemplar genau einer Komponentenklasse.

In Verbindung mit einem objektorientierten Implementierungsmodell stellt eine Komponente im allgemeinen eine grobkörnigere Einheit dar als ein Objekt.

Die Unterscheidung der Komponentenschnittstellen in Call-In- und Call-Out-Schnittstellen darf nicht mit der Festlegung der Informationsflußrichtung verwechselt werden. Zu den Dienstleistungen, die eine Komponente über Call-In-Schnittstellen anbietet, gehören neben verändernden Aktionen auch das Infor-

mieren über den Komponentenzustand und aufgetretene Ereignisse. Das bedeutet, Ereignisse (Events) sind prinzipiell über Call-In- und nicht etwa über Call-Out-Schnittstellen zugänglich, ungeachtet der Tatsache, daß der Informationsfluß bei der Ereignisbenachrichtigung aus der Komponente heraus verläuft.[2]

Viele, der in der Literatur anzutreffenden Komponentendefinitionen unterscheiden nicht im einzelnen zwischen konkreten Exemplaren einer Komponentenart und der allgemeinen Beschreibung der gesamten Klasse. Teilweise wird neben dem Begriff *Komponente* der Begriff *Komponentenexemplar* (compontent instance) verwendet, um eine entsprechende Unterscheidung zu ermöglichen (z.B. in UML, [RJB99]).

In dieser Diplomarbeit soll klar zwischen der Beschreibung von Komponenten und den Komponenten selbst unterschieden werden. Genauso wie ein Objekt in einem objektorientierten Modell das Exemplar einer Klasse ist, wird hier eine konkrete Komponente immer als das Exemplar einer Komponentenklasse angesehen. Mit einem objektorientierten Implementierungsmodell entspricht eine Komponentenklasse jedoch nicht direkt einer programmiersprachlicher Klasse. In der Regel wird eine Komponentenklasse nicht nur auf eine einzelne programmiersprachlicher Klasse, sondern auf ein ganzes Konglomerat von Klassen abgebildet.

Komponentenklasse

Eine Komponentenklasse ist eine abstrakte Beschreibung der Eigenschaften ihrer Exemplare. Zu diesen Eigenschaften gehört vor allem auch die Definition der Schnittstellen und ihrer Protokolle.

Spezialisierungs-/Generalisierungsbeziehungen zwischen Komponentenklassen können durch Vererbung ausgedrückt werden. Eine Komponentenklasse erbt hierbei von ihrer Oberklasse immer sämtliche Schnittstellen. Sie kann geerbte Schnittstellen erweitern oder zusätzliche neue Schnittstellen einführen.

Komponenten bilden die Basis zur Beschreibung von Softwarearchitekturen. Da Komponenten aber vom konkreten Kontext unabhängige Einheiten darstellen, legt eine Softwarearchitektur noch mehr fest. Shaw und Garlan definieren den Begriff Softwarearchitektur folgendermaßen:

> *"Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among those components. Such a system may in turn be used as a (composite) element in a larger system design." ([SG96], S. 1)*

Diese Definition betont, daß zur Architekturbeschreibung auch die Festlegung der Komponenteninteraktionen sowie Kompositionsregeln der Komponenten gehören. Shaw und Garlan verwenden zur Modellierung der Interaktionsbeziehungen zwischen Komponenten explizite Modellelemente, die sie Konnektoren nennen. Ein Konnektor definiert das (minimale) Interaktionsprotokoll, dem sich die

[2] In der Architekturbeschreibungssprache Darwin [MDEK95] werden die in dieser Diplomarbeit Call-In genannten Schnittstellen mit dem Schlüsselwort *provides* und die Call-Out-Schnittstellen mit dem Schlüsselwort *requires* ausgezeichnet.

beteiligten Komponenten unterwerfen müssen. Im einfachen Fall spezifiziert ein Konnektor lediglich einen schlichten Prozeduraufruf. Konnektoren können aber beliebig komplexer sein und beispielsweise weitere Filter-, Synchronisierungs-, Protokollierungs-, Sicherheits- oder Schnittstellenanpassungsaufgaben übernehmen. Konnektoren sind vor allem konzeptionelle Elemente einer Architektur. Während Komponenten sich im allgemeinen direkt und eindeutig auf Elemente des Implementierungsmodells abbilden lassen, ist das bei Konnektoren nicht zwangsläufig der Fall. Die Implementation eines Konnektors kann sich auf die beteiligten Komponentenimplementationen verteilen und manifestiert sich bspw. in gemeinsam benutzten Variablen oder in Form von Anweisungen an den Linker (vgl. [SG96]).

In anderen Ansätzen, wie sie beispielsweise die Architekturbeschreibungssprachen Darwin [MDK93] und Rapide [LKA+95] vertreten, stellen Konnektoren dagegen keine expliziten Modellelemente der Softwarearchitektur dar. Interaktionsbeziehungen werden hier durch direkte Bindungen zwischen Komponentenschnittstellen modelliert. Werden die Interaktionsprotokolle komplexer, können sie aus den Komponenten herausgezogen und in separaten Interaktionskomponenten gekapselt werden.

Im Rahmen dieser Arbeit wird auf die explizite Modellierung von Konnektoren verzichtet. Die Interaktionsprotokolle werden in den Schnittstellendefinitionen beschrieben und eine Interaktionspfad zwischen Komponenten ergibt sich durch die direkte Bindung einer Call-Out- an eine Call-In-Schnittstelle.

2.1.1 Graphische Notation

Im Bereich der objektorientierten Modellierung hat sich die Unified Modeling Language (UML) zum de facto Standard entwickelt. UML wird daher auch in dieser Arbeit zur Darstellung objektorientierter Modelle verwendet. Die Bedeutung der Symbole und Diagramme ist in [RJB99] und [BRJ99] genau beschrieben.

Zur graphischen Darstellung von Komponenten existiert dagegen bisher keine allgemein anerkannte Notation. In dieser Arbeit wurde deshalb eine eigene Darstellung gewählt, die Elemente aus der Architekturbeschreibungssprache Darwin [MDEK95] mit Elementen von UML verknüpft.

Komponentenklassen sind als rechteckige Kästen mit einem Namen und expliziten Schnittstellensymbolen dargestellt. Konkrete Komponenten verwenden die gleiche Darstellung, haben jedoch einen unterstrichenen Namen.

Ausgefüllte Schnittstellenkästen bezeichnen Komponentenschnittstellen, die eine Komponentenklasse selbst implementiert (Call-In) und offene Schnittstellenkästen symbolisieren externe Dienstleistungen, auf die die Komponente angewiesen ist und die von einer anderen Komponente erbracht werden müssen (Call-Out).

Ein Interaktionspfad zwischen zwei Komponenten ist durch eine Verbindungslinie zwischen einer Call-In- und einer Call-Out-Schnittstelle symbolisiert.

Vererbungsbeziehungen zwischen Komponentenklassen werden mit den in UML üblichen Vererbungspfeilen dargestellt.

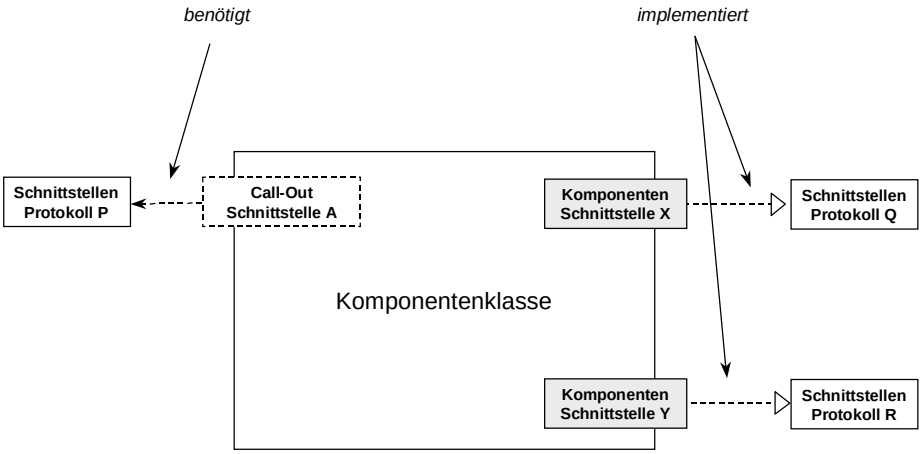

Abbildung 1: Komponentenklasse

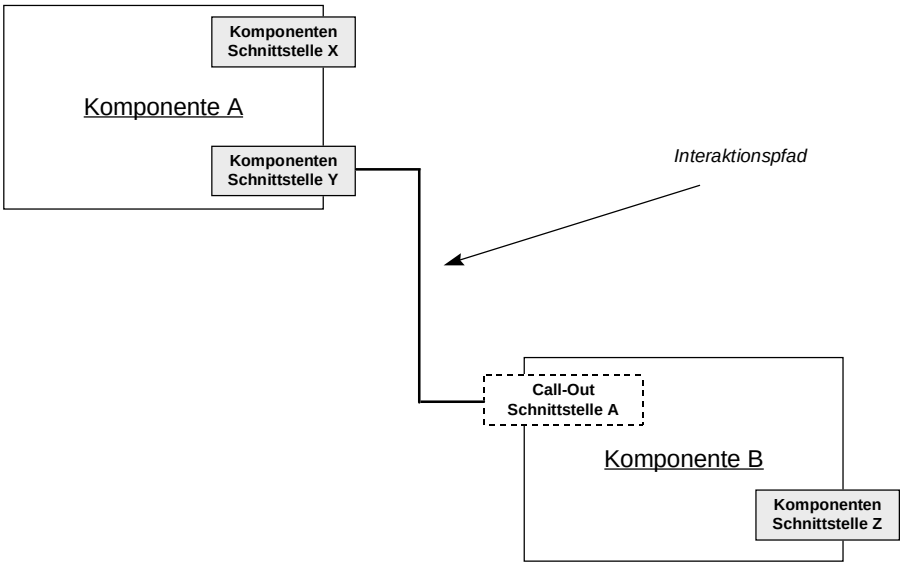

Abbildung 2: Komponenteninteraktion

2.2 Stärken der Komponentenorientierung

Ein Ziel der komponentenorientierten Softwareentwicklung ist die effizientere Herstellung qualitativ hochwertiger Softwaresysteme. Das bekannte Geheimnisprinzip (vgl. [Parnas72]) ist bei Komponenten noch stärker ausgeprägt, als es im objektorientierten Paradigma der Fall ist. Bei Komponenten wird konsequent zwischen Schnittstellen und Implementation getrennt. Durch die Abstraktion von

Implementationsdetails lassen sich so mit einem Komponentenmodell auch komplexe Systemarchitekturen übersichtlich darstellen und Änderungen in der Komponentenimplementation beeinflussen nicht die Architektur des gesamten Systems.

Zwei Aspekte, die oft mit Komponentenorientierung assoziiert werden, sind *Wiederverwendung* und *Plug-and-Play*.

2.2.1 Wiederverwendung

Das Bestreben der Softwareentwicklung, eine einmal entwickelte Lösung wiederholt in mehreren Systemen einzusetzen, ist nicht neu. Schon Ende der 60er Jahre wurde die Wiederverwendung als ein Mittel gegen die sog. Softwarekrise propagiert (vgl. [Krueger92]). Werden nur ausgetestete und robuste Komponenten wiederverwendet, kann das die Softwarequalität eines gesamten Systems erhöhen. Auch konvergieren Komponenten mit zunehmender Wiederverwendung gegen einen immer robusteren und stabileren Zustand.

Der Ruf nach Wiederverwendung ist im wesentlichen betriebswirtschaftlich motiviert. Durch den Rückgriff auf bereits entwickelte Komponenten soll sich der Entwicklungszyklus eines Softwaresystems verkürzen. Bei der Systementwicklung braucht weniger programmiert und weniger getestet zu werden, wodurch der gesamte Entwicklungsaufwand geringer ausfällt. Das gilt insbesondere, wenn Standardkomponenten von externen Herstellern erworben werden können (Off-the-Shelf-Komponenten), die sich in das eigene System integrieren lassen. Auf der anderen Seite relativieren sich die Kosten für einmal selbst entwickelte Komponente, sobald sie mehrfach Verwendung finden.

So wünschenswert die Wiederverwendung von Komponenten unter wirtschaftlichen Gesichtspunkten auch ist, so problematisch erweist sie sich häufig in der Praxis. Um eine bereits existierende Komponente in einem Programmsystem wiederverwenden zu können, muß sie zum einen genau die benötigte Funktionalität besitzen und sich zum anderen harmonisch in das Gesamtsystem integrieren lassen. Gerade die Integration ist oftmals keine triviale Aufgabe und die Probleme betreffen nicht nur Aspekte der systemtechnischen Ebene, sondern wirken sich bis auf die Architekturebene aus (vgl. [GAO95]). Eine Komponente per se ist noch nicht automatisch wiederverwendbar. Sie muß eine große Stabilität aufweisen und bewußt für die Wiederverwendung entworfen worden sein. In der Regel wird eine Komponente erst mehrere Entwicklungszyklen durchlaufen, bis sie einen für eine Wiederverwendung ausreichenden Grad an Stabilität erreicht hat.

Abstraktion spielt eine zentrale Rolle bei der Wiederverwendung (vgl. [Krueger92]). Nur wenn ein softwaretechnisches Artefakt eine klare Abstraktion besitzt, besteht eine Chance, daß es sich in verschiedenen Systemen wiederholt einsetzen läßt. Bei Komponenten liegt diesbezüglich ihre Stärke darin, daß sie die abstrakte Interaktion mit ihrer Umgebung explizit, aber implementierungsunabhängig, in ihren Schnittstellen spezifizieren. Die Schnittstellenbeschreibung einer Komponente umfaßt hierbei nicht nur einen programmiersprachlichen abstrakten Datentyp, sondern beinhaltet immer auch ein Protokoll für die Verwendung der in der Schnittstelle enthaltenen Operationen. Diese explizite Spezifikation der Komponenteninteraktion vereinfacht die Wiederverwendung einer Komponente.

Nicht nur technische Schnittstellenprobleme und Inkompatibilitäten können die Wiederverwendung von Komponenten schwierig gestalten. Gerade unter dem Blickwinkel einer anwendungsorientierten Softwareentwicklung muß abgewogen werden, ob eine vorgefertigte Komponente die Bearbeitung der eigentlichen anwendungsfachlichen Aufgabe ausreichend unterstützt. Damit als Nebenwirkung der Wiederverwendung nicht die Gebrauchsqualität (vgl. [Züllighoven98], S. 125) zu sehr leidet, darf Wiederverwendung nicht ausschließlich unter einem technischen Blickwinkel betrachtet werden.

2.2.2 Plug-and-Play

Die explizite Schnittstellenspezifikation unterstützt eine einfache Komposition von Komponenten. Eine Komponente definiert nicht nur welche Dienste sie ihrer Umwelt anbietet, sondern auch auf welche externen Dienste sie selbst angewiesen ist. Hierdurch sind die Abhängigkeiten zum Einsatzkontext der Komponente klar definiert. Grundsätzlich gilt, daß das Plug-and-Play einer Komponente um so einfacher wird, je weniger Abhängigkeiten die Komponente zu ihrem Kontext besitzt. Sinnvolle, reale Komponenten werden aber niemals gänzlich unabhängig von ihrem Einsatzkontext sein können. Die Stärke von Komponenten liegt darin, daß diese Abhängigkeiten explizit aufgezeigt werden.

Eine weitere Eigenschaft von Komponenten, die ebenfalls das Plug-and-Play fördert, ist ihre große Eigenständigkeit im Erbringen ihrer Funktionalität. In der Regel kapselt eine Komponente eine Dienstleistung soweit, daß sie diese autark, ohne größere Hilfe anderer Komponenten realisieren kann. Es gibt Komponentenmodelle, wie beispielsweise das in Kapitel 4.4 beschriebene C2-Modell, die ihren Komponenten soviel Eigenständigkeit zuordnen, daß jede Komponente generell in einem separaten Adreß- und Prozeßräumen agieren kann.

2.3 Komponentenorientiert vs. Objektorientiert

Worin unterscheiden sich komponentenorientierte von objektorientierten Ansätzen? Ersetzt die Komponentenorientierung gar die Objektorientierung? Zur Beantwortung dieser Fragen ist es wichtig die verschiedenen Softwareentwurfsebenen klar voneinander zu trennen. Wie bereits oben aufgeführt, läßt sich eine komponentenorientierte Sichtweise auf verschiedenen Ebenen verwenden.

Hauptelemente der objektorientierten Modellierung sind Klasse und Objekt. Wie Züllighoven und andere vorschlagen, sollen Klassen hierbei die fachlichen Konzepte und Begriffe, die hinter den Gegenständen der täglichen Arbeit stehen, modellieren (vgl. [Züllighoven98], S. 30). In einer objektorientierten Programmiersprache stellen Klassen aber auch immer das grundsätzliche, technische Konstrukt der Programmierung dar. Dadurch haben Klassen eine doppelte Rolle: auf der einen Seite werden sie verwendet um allgemeine, abstrakte Konzepte darzustellen, auf der anderen Seite besitzen sie aber konkrete Implementierungen. Mit Klassen müssen nicht nur abstrakte fachliche Aspekte, sondern auch alle konkreten softwaretechnischen Aspekte beschrieben werden. Die enge Beziehung zwischen der objektorientierten Modellierung und der zur Konstruktion verwendeten objektorientierten Programmiersprache erleichtert ohne Zweifel den Übergang vom Modell zur Implementation. In vielen Bereichen erweist sich eine Klasse als Modellierungselement für komplexe Konzepte jedoch als zu feingranular. Sobald zur Modellierung eines fachlichen Konzepts nicht nur eine einzelne Klasse, sondern eine ganze Kollaboration von Klassen erforderlich ist,

verwischt eine rein klassenbasierte Modellierung wichtige Architekturaspekte des Systems, da das fachliche Konzept als Ganzes nicht als separates Modellelement existiert.

Bei der komponentenorientierte Modellierung stellen Komponenten und Komponentenklassen die wesentlichen Modellelemente dar. Mit ihnen ist es möglich, eine weitere Abstraktion oberhalb von Objekten und Klassen zu beschreiben. Eine Komponentenimplementation umfaßt im allgemeinen mehrere Klassen und erlaubt damit auch die explizite Modellierung komplexer fachlicher Konzepte. Wichtig für die Modellierung mit Komponenten ist, daß Komponenten weitgehend implementierungsunabhängig spezifiziert werden. Die Betonung liegt stärker auf den Schnittstellen, über die Komponenten miteinander interagieren, als auf deren Implementierung. In diesem Sinne können Komponentenorientierung und Objektorientierung als zwei sich ergänzende Techniken aufgefaßt werden. Komponentenorientiert entworfene Systeme lassen sich auf harmonische Weise mit objektorientierten Techniken implementieren.

Eine nähere Betrachtung zeigt, daß auf der Implementierungsebene auch bei objektorientierten Techniken größere Einheiten als einzelne Klassen existieren. Rahmenwerke implementieren allgemeine generische Lösungen zur Verwendung in einem bestimmten Kontext. In einem Softwaresystem interagieren hierbei meist mehrere Rahmenwerke in geordneter Weise miteinander um die Gesamtsystemfunktionalität zu erbringen. Die Interaktion zwischen den Rahmenwerken läßt sich beispielsweise mit Entwurfsmustern beschreiben (vgl. [Bäumer98]). Hier werden Parallelen zu den Komponenten deutlich. Auch Komponenten kapseln allgemeine generische Lösungen, die in einem größeren Kontext ihre Wiederverwendung finden. Der wesentliche Unterschied besteht darin, daß eine Komponentenklasse ein Konzept auf der Modellierungsebene beschreibt, während ein Rahmenwerk ein Konzept implementiert. Rahmenwerke sind daher bei einem objektorientierten Implementierungsmodell ein geeignetes Mittel zur Realisierung von Komponentenklassen.

Objektorientierte Techniken sind auch bei der komponentenorientierten Softwareentwicklung nicht überflüssig geworden. Vielmehr lassen sich Komponenten und deren Interaktionen mit den heute zur Verfügung stehenden Softwaretechniken am effektivsten mit objektorientierten Sprachen realisieren.

2.4 Modellarchitekturen

Eine Softwarearchitektur ist die abstrakte Beschreibung der Komponenten eines Softwaresystems sowie deren Interaktionsbeziehungen untereinander. Eine Softwarearchitektur besitzt immer einen Bezug auf eine konkrete Anwendung. Obgleich jedes System verschieden ist, lassen sich doch Gemeinsamkeiten in unterschiedlichen Architekturen einer Klasse ähnlicher Systeme erkennen. So weisen beispielsweise alle Client-Server-Systeme typische Ähnlichkeiten in den verwendeten Komponentenklassen und der Art der Komponenteninteraktion auf - unabhängig davon, ob es sich im einzelnen um ein E-Mail- oder ein Hypertextsystem handelt. Ein anderes Beispiel sind die Strukturen von Softwarewerkzeugen, wie sie in Kapitel 4 genauer behandelt werden.

In Anlehnung an die klassische Gebäudearchitektur bezeichnen Shaw und Garlan die architekturspezifischen Gemeinsamkeiten einer Familie ähnlicher Architekturen als *Architekturstil* [SG96]. Architekturstile sind in zweierlei Hinsicht

nützlich. Zum einen ermöglichen Architekturstile das Beschreiben einer Klasse von Architekturen. Sie listen die typischen architektonischen Elemente auf und beschreiben wie diese Elemente zueinander angeordnet sind. Ein Architekturstil abstrahiert hierbei von den konkreten Elementen einer konkreten Architektur. Zum zweiten hat ein Architekturstil auch einen vorschreibenden Charakter. Ein Stil definiert immer auch Grenzen in Bezug auf die überhaupt verfügbaren Elemente und ihren möglichen Anordnungen. Eine Architektur muß sich den Grenzen ihres Stils unterwerfen. Ein Architekturstil abstrahiert von konkreten Architekturelementen und formalen Aspekten und hilft den Rahmen und die Grenzen, in denen sich eine konkrete Architektur bewegt, explizit sichtbar zu machen. Architekturstile definieren die Bereiche, in denen sich eine konkrete Architektur sinnvoll verwenden oder gerade nicht verwenden läßt. Nach Shaw und Garlan besitzt ein Architekturstil die folgenden vier Eigenschaften (vgl. [SG96], S. 191):

(1) Ein *Vokabular* von Entwurfselementen, d.h. einer Menge von Komponenten- und Konnektortypen, die in diesem Stil verwendet werden.

(2) *Designregeln,* welche die Komposition der Entwurfselemente festlegen. Beispielsweise Regeln, die festlegen, daß Zyklen in einer Pipe-Filter-Architektur nicht zulässig sind.

(3) Eine *Semantischer Interpretation* der Entwurfselemente.

(4) Ein Katalog von *Analysen,* die auf Systemen dieses Stils durchgeführt werden können. Beispielsweise Analysen zum Erkennen von potentiellen Systemverklemmungen (Deadlocks). Eine besondere, aber wichtige Form von Analyse stellen Codegeneratoren dar, die architekturstilspezifischen Code erzeugen können.

Shaw und Garlan haben beispielsweise Architekturstile für Pipe-Filter-Architekturen, ereignisgesteuerte oder schichtenbasierte Architekturen formuliert.

[Züllighoven98] benutzt ein vergleichbares Konzept zur Abstrahierung von konkreten Softwarearchitekturen. Züllighoven verwendet hierfür den Begriff *Modellarchitektur.* Eine Modellarchitektur faßt eine Klasse ähnlicher Softwarearchitekturen zusammen. Eine Modellarchitektur definiert er wie folgt:

> *"Eine Modellarchitektur beschreibt die allgemeinen Prinzipien hinter einer Softwarearchitektur. Sie umfaßt die grundlegenden Elemente, deren Verknüpfungen und die Regeln, die für eine Softwarearchitektur gelten.*
>
> *Eine Modellarchitektur gibt Anleitung bei der softwaretechnischen Realisierung eines Softwaresystems." ([Züllighoven98], S. 325)*

3 Komponentenstandards

Von praktischer Relevanz im Bereich komponentenorientierter Systementwicklung sind heute vor allem zwei Standards: Microsoft's COM-Modell und Sun's Java Beans-Modell. In diesem Kapitel werden die Kernkonzepte der beiden Ansätze vorgestellt und dem in dieser Diplomarbeit zugrundegelegten Komponentenverständnis gegenübergestellt.

3.1 COM

Begründet auf der Verbunddokumenten-Technik OLE (Object Linking and Embedding) veröffentlichte Microsoft 1993 eine allgemeine technische Basis für komponentenbasierte Systemarchitekturen. COM (Componenten Object Model) ist ein Komponentenstandard, der die Interaktion zwischen separat entwickelten Komponenten definiert und der die Grundlage für alle komponentenorientierten Systeme aus dem Hause Microsoft bildet. Ursprünglich war der Einsatz von COM ausschließlich auf Windows-Plattformen beschränkt. Mittlerweile existieren aber auch erste Portierungen in die Macintosh- und in die Unix-Welt.

Das wesentliche Element im COM-Modell ist die Schnittstelle (Interface). Eine COM-Schnittstelle besteht aus einer Menge von Operationen, über die eine Komponente ihre Dienste anbietet. Im COM-Modell kann eine Komponente beliebig viele Schnittstellen besitzen. COM ist ein binärer Standard, d.h. es ist genau spezifiziert, wie sich eine COM-Komponente im Hauptspeicher darstellt und wie ein Klient über verkettete Zeiger (Pointer) auf die Schnittstellenoperationen zugreifen kann. (Abbildung 3). Der COM-Standard legt dagegen jedoch nicht fest, wie eine Komponente auf eine konkrete Programmiersprache abgebildet werden muß.

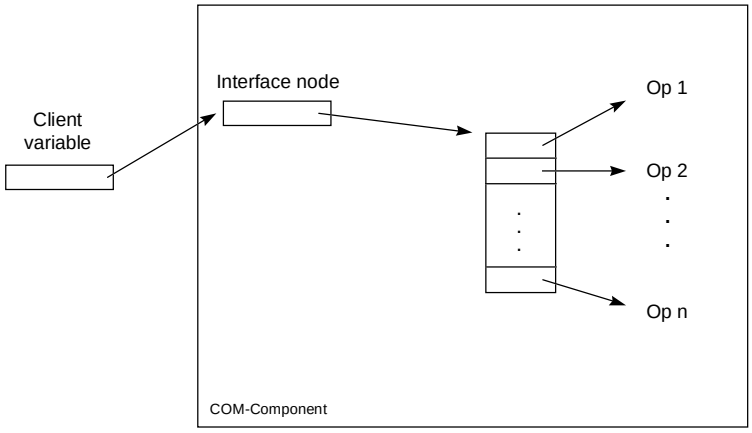

Abbildung 3: Binäre Repräsentation einer COM-Komponente

Damit er mit einer COM-Komponente arbeiten kann, muß ein Klient einen Verweis auf einen Schnittstellenknoten besitzen, der selbst wiederum einen Zeiger

auf eine Tabelle von Zeigern auf Operationen (function pointers) enthält. Die Zeigertabellen sind vergleichbar mit den aus C++ Implementationen bekannten *vtables*[3] (vitual tables). Über diese Schnittstellenzeiger können die Operationen einer Komponente verwendet werden.

Die programmiersprachliche Implementation einer COM-Komponente ist nicht weiter festgelegt. In einer objektorientierten Sprache kann die gesamte Implementation in einer einzelnen Klasse erfolgen - sie muß es jedoch nicht. Es können auch mehrere Klassen zur Implementierung der verschiedenen COM-Schnittstellen eingesetzt werden, wobei erst mehrere Objekte zusammen ein COM-Objekt ausmachen. Der COM-Standard enthält keine Spezifikation wie einzelne Programmiersprachen an das COM-Modell gebunden werden und definiert daher diesbezüglich auch keinerlei Einschränkungen. Ein wichtige Folge hieraus ist, daß es keine einzelne Objektreferenz gibt, die ein COM-Objekt als Ganzes repräsentiert. Ein Klient kann deshalb niemals direkt ein COM-Objekt referenzieren, sondern immer nur eine einzelne COM-Schnittstelle.

Es gibt eine ausgezeichnete Schnittstelle, `IUnknown`, die jede COM-Komponente besitzt. Diese Schnittstelle bildet einen generischen, allgemeinen Zugang, über den ein Klient Zugriff auf die übrigen Schnittstellen erlangen kann. Hierfür besitzt `IUnknown` die Funktion `QueryInterface()`. Jede COM-Schnittstelle besitzt einen 128 Bit großen, systemweit eindeutigen Identifikator (IID: Interface Identifier), mit dem `QueryInterface()` aufgerufen werden kann, um einen Zeiger auf die Schnittstelle zu erhalten. Darüber hinaus besitzt `IUnknown` zur Verwaltung eines Referenzzählers noch die zwei Operationen `AddRef()` und `Release()`. Das COM-Modell enthält keinen separaten Garbage Collector-Mechanismus, sondern realisiert statt dessen ein Reference Counting-Verfahren zur Freigabe nicht mehr benötigter Ressourcen. Jedesmal, wenn eine Schnittstellenreferenz weitergegeben wird, muß `AddRef()` aufgerufen werden und sobald eine Schnittstellenreferenz nicht mehr benötigt wird, muß ein `Release()`-Aufruf erfolgen. Sobald der Referenzzähler eines COM-Objekts auf null steht, wird dieses mit allen seinen belegten Ressourcen wieder freigegeben.[4]

Jede COM-Schnittstelle muß neben ihren eigenen Operationen immer auch die drei Operationen der `IUnknown`-Schnittstelle enthalten. Das COM-Modell schreibt hierbei vor, daß ein Aufruf von QueryInterface() mit der IID von `IUnknown` bei allen Schnittstellen immer dieselbe Refernz liefern muß. Die `IUnknown`-Refernz kann damit zur Identifizierung eines COM-Objekts dienen.

Jedes COM-Objekt ist Exemplar einer COM-Klasse. COM-Schnittstellen und COM-Klassen können sprachunabhängig in den speziellen Beschreibungssprachen IDL[5] (Interface Definition Language) und ODL (Object Description Language) definiert werden. COM-Klassen besitzen einen systemweit eindeutigen Identifikator (CLSID: class identifier) unter dem sie im System registriert sind.

---

[3] In der COM-Implementation von Microsofts Programmiersprache Visual C++ sind diese Zeigertabellen tatsächlich identisch mit den vtables.

[4] Der Name `IUnknown` ist eigentlich etwas irreführend: `IUnknown` ist gerade keine "unbekannte" Schnittstelle, sondern die einzige Schnittstelle, die sicher bei jeder COM-Komponente zu finden ist.

[5] Obgleich des Namens ist COM-IDL nicht identisch mit der CORBA-IDL (vgl. [OMG95])

COM-Objekte registrierter COM-Klassen können über die COM-Operation `CoCreateInstance()`[6] unter Angabe der CLSID und der gewünschten Schnittstellen-IID erzeugt werden. Zugänglich wird eine COM-Klasse dem System über einen sog. COM-Server (Abbildung 4). Ein COM-Server beinhaltet eine oder mehrere COM-Klassen sowie für jede Klasse ein eigenes Fabrikobjekt zur Erzeugung der COM-Objekte (vgl. Factory-Muster in [GHJV95]). Das Fabrikobjekt implementiert hierbei die Schnittstelle `IClassFactory`[7] oder die um einen Lizensmechanismus erweiterte Version `IClassFactory2.` COM kennt drei verschiedene Servertypen: *local servers*, *in-process servers* und *remote servers*. Lokale Server liegen als separate, ausführbare Programmmodule (EXE-Datei) vor. In-Process Server sind dynamisch ladbare Bibliotheken (DLL: dynamic link library), die erst zur Laufzeit in den Prozeßraum des Klienten geladen werden und Remote Server starten eine Komponente auf einer entfernten Maschine.

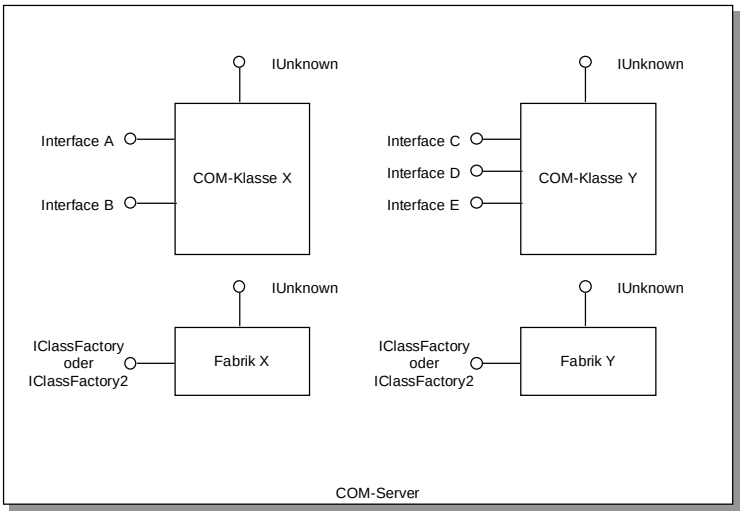

Abbildung 4: COM-Server mit zwei Klassen samt Fabriken

COM kennt keinen Vererbungsmechanismus zwischen Komponentenklassen. Einzig bei den Schnittstellen wird Vererbung unterstützt. Beispielsweise erben alle Schnittstellen von `IUnknown`. Natürlich kann bei der Implementierung von COM-Komponenten in einer objektorientierten Programmiersprache komponentenintern auch Vererbung eingesetzt werden, da die Implementierung von Komponenten in COM nicht festgelegt ist. Komponentenübergreifend ist jedoch keine Vererbung möglich. Wiederverwendung von Komponentenimplementationen wird im COM-Modell durch zwei andere Mittel erreicht: Containment und Aggregation. Hierbei handelt es sich um Formen von Kompositionsmechanismen, die darauf basieren, daß eine äußere Komponente eine innere umschließt und die äußere Komponente auch Dienste, d.h. Schnittstellen, der inneren anbietet. Nur die äußere Komponente ist für Klienten zugreifbar. Abbildung 5 zeigt wie sich eine Komposition per Containment darstellt.

[6] Alle Operationen der COM-Bibliothek besitzen den Präfix Co.

[7] Auch der Name IClassFactory ist etwas irreführend: Die Fabrik erzeugt keine Klassen, sondern Exemplare einer COM-Klasse.

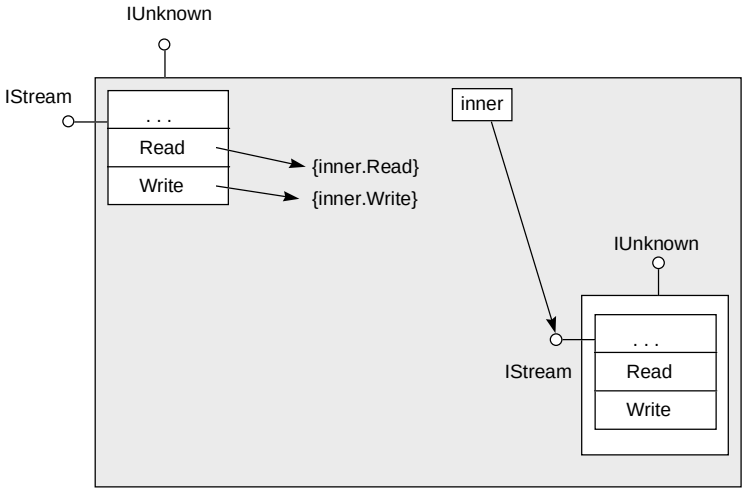

Abbildung 5: Containment-Beziehung

In der Containment-Beziehung implementiert die äußere Komponente die Schnittstelle `IStream` nicht selbst, sondern delegiert die Operationsaufrufe an die innere Komponente weiter. Ein Klient erhält jedoch immer nur eine Referenz auf die `IStream`-Schnittstelle der äußeren Komponente.

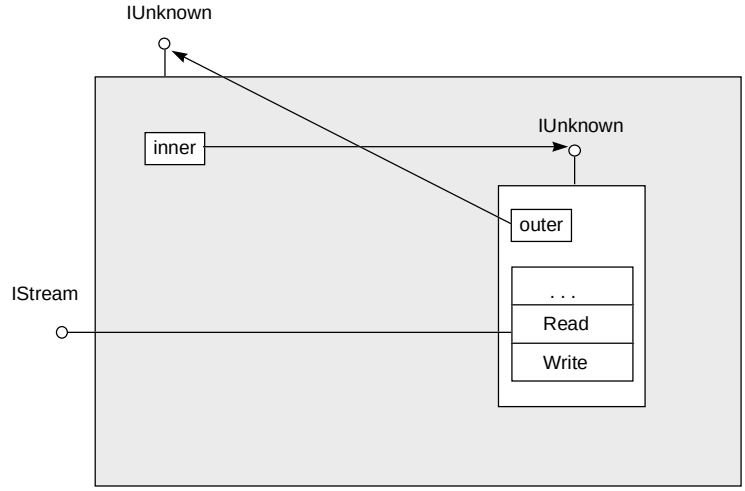

Abbildung 6: Aggregation-Beziehung

Bei einer Aggregation-Beziehung (Abbildung 6) liefert ein `QueryInterface()`-Aufruf mit der IID von `IStream` dagegen direkt eine Schnittstellenreferenz auf die entsprechende Schnittstelle des inneren Objekts. Hierdurch ist keine Delegation der Operationsaufrufe nötig und die Lösung ist etwas performanter. Realisiert wird die Aggregation-Beziehung dadurch, daß `QueryInterface()` für alle Schnittstellen, die die Komponente nicht selbst implementiert, den `QueryInterface()`-Aufruf an die inneren Komponente weiterreicht. Wie in der Abbildung zu sehen ist, besitzt auch die innere Komponente einen Verweis auf die äußere Komponente, damit die COM-Regel eingehalten werden kann, daß ein `QueryInterface()`-Aufruf mit der IID von

`IUnknown` an jeder Schnittstelle immer dieselbe Referenz, also immer IUnknown der äußeren Komponente, zurückliefert.

Neben den Schnittstellen über die ein COM-Objekt Dienste anbietet, die von anderen Objekten aufgerufen werden können, kann ein COM-Objekt spezielle Schnittstellen spezifizieren, über die sich eine Form von Rückrufmechanismus zu verbundenen Objekten etablieren läßt. Im COM-Sprachgebrauch werden diese Schnittstellen *Outgoing Interfaces* genannt. Sie sind vom Typ `IConnection-Point` und bietet im wesentlichen Operationen zum Registrieren von Schnittstellen anderer Komponenten, die von dem COM-Objekt gerufen werden sollen. Über Outgoing Interfaces werden in COM im allgemeinen Ereignismechanismen zwischen Komponenten realisiert.

Als eine Weiterentwicklung der sprachabhängigen *Visual Basic Controls* (VBX) und als Alternative zum Applet-Konzept von Java hat Microsoft 1996 *ActiveX* veröffentlicht. ActiveX basiert auf dem COM-Fundament. Bei einer ActiveX-Komponente handelt es sich im wesentlichen um ein COM-Objekt, das spezielle Schnittstellen unterstützt und das auf einem besonderen Server implementiert ist. ActiveX-Server besitzen die Fähigkeit sich selbständig beim COM-System zu registrieren. Hierdurch können ActiveX-Komponenten dynamisch, beispielsweise von einem Web-Browser, über ein Netzwerk geladen und in ein laufendes System integriert werden.

Im Oktober 1997 veröffentlichte Microsoft COM+ eine Erweiterung des COM-Standards. COM+ ist ein auf lokale In-Process Server beschränktes Modell, das besonders leichtgewichtige Komponenten unterstützt. COM+ läßt sich harmonischer auf moderne objektorientierte Sprachen wie C++ oder Java abbilden, als es mit COM der Fall ist. Die Programmierung von COM+ Komponenten gestaltet sich dadurch sehr viel einfacher. Die Implementierungssprache wird um einige COM+ spezifische Schlüsselwörter ergänzt, die beim Übersetzungsvorgang bzw. bei der Codeinterpretation von entsprechenden COM+ Werkzeugen ausgewertet werden. Eine explizite Komponentendefinition mit der COM-IDL (Interface Definition Language) ist nicht nötig.

Im COM+ Modell fällt dem Klassenkonzept eine stärkere Bedeutung zu. Methoden können direkt in einer COM+ Klasse definiert werden und nicht nur ausschließlich in den Schnittstellen. Hierdurch kann nun auch Implementationsvererbung zwischen COM+ Klassen verwendet werden. Außerdem ist es durch das explizite Klassenkonzept nicht mehr unbedingt erforderlich, daß ein Server zu jeder Klasse immer auch eine passende Fabrik enthält. COM+ Objekte können vom System direkt aus den COM+ Klassen instanziiert werden.

Während Metainformationen in COM in separaten Typbibliotheken (type libraries) abgelegt sind, bietet COM+ einen direkten Zugang zu den Metainformationen einer Komponente. Dadurch bieten COM+ Komponenten eine größere Flexibilität und eine leichtere dynamische Anpaßbarkeit an den aktuellen Einsatzkontext. Da COM+ auf einen einzelnen Prozeß beschränkt ist, wird außerdem ein Garbage Collection ohne den Mechanismus des Reference Counting unterstützt.

3.2 Java Beans

Ende 1996 erweitertet Sun Microsystems Java um ein eigenes Komponentenmodell, mit Namen Java Beans. Der Komponentenbegriff im Java Beans-Modell

ist recht weitgefaßt. Er reicht von einfachen graphischen GUI-Komponenten, wie beispielsweise einer Schaltfläche (Button), bis hin zur hoch komplexen, anwendungsähnlichen Komponenten, wie einer kompletten Textverarbeitungskomponente. Die Java Beans-Spezifikation definiert eine Java Bean folgendermaßen:

> *"A Java Bean is a reusable software component that can be manipulated visually in a builder tool." ([JavaSoft96], S. 9)*

Das Komponentenkonzept ist in Java nicht durch eine allgemeine Wurzelklasse realisiert, von der alle konkreten Komponenten erben müßten, sondern manifestiert sich durch eine Reihe von Namens- und Verhaltenskonventionen, Konstruktionsprinzipien sowie beschreibenden Metaklassen. Da Java nur die Einfachvererbung unterstützt, hätte eine Bean-Wurzelklasse auch eine zu große Einschränkung für die Gestaltung von Klassenhierarchien bedeutet. Die wesentlichen charakteristischen Aspekte des Java Beans-Modell sind folgende:

- *Methoden, Ereignisse, Eigenschaften*: Die Schnittstelle einer Java Bean besteht aus einer Menge von Methoden (Methods), über die die Komponente ihre Dienste anbietet, einer Menge von Ereignissen (Events), die die Komponente aussendet sowie einer Menge von einstellbaren Komponenteneigenschaften (Properties).

  Komponentenmethoden sind öffentliche (public) Funktionen und Prozeduren, die eine Komponente ihrem Umfeld zur Verfügung stellt. Generell können dieses alle öffentlichen Operationen sein, die eine Java Beans-Klasse besitzt. Häufig wird die Menge der Komponentenmethoden jedoch lediglich eine Teilmenge aller öffentlichen Operationen sein, wenn nicht alle Operationen gleichermaßen für die Komponenteninteraktion konzipiert sind.

  Ereignisse stellen den Mechanismus dar, wie eine Komponente ihren Kontext über Zustandsänderungen informiert. Der Java Ereignismechanismus (AWT Event Model) ist hierbei so aufgebaut, daß die ereignisaussendende Komponente (event source) an ihrer Schnittstelle Ereignistyp-spezifische Registrierungsoperationen anbietet, unter denen sich ein Beobachter (Listener) mit entsprechenden Objekten registrieren lassen kann. An einem Ereignistyp können sich beliebig viele Beobachter registrieren lassen. Die Schnittstelle eines Beobachters ist jeweils in einem eigenen EventListener-Interface spezifiziert und die ereignisaussendende Komponente kennt ihre Beobachter ausschließlich unter diesem Interface. Hierdurch ist eine lose Kopplung der aussendenden Komponente zu ihren Beobachtern gegeben.

  Eigenschaften sind einstellbare, benannte Komponentenattribute, wie beispielsweise Farbe oder Schriftart der Oberflächenpräsentation. Eine Eigenschaft wird über generische get/set-Operationen gelesen und verändert. Komponenteneigenschaften wirken sich direkt auf die Darstellung und das Verhalten einer Komponente aus. In den meisten Fällen informiert eine Java Bean ihren Kontentext über die Änderung einer Eigenschaft durch das Aussenden spezieller Ereignisse (`PropertyChangeEvent`).

- *Entwurfszeit vs. Laufzeit*: Die Möglichkeit der werkzeugunterstützten Konfiguration von Java Beans ist fester Bestandteil des Komponentenkonzepts. Damit eine Komponente von einem Werkzeug bearbeitet werden kann, muß sie in der Lage sein, zwischen Entwurf und Einsatz zu unterscheiden. In der Entwurfsumgebung ermöglicht eine Java Bean die Einstellung ihrer Eigen-

schaften durch ein entsprechendes Werkzeug (builder tool). In der Laufzeitumgebung verhält sich die Komponente dann entsprechend ihrer Einstellungen.

- *Persistenz*: Prinzipiell sollte jede Java Bean mit dem Java Serialisierungsmechanismus serialisierbar sein und damit ermöglichen, ihren Zustand persistent zu speichern und nachher wiederherzustellen. In Verbindung mit einem Builder Tool ist der Serialisierungsmechanismus das technische Mittel um eine konfigurierte Java Bean zu speichern. Serialisierbare Komponenten implementieren die Java-Interfaces `Serializable` oder `Externalizable`. Die Instanziierung einer konfigurierten Komponente in einem laufenden System erfolgt anschließend nicht über den üblichen `new`-Operator, sondern über die statische Methode `java.beans.Beans.instantiate()`.
- *Introspektion*: Vor allem während der Entwurfszeit im Builder Tool, aber auch zur Laufzeit, kann es erforderlich sein herauszufinden, welche Methoden, Ereignisse und Eigenschaften eine Java Bean besitzt. Dieser Vorgang wird als Introspektion bezeichnet. Unter Zuhilfenahme des Package `java.lang.reflection` lassen sich diese Metainformationen innerhalb der Sprache Java selbst ermitteln. Die Basis hierfür bilden spezielle Namenskonventionen[8] denen Java Beans sich unterwerfen müssen. Das Beans-Rahmenwerk beinhaltet Metaklassen, wie `BeanInfo` und `FeatureDescriptor`, über die ein Zugang zu den Metainformationen auf einer höheren Ebene möglich ist.
- *Komponentenhierarchien*: Seit Java 2 (JDK 1.2) enthält das Beans-Rahmenwerk die Möglichkeit Komponenten in logischen Hierarchien anzuordnen. Eine Java Bean, die das Interface `BeanContextChild` erfüllt, läßt sich zur Laufzeit in einen sog. `BeanContext` einbetten. Die hauptsächliche Verwendung dieser Hierarchien liegt darin, daß eine Komponente dynamisch und erst zur Laufzeit verfügbare Dienstleistungen ihres Kontext erfragen und in Anspruch nehmen kann. Eine Java Bean kann auch eigene Dienstleistungen bei ihrem Kontext anmelden und diese damit anderen Komponenten des gleichen Kontext zugänglich machen.

Obgleich es im Java Beans-Modell immer eine ausgezeichnete Klasse gibt, die eine Komponente als Ganzes repräsentiert, besteht eine Java Bean-Komponente in der Regel aus mehreren, zusammenhängenden Klassen. Die GUI-Komponenten des Swing-Rahmenwerks sind beispielsweise nach einer Variation des MVC-Musters (vgl. [BMR+96], [KP88]) aufgebaut und besitzen neben der Komponentenhauptklasse immer auch eine Model- und eine UI-Klasse.

Das Java Beans-Modell hat seinen Ursprung in den GUI-Komponenten des AWT-Rahmenwerks (Abstract Windowing Toolkit) und die meisten der heute verfügbaren Java Beans besitzen eine Repräsentation an der graphischen Benutzungsoberfläche. Per Definition entsprechen alle GUI-Komponenten, die von der Klasse `java.awt.Component` abgeleitet sind dem Java Beans Komponentenbegriff. Das Java Beans-Modell ist aber nicht auf graphische

[8] In der Java Beans-Spezifikation werden diese Namenskonventionen etwas unglücklich als "Design Patterns" bezeichnet. Der Begriff ist sehr irreführend, da mit "Design Patterns" allgemein eher Entwurfsmustern nach [GHJV95] assoziiert werden. Mit diesen haben die Java Beans Design Patterns aber nicht das geringste zu tun.

Komponenten beschränkt. Prinzipiell können auch *unsichtbare* Komponenten der Java Beans-Spezifikation genügen.

Im März 1998 ergänzte Sun Java um einen weiteren Komponentenstandard, *Enterprise Java Beans*, kurz EJB [JavaSoft98]. Enterprise Java Beans können als eine Erweiterung des Java Beans-Modells angesehen werden, in dem Sinne, daß auch Enterprise Java Beans mit den oben aufgeführten Merkmalen, wie Methoden, Ereignisse, Persistenz, etc., ausgestattet sind. EJBs sind jedoch eine ganz spezielle Erweiterung des allgemeinen Java Beans-Standards. Java Beans und Enterprise Java Beans kommen in sehr unterschiedlichen Kontexten zum Einsatz. Auch wenn es der allgemeine Java Beans-Standard nicht zwingend vorschreibt, so werden Java Beans doch gemeinhin mit klientenseitigen Komponenten assoziiert. Enterprise Java Beans kommen dagegen ausschließlich in mehrschichtigen, verteilten, transaktionsorientierten Anwendungssystemen auf der Server-Seite zum Einsatz. EJBs sind serverseitige Komponenten, auf die ein Klient entfernt zugreifen kann, ohne daß dieser sich mit den systemtechnischen Verteilungsaspekten beschäftigen muß. Eine EJB-Komponente kapselt entweder eine klientenspezifische Transaktion (session bean) oder eine auf einem persistenten Massenspeicher ausgelagerte langlebige Entität (entity bean), beispielsweise einen Kundensatz in einer Datenbank.

Der Fokus dieser Diplomarbeit liegt auf Komponentenmodellen zur Modellierung und Konstruktion von Softwarewerkzeugen. Die Kernaspekte von Enterprise Java Beans, wie Verteilung, Transaktionssteuerung sowie Massendaten bzw. Massenkomponenten spielen hierfür keine Rolle. Es wird daher nicht weiter im Detail auf das Enterprise Java Beans-Modell eingegangen.

3.3 Abgenzung der Konzepte

Der Komponentenbegriff in COM und Java Beans ist in erster Linie sehr implementierungsbezogen. Das COM-Modell ist eng an die MS Windows-Betriebssystemfamilie gekoppelt und Java Beans ist ein Komponentenstandard, der sich nicht von der Programmiersprache Java trennen läßt. In diesem Sinne betrachten die beiden Modelle Komponenten nicht auf der abstrakten Architekturebene, sondern auf der konkreteren Implementierungsebene. Die Konzepte hinter den konkreten Modellen besitzen aber Gemeinsamkeiten mit dem im vorangegangenen Kapitel definierten Komponentenverständnis.

Im COM-Modell liegt, sehr viel stärker als im Java Beans-Modell, die Betonung auf den Komponentenschnittstellen.[9] Erst das explizite Schnittstellenkonzept ermöglicht eine weitreichende Abstraktion von der konkreten Komponentenimplementierung. In diesem Punkt deckt sich die COM-Sicht einer Komponente mit dem in dieser Diplomarbeit verwendeten Komponentenverständnis.

Weder COM noch Java Beans spezifizieren jedoch explizit die Kontextabhängigkeiten einer Komponente. Keines der Modelle besitzt ein zu den Call-Out-Schnittstellen vergleichbares Konzept. Die Outgoing Interfaces im COM-Modell beschreiben zwar explizit, daß ein Informationsfluß aus der Komponente heraus

[9] Zu den Stärken der Programmiersprache Java gehört ihr Interface-Konzept. Es ist daher verwunderlich und oft kritisiert worden, daß im Java Beans-Modell selbst so wenig Gebrauch von diesem Konzept gemacht wurde. Im Gegensatz zu Java Beans werden Java-Interfaces im Enterprise Java Beans-Modell dagegen sehr intensiv genutzt.

existiert, jedoch bedeutet ein Outgoing Interface nicht, daß die Komponente in irgendeiner Weise davon abhängig ist, ob eine Komponente an dieser Schnittstelle angeschlossen ist. Outgoing Interfaces entsprechen im wesentlichen dem Ereignismechanismus im Java Beans-Modell.

Wie auch in dieser Diplomarbeit, wird im Java Beans-Modell eindeutig zwischen einer Komponentenklasse und Komponentenexemplaren differenziert. Wie alle objektorientierten Klassen, können auch Java Beans-Klassen durch Vererbung spezialisiert werden. Auch im COM-Modell gibt es ein Klassenkonzept jedoch ist hier keine Vererbung zwischen COM-Klassen möglich. Erst in der Erweiterung COM+ ist ein Vererbungsmechanismus vorhanden. In dieser Diplomarbeit wird die Auffassung vertreten, daß auch bei der komponentenorientierten Systementwicklung Vererbung sinnvoll zur Beschreibung von Ähnlichkeiten zwischen Komponentenklassen sowie zur Unterstützung der Komponentenimplementierung eingesetzt werden kann.

4 Werkzeugarchitekturen

In den vorangegangenen Kapiteln wurde die Komponentenorientierung im allgemeinen betrachtet. In diesem Abschnitt wird nun untersucht, wie sich komponentenorientierte Konzepte im speziellen Umfeld der Entwicklung von Softwarewerkzeugen auswirken.

Softwarewerkzeuge, wie sie im WAM-Ansatz verstanden werden, sind interaktive Anwendungssysteme, die zum Sondieren und Verändern von Materialien verwendet werden. Die Funktionalität eines Softwarewerkzeugs orientiert sich hierbei an den fachlichen Arbeitsaufgaben, die es unterstützen soll. Neben seiner Funktionalität besitzt ein Softwarewerkzeug immer auch eine graphische Repräsentation und bietet an seiner Benutzungsschnittstelle flexible Möglichkeiten der Handhabung an.

In WAM ist das Konzept Werkzeug eine Entwurfsmetapher. Das bedeutet, daß innerhalb des WAM-Ansatzes konkrete Vorstellungen über die Architektur eines Softwarewerkzeugs existieren. Zum Begriff der Entwurfsmetapher schreibt Züllighoven:

> *"Eine Entwurfsmetapher hat im WAM-Ansatz immer auch eine technisch konstruktive Interpretation in Form von Konstruktionsanleitungen und Entwurfsmustern." ([Züllighoven98], S.79)*

Kapitel 4.2 betrachtet die im Rahmen des WAM-Ansatzes verwendeten Entwurfsmuster zur Werkzeugkonstruktion und die daraus resultierenden Werkzeugarchitekturen. Der reguläre Aufbau von Softwarewerkzeugen erlaubt aber auch andere Architekturen, die nicht durch die WAM-Entwurfsmuster abgedeckt sind. In Kapitel 4.3 wird eine komponentenorientierte Modellarchitektur für Softwarewerkzeuge vorgestellt, die im Rahmen dieser Arbeit entwickelt wurde. Im Anschluß daran stellt Kapitel 4.4 eine andere komponentenorientierte Modellarchitektur der Universität Irvine, Kalifornien, vor, die ebenfalls speziell für die Konstruktion interaktiver Anwendungssysteme entwickelt worden ist und die sich C2 nennt.

4.1 Begleitendes Beispiel

In den folgenden Kapiteln wird ein wiederkehrendes Beispiel die Diskussion um Entwurfs- und Konstruktionsansätzen begleiten. Bei dem Beispiel handelt es sich um ein Werkzeug zur Bearbeitung von Terminen. Das Werkzeug soll im folgenden kurz skizziert werden.

Entsprechend der begrifflichen Trennung von Werkzeug und Material können Terminkalender als eine Form von Material verstanden werden. Während in der realen Welt für die Arbeit mit einem Terminkalendern ein einfacher Schreibstift ausreicht, benötigt ein elektronischer Kalender zur Bearbeitung spezielle softwaretechnische Werkzeuge. Erst durch ein Werkzeug wird ein elektronischer Terminkalender für den Anwender überhaupt zugänglich. Ein Kalenderwerkzeug ermöglicht beispielsweise das Blättern in einem Terminkalender, das gezielte Suchen, Hinzufügen, Verändern oder Streichen von Terminen. Abbildung 7 zeigt

wie sich das Kalenderwerkzeug während der Arbeit an der Benutzungsoberfläche präsentiert.

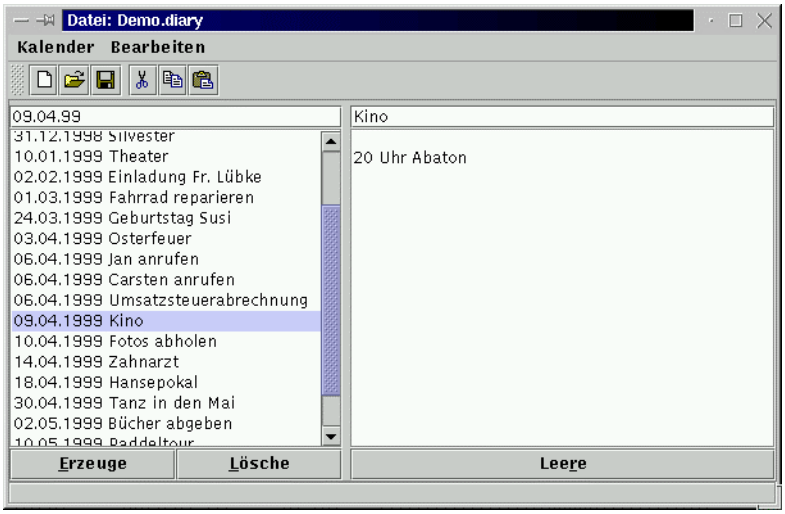

Abbildung 7: Kalenderwerkzeug

Das Werkzeug stellt einen Terminkalender als eine Liste von Terminen dar. Jeder Termin besitzt ein Datum und eine kurze Bezeichnung, die diesen Termin charakterisiert. Neben der Bezeichnung können einem Termin beliebige weitere, ausführliche textuelle Angaben zugeordnet sein.

Das Kalenderwerkzeug verwendet die heute allgemein üblichen Steuerungselemente interaktiver Anwendungssysteme. Ein statisches Menü am oberen Fensterrand sowie eine Symbolleiste ermöglichen den einfachen Zugriff auf die Funktionalität des Werkzeugs. Viele Aktionen sind darüber hinaus direkt über eine Tastenkombination erreichbar. Für eine einfachere Handhabung läßt sich außerdem über die rechte Maustaste ein kontextsensitives Popup-Menü aufrufen, in dem nur die Menüpunkte angeboten werden, die in der augenblicklichen Situation fachlich relevant und sinnvoll sind.

Zur Förderung einer intuitiven Handhabung ist die Benutzungsoberfläche entsprechend dem Prinzip aufgebaut, daß Menüeinträge und Schaltflächen, die in einem speziellen Kontext nicht möglich bzw. fachlich nicht sinnvoll sind, ausgegraut und deaktiviert sind. So ist beispielsweise die *Löschen*-Schaltfläche immer deaktiviert, wenn gerade kein Termin in der Liste ausgewählt worden ist.

Termine des Kalenders können aus der Liste per Maus oder Tastatur ausgewählt und direkt in einem Texteditor auf der rechten Seite des Werkzeugs sowie zwei Eingabefeldern im oberen Teil bearbeitet werden. Änderungen und Neuerfassung von Terminen erfolgen ebenfalls direkt über diese Felder (d.h. kein Eingabedialog mit Bestätigung o.ä.).

Für einen sicheren Umgang mit dem Werkzeug können alle verändernden Aktionen über einen mehrstufigen Undo-Mechanismus nachträglich widerrufen werden.

Es genügt an dieser Stelle, wenn der Leser eine intuitive Vorstellung über die Funktionalität und Handhabung des Kalenderwerkzeugs entwickelt. Das Ziel dieser Arbeit ist nicht die Entwicklung und Beschreibung eines ausgefeilten Werkzeugs zur Terminverwaltung. Das Kalenderwerkzeug dient nur zur Erläuter-

ung der diskutierten softwaretechnischen Entwurfs- und Konstruktionsansätze. Einzelne Werkzeugaktionen und spezielle Handhabungen werden in den folgenden Kapitel an entsprechender Stelle noch genauer beschrieben.

Das Kalenderwerkzeug ist mit seiner Benutzungsoberfläche und Handhabung bewußt an einem bereits in [Riehle95] vorgestellten Beispiel orientiert. Die Wiederaufnahme des Beispiels soll die Vergleichbarkeit der hier vorgestellten Konzepte mit den aus [Riehle95] bekannten *klassischen* Formen der Werkzeugkonstruktion nach WAM fördern.

4.2 Werkzeugentwurf im WAM-Ansatz

Innerhalb des WAM-Ansatzes gibt es konkrete technische Vorstellungen, wie die Architektur eines Softwarewerkzeugs auszusehen hat. Beschrieben sind diese in einer Reihe von Konzeptions- und Entwurfsmustern (Abbildung 8). Die WAM-Muster stehen in hierarchischen Beziehungen zueinander, die ausdrücken, daß ein untergeordnetes Muster das ihm übergeordnete Muster voraussetzt.

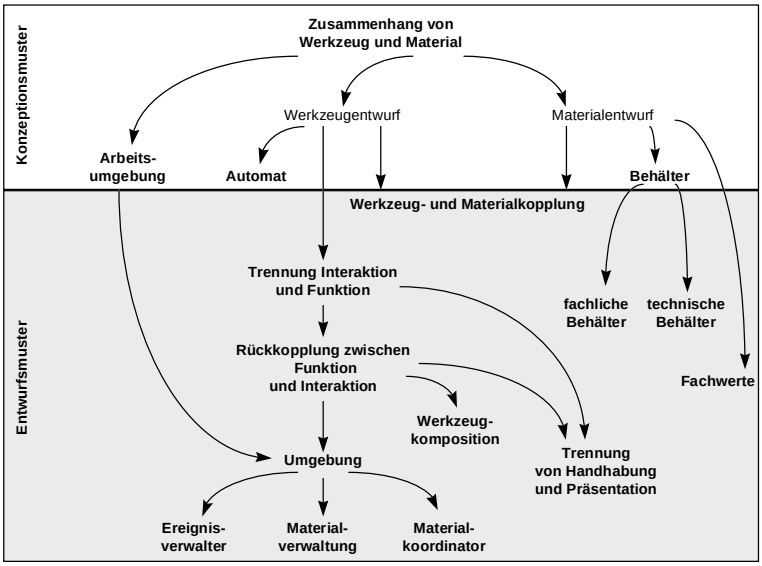

Abbildung 8: WAM-Muster (aus [Züllighoven98], S. 202)

Wie aus der Abbildung ersichtlich, setzen die Entwurfsmuster zum Werkzeugentwurf bei Züllighoven prinzipiell das Muster *Trennung Interaktion und Funktion* voraus. Das Muster beschreibt ein häufig verwendetes Architekturprinzip interaktiver Softwaresysteme. Wie in dem aus Smalltalk bekannten Model-View-Controller-Paradigma (vgl. [KP88]) erfolgt eine strikte Trennung der Werkzeugbestandteile, welche die fachliche Funktion realisieren (Funktionskomponente) von den Werkzeuganteilen, die für die Darstellung an der Benutzungsoberfläche und Handhabung des Werkzeugs verantwortlich sind (Interaktionskomponente). Hierbei stehen die beiden technischen Komponenten in einer asymmetrisch Benutzungsbeziehung zueinander. Die Interaktionskomponente kennt und verwendet die volle Schnittstelle der Funktionskomponente, die Funktionskomponente hat jedoch nur ein sehr eingeschränktes Wissen über ihre Interaktionskomponente. Rückkopplungen von der Funktionskomponente an ihre Interaktionskomponente erfolgen ausschließlich über einen Ereignismechanismus. Der große Nutzen dieses Ansatzes liegt darin, daß die Interaktionskomponente verändert werden kann, ohne daß es eine Auswirkung auf die Funktions-

komponente hat. Umgekehrt wird dagegen eine Änderung der Funktionskomponente in der Praxis in den meisten Fällen auch eine Änderung der Interaktionskomponente bedingen.

Die von einem Werkzeug bearbeitete anwendungsfachliche Aufgabe oder Handlungseinheit ist häufig komplex und aus mehreren von einander getrennten Teilaufgaben zusammengesetzt. Je größer die Komplexität einer Aufgabe ist, desto größer ist auch zwangsläufig die Komplexität des bearbeitenden Werkzeugs. Hierbei zeigt sich, daß Werkzeuge für komplexe Aufgaben aus softwaretechnischer Sicht schnell "unhandlich" und unüberschaubar werden, wenn sie monolithisch aus nur einer Funktions- und einer Interaktionskomponente konstruiert sind. Weiterentwicklung und Anpassung der Werkzeuge werden mit wachsender Werkzeuggröße zunehmend aufwendiger. Züllighoven empfiehlt daher einzelne Teilaufgaben in eigenen Teilwerkzeugen zu realisieren und diese bausteinartig zusammenzusetzen. Auf der anwendungsfachlichen Seite erfolgt bei Werkzeugen entsprechend eine Unterscheidung zwischen einfachen Werkzeugen, die nur eine einzelne Aufgabe bearbeiten und Kombi-Werkzeugen, die komplexere Aufgaben unterstützen und die softwaretechnisch selbst aus separaten Sub-Werkzeugen zusammengesetzt sind.

Zur Realisierung von Kombi-Werkzeugen wird im WAM-Ansatz das Entwurfsmuster *Werkzeugkomposition* verwendet. Das Muster beschreibt softwaretechnisch die hierarchische Zerlegung eines komplexen Werkzeugs in ein Kontext-Werkzeug sowie mehrere unabhängige Sub-Werkzeuge. Jedes Sub-Werkzeug bearbeitet eine weitgehend abgeschlossene Teilaufgabe. Kontext-Werkzeug und Sub-Werkzeug stehen wiederum in einer asymmetrischen Benutzungsbeziehung. Während das Kontext-Werkzeug seine Sub-Werkzeuge unter ihrer vollen Schnittstelle kennt, erfolgt die Rückkopplung vom Sub-Werkzeug zum Kontext-Werkzeug ausschließlich über den Ereignismechanismus.

Das Kalenderwerkzeug kann nach WAM in ein Kontext- sowie zwei Sub-Werkzeuge aufgegliedert werden (vgl. [Riehle95]): Ein Sub-Werkzeug Auflister verwaltet die Terminliste während ein Sub-Werkzeug Texteditor für die Bearbeitung des selektierten Termins zuständig ist. Das Kontext-Werkzeug Kalendereditor koordiniert entsprechend die beiden Sub-Werkzeuge und realisiert die übergeordneten Aufgaben.

Das Muster *Werkzeugkomposition* ist dem Muster *Trennung Interaktion und Funktion* untergeordnet. In WAM wird davon ausgegangen, daß bei einer Zerlegung komplexer Werkzeuge in mehrere unabhängige Sub-Werkzeuge immer auch eine Trennung von Interaktion und Funktion in jedem einzelnen Teilwerkzeug vorliegt. Das führt zu zwei parallelen Hierarchien aus Funktions- und Interaktionskomponenten. Abbildung 9 zeigt die Architektur des Kalenderwerkzeugs.

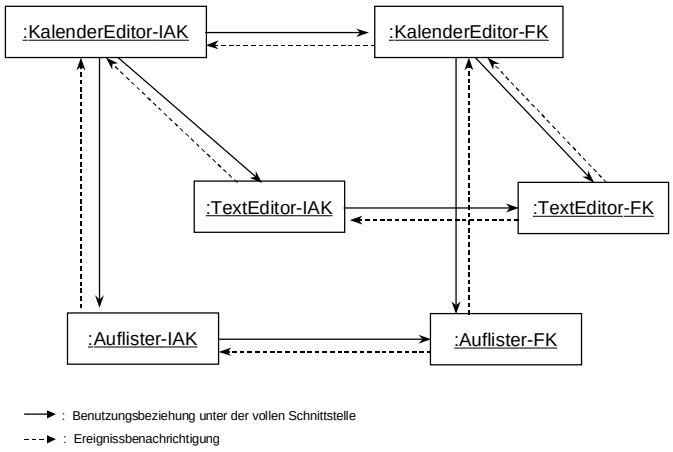

Abbildung 9: Kalenderwerkzeug nach WAM

Betrachtet man es genauer, so zeigt sich, daß die beiden Muster unterschiedliche Abstraktionsebenen beschreiben. Während das Muster *Trennung Interaktion und Funktion* ausschließlich softwaretechnisch motiviert ist, beschreibt das Muster *Werkzeugkomposition* die Werkzeugarchitektur auf einer abstrakteren Ebene. Interaktionskomponenten und Funktionskomponenten existieren direkt als Klassen und Objekte im Implementierungsmodell. Die Begriffe Sub-Werkzeug und Kontext-Werkzeug beschreiben dagegen Konzepte, die sich im Klassen- oder Objektdiagramm nicht direkt wiederfinden. Der Begriff Sub-Werkzeug beschreibt vielmehr eine Einheit bestehend aus Interaktionskomponente **und** Funktionskomponente. Auch besitzt das Muster *Werkzeugkomposition* eine stärkere anwendungsfachliche Ausrichtung. Die Aufteilung in Sub-Werkzeuge orientiert sich an den vom Werkzeug zu bearbeitenden Aufgaben. Die Aufteilung in Interaktions- und Funktionskomponente läßt sich dagegen mit anwendungsfachlichen Argumenten nicht begründen.[10]

Es stellt sich daher die Frage, ob die in Abbildung 8 beschriebene Abhängigkeit zwischen den beiden Mustern zwangsläufig in dieser Art sein muß oder ob die *Werkzeugkomposition* nicht unabhängig von der *Trennung Interaktion und Funktion* betrachtet werden kann. So bleibt es zwar unbestritten, daß eine Trennung von Interaktion und Funktion in vielen Systemen softwaretechnisch eine sinnvolle Lösung darstellt. Auf der anderen Seite gibt es aber auch Anwendungsfälle, in denen sich der Verzicht auf eine Trennung sehr wohl begründen läßt. Beispiele hierfür könnten sehr einfache Sub-Werkzeuge sein oder auch Werkzeuge, die auf dem freien Markt verfügbare Standard-Module (Off-the-Shelf-Komponenten) verwenden, welche die Funktionalität eines Sub-Werkzeugs bereits vollständig abdecken. Unter diesem Blickwinkel beschreibt die Trennung von Interaktion und Funktion lediglich *eine mögliche* Form einer Werkzeugarchitektur und es sind Architekturen vorstellbar, die zwar nach dem Muster der

[10] Ein Indiz dafür, daß die *Werkzeugkomposition* ein abstrakteres Konzept darstellt liefert [Züllighoven98] in der Beschreibung des Musters selbst. Klassendiagramme allein reichen offensichtlich nicht aus, um die Beziehung zwischen Sub- und Kombi-Werkzeugen darzustellen. Auf Seite 280 verwenden die Autoren daher eine Art Komponentendiagramm.

Werkzeugkomposition aufgebaut sind, bei denen jedoch nicht jedes Teilwerkzeug in seiner Implementation zwischen Interaktion und Funktion trennt.

4.3 Komponentenorientierter Werkzeugentwurf

Im folgenden sollen Werkzeugarchitekturen vorgestellt werden, die auf dem Prinzip der Werkzeugkomposition aufbauend einen komponentenorientierten Entwurf verwenden. Ein komplexes Kombi-Werkzeug wird als Komposition mehrerer, miteinander interagierender Werkzeugkomponenten modelliert. Die Aufteilung in Werkzeugkomponenten orientiert sich an den anwendungsfachlichen Teilaufgaben und weniger an einer softwaretechnischen Werkzeugstruktur.

Auf der Ebene der Modellierung abstrahiert eine Werkzeugkomponentenklasse von implementierungstechnischen Details. In einem komponentenorientierten Werkzeugmodell werden die beteiligten Werkzeugkomponenten sowie die Interaktion zwischen ihnen modelliert, aber nicht der innere Aufbau einzelner Werkzeugkomponenten. Das Konzept der Werkzeugkomponentenklasse unterscheidet sich hierdurch klar von dem Konzept der Klasse in einer objektorientierten Sprache.

Eine Werkzeugkomponente entspricht im wesentlichen dem Konzept des Sub-Werkzeugs in WAM. Sie umfaßt also nicht nur die funktionalen Anteile, sondern auch die Bestandteile, die für die Interaktion mit dem Benutzer zuständig sind. Im Gegensatz zu Sub-Werkzeugen ist jedoch die innere Struktur von Werkzeugkomponenten nicht a priori festgelegt. Beispielsweise impliziert eine Werkzeugkomponente nicht automatisch eine Trennung von Interaktion und Funktion. Dagegen sind die Komponentenschnittstellen zu anderen Werkzeugkomponenten sowie die verwendeten Interaktionsprotokolle explizit definiert.

Zur Modellierung komponentenorientierter Werkzeuge sind die klassischen Klassen- oder Objektdiagramme nicht gut geeignet. Eine Klasse beinhaltet immer auch eine konkrete Form der Implementierung und entsprechend sind Klassendiagramme zu eng mit dem konkreten Implementierungsmodell verbunden. Abbildung 10 zeigt die Komponentenstruktur des Kalenderwerkzeugs.

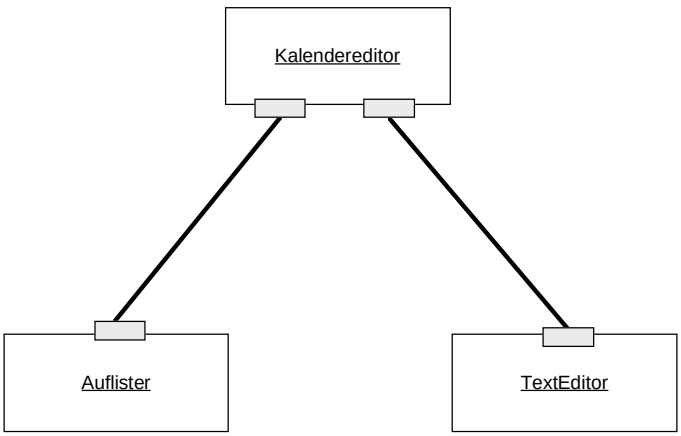

Abbildung 10: Werkzeugkomponenten des Kalenderwerkzeugs

Der Vergleich zu Abbildung 9 macht die unterschiedliche Sichtweise beim komponentenorientierten Entwurf deutlich. Kalendereditor, Auflister und Texteditor erscheinen als explizite Modellelemente. Technische Komponenten wie Interaktionskomponente und Funktionskomponente werden dagegen nicht dargestellt. Die Verbindungslinien zwischen den Werkzeugkomponenten symbolisieren abstrakte[11] Interaktionspfade. Erkennbar ist, daß die Komponenten in einer hierarchischen Beziehung zueinander stehen und daß keine direkte Interaktion zwischen den Sub-Komponenten Auflister und Texteditor existiert.

Die Kommunikation zwischen Werkzeugkomponenten weist eine reguläre Form auf, die sich bei allen Werkzeugen wiederfindet. Die Art der Interaktion ist vergleichbar mit den hierarchischen Strukturen einer Unternehmensbürokratie und der Form wie typischerweise die Aufgaben zwischen Vorgesetzten und Untergebenen verteilt werden. Wie die Mitarbeiter innerhalb einer Unternehmensbürokratie, können auch Werkzeugkomponenten unterschiedliche Rollen innerhalb der Hierarchie übernehmen. Steht eine Werkzeugkomponente am unteren Ende der Hierarchie ist sie nur ausführendes Organ. Ihre Aufträge bekommt sie hierbei direkt von einem externen Auftraggeber bzw. Kunden mit dem sie direkt kommuniziert oder aber von ihrem Vorgesetzten. Ist eine Komponente hierarchisch höher angesiedelt, übernimmt sie neben ihren eigenen Aufgabenbereich außerdem eine Vorgesetztenrolle für die ihr untergeordneten Komponenten. Hieraus ergeben sich typische Formen der Interaktion und des Nachrichtenfluß innerhalb der Hierarchie. Während ein Vorgesetzter einen Untergebenen mit beliebigen Aufgaben beauftragen kann, sofern dieser nur fachlich fähig ist, sie auch zu bearbeiten, wendet sich ein Untergebener nur unter ganz bestimmten Umständen an seinen Vorgesetzten. Zum einen informiert er diesen zur Kenntnisnahme, sobald er eine Aufgabe erledigt hat. Zum anderen reicht er eine Arbeit an seinen Vorgesetzten weiter, falls er fachlich oder technisch nicht in der Lage ist, diese vollständig selbst auszuführen. Hierarchische Systemarchitekturen mit diesen Merkmalen beschreibt [Riehle98] als Bürokratie-Muster. Wendet man das Muster für den Entwurf vom komponentenorientierten Werkzeugen an, ergeben sich für Werkzeugkomponenten typische charakteristische Merkmale. Eine Werkzeugkomponente ist dementsprechend:

- ***eigenständig*** **-** Jede Werkzeugkomponente besitzt ein eigenes Aufgabengebiet, in dessen Rahmen sie Aufgaben selbständig bearbeitet. Im Rahmen ihres Aufgabenbereichs interagiert eine Werkzeugkomponente auch eigenständig direkt mit dem Benutzer. Die Eigenständigkeit impliziert, daß Fragen, was eine sinnvolle Werkzeugkomponente darstellt bzw. welche Granularität Werkzeugkomponenten aufweisen an den anwendungsfachlichen Aufgaben orientiert sein müssen. Werkzeugkomponenten müssen so konzipiert sein, daß sie immer ganze Teilaufgaben bearbeiten können. Im Gegensatz dazu haben die meisten existierenden Komponentenmodelle eher eine technische Ausrichtung. Sehr häufig werden hier Komponenten primär über ihr Erscheinungsbild an der graphischen Benutzungsoberfläche definiert.

  Zum anderen liegt es allein im Verantwortungsbereich einer Komponente wie sie eine Aufgabe abwickelt. Die Art der Aufgabenabwicklung darf für das Umfeld also auch für die Vorgesetztenkomponente nicht von Belang sein.

[11] Die Darstellung ist eine vergröberte Sicht auf die Komponentenstruktur, die nicht genau zeigt, welche Komponentenschnittstellen im Detail an der Interaktion beteiligt sind und welche Protokolle bei der Interaktion verwendet werden.

Technisch bedeutet das, daß eine Werkzeugkomponente eine starke Abgeschlossenheit gegenüber jeglichen Implementationsdetails aufweist (Geheimnisprinzip).

- ***zusammensetzbar –*** Eine Werkzeugkomponente kann als eigenständige Einheit agieren oder zusammen mit anderen Komponenten in einer Komposition eine komplexe größere Werkzeugkomponente bilden. In einer zusammengesetzten Werkzeugkomponenten kann eine ganze Hierarchie von Werkzeugkomponenten als eine Einheit behandelt und eingesetzt werden.
- ***direkter ansprechbar*** **-** Eine Werkzeugkomponente kann Arbeitsaufträge von zwei Seiten entgegennehmen. Zum einen bekommt sie über spezielle Schnittstellen Arbeitsaufträge von ihrer übergeordneten Komponente. Zum anderen kann sie über die Benutzungsschnittstelle Anweisungen direkt vom Anwender erhalten ohne, daß die übergeordnete Komponente hieran beteiligt ist.
- ***verantwortlich*** **-** Eine Werkzeugkomponente fühlt sich immer für alle an sie gerichteten Arbeitsaufträge verantwortlich. Entweder bearbeitet sie einen Auftrag vollständig selbst - gegebenenfalls unter Beanspruchung von Dienstleistungen der ihr selbst unterstellten Werkzeugkomponenten - oder sie delegiert diesen an ihre Vorgesetztenkomponente weiter. Entsprechend dem Muster *Chain of Responsibility* [GHJV95] existiert daher immer eine Zuständigkeitskette in Aufwärtsrichtung der Komponentenhierarchie.
- ***informativ*** **-** Über jede Änderung am Material- oder am sichtbaren Komponentenzustand informiert eine Werkzeugkomponente registrierte Beobachter über entsprechende Ereignisse. Die Werkzeugkomponente ist aber in keiner Weise davon abhängig, ob sich überhaupt eine Komponente für ein Ereignis registriert hat oder wie ein Beobachter auf ein Ereignis reagiert. Innerhalb der Komponentenhierarchie ist es üblich, daß sich Vorgesetztenkomponenten für Ereignisse ihrer Untergebenen registrieren.
- ***delegierend*** **-** Sind einer Werkzeugkomponente andere Komponenten unterstellt, so kennt die Komponente die Schnittstellen ihrer Untergebenen genau und ist daher von diesen auch abhängig. Zur Bearbeitung ihrer eigenen Aufgaben delegiert eine Vorgesetztenkomponente Teilaufgaben an ihre Untergebenen weiter. Eine Vorgesetztenkomponente übernimmt auch immer die Rolle eines Mediators für die ihr untergeordneten Werkzeugkomponenten. Hierdurch wird vermieden, daß die untergeordneten Komponenten direkt miteinander kommunizieren müssen. Die Komponenten werden hierdurch unabhängig von anderen Komponenten der Hierarchie. Eine Vorgesetztenkomponente bildet innerhalb einer Komponentenhierarchie die Schnittstelle zu allen ihr unterstellten Werkzeugkomponenten.
- ***kontextunabhängig*** **-** Eine Werkzeugkomponente interagiert mit ihrem Umfeld ausschließlich über die definierten Schnittstellen. Eine Werkzeugkomponente ist auch nicht von Komponenten, die in der Hierarchie über ihr stehen, abhängig. Obgleich sich eine Werkzeugkomponente darüber "bewußt" ist, daß sie in einer Hierarchie arbeiten, weiß sie von ihrer Vorgesetztenkomponente nur soviel, daß sie an diese unerledigte Aufträge weiterreichen kann.

Ein wesentliches Merkmal komponentenorientierter Werkzeugarchitekturen ist, daß die Arbeit dezentral in den Komponenten ausgeführt wird. Es gibt keine zentrale, allmächtige Instanz, die alle Komponenten koordiniert und zusammen-

hält. Zentrale Koordinatoren neigen dazu, zu inflexiblen und änderungsanfälligen Flaschenhälsen einer Architektur zu werden. Die bürokratische Organisation innerhalb eines Werkzeugs gewährleistet eine selbstregulierende Konsistenz innerhalb der Werkzeugkomponentenhierarchie.

4.3.1 Metamodell

Die Beziehung zwischen Werkzeugkomponenten und Komponenten im allgemeinen wie sie im Rahmen dieser Diplomarbeit verstanden wird, ist als Metamodell in Abbildung 11 dargestellt.

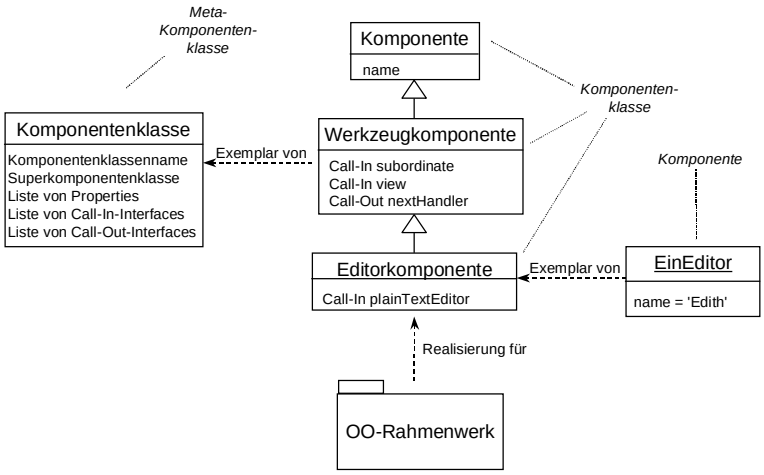

Abbildung 11: Komponenten-Metamodell

Komponenten benennen explizit die Schnittstellen, über die sie mit ihrer Umgebung interagieren. Jede Schnittstelle ist durch einen Typ und einen eindeutigen Namen spezifiziert. Es werden zwei Arten von Schnittstellen unterschieden:

(1) Schnittstellendefinitionen, über die eine Komponente ihre Dienstleistungen anderen zugänglich macht (Call-In-Schnittstellen)

(2) Spezifikationen von Dienstleistungen auf die eine Komponente selbst zurückgreift und die von anderen Komponenten erbracht werden müssen (Call-Out-Schnittstellen).

Aus fachlicher Sicht beschreibt eine Komponentenschnittstelle[12] eine gewisse Rolle, welche die Komponente in einem Kontext einnehmen kann bzw. eine Dienstleistung, die die Werkzeugkomponente anderen anbietet. Eine Komponentenschnittstelle besitzt immer eine entsprechende, dem Kontext bekannte definierte Semantik. Vergleichbar mit der Kopplung von Werkzeug und Material, kann eine Schnittstelle auch als eine Art Aspekt betrachtet werden, unter dem sich die Komponente in einem Kontext (z.B. einem Werkzeug) einsetzen läßt. Technisch betrachtet besteht eine Schnittstelle aus einer Menge von Operationen und einem Protokoll, welches deren gültige Verwendung definiert.

[12] Der sprachlichen Vereinfachung wegen wird im folgenden meist einfach von *Schnittstellen* gesprochen, wenn aus dem Kontext klar ist, ob hierunter Call-In- und/oder Call-Out-Schnittstellen zu verstehen sind.

Neben den Schnittstellen besitzt eine Komponente einstellbare Eigenschaften (Properties), über die ihr generelles Verhalten und Aussehen an der Benutzungsoberfläche konfiguriert werden kann.

Die konzeptionellen Ähnlichkeiten zwischen Objekten und Komponenten werden deutlich, wenn man zum Vergleich das Objekt-Metamodell der Programmiersprache Java betrachtet (Abbildung 12).

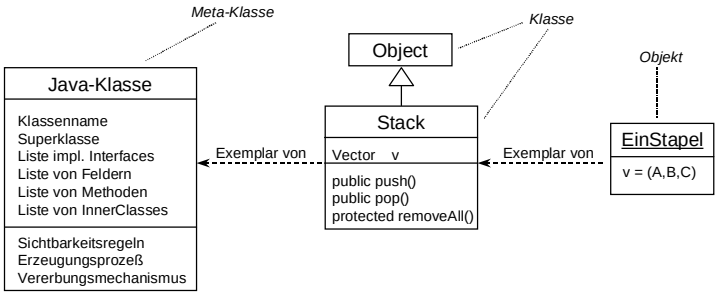

Abbildung 12: Java Objekt-Metamodell

4.3.2 Modellarchitektur für komponentenorientierte Werkzeugarchitekturen

Die allgemeinen Architekturprinzipien von Werkzeugarchitekturen, die aus hierarchisch interagierenden Werkzeugkomponenten bestehen, können als eine eigene Modellarchitektur beschrieben werden. Wie in Kapitel 2.4 ausgeführt, definiert eine Modellarchitektur die zu Verfügung stehenden Modellelemente, ihre Verknüpfungsmöglichkeiten und enthält Regeln, Einschränkungen sowie eine semantische Interpretation der einzelnen Elemente.

4.3.2.1 Modellelement: Werkzeugkomponente

Die wichtigsten Modellelemente sind Werkzeugkomponenten. Eine Werkzeugkomponente ist eine abgeschlossene Modelleinheit, die eine bestimmte anwendungsfachliche Teilaufgabe innerhalb eines Softwarewerkzeugs bearbeitet. Jede Werkzeugkomponente realisiert die Schnittstellen *Subordinate* und *View* und benötigt innerhalb eines Systems selbst eine Bindung an die *RequestHandler*-Schnittstelle einer anderen Komponente. Speziellere Werkzeugkomponenten, wie die Auflisterkomponente oder die Texteditorkomponente des Kalenderwerkzeugs, können dem weitere Schnittstellen hinzufügen oder die Basisschnittstellen typkonform erweitern. Im folgenden wird die Semantik der Schnittstellen im einzelnen erläutert.

Subordinate

Werkzeugkomponenten bieten mindestens eine (anwendungsfachliche) Dienstleistung, zur Bearbeitung ihrer Aufgabe innerhalb eines Werkzeugs. Damit sich

eine Dienstleistung in eine Werkzeugkomponentenhierarchie integrieren läßt, muß sie die Schnittstelle Subordinate erfüllen.

Die Bezeichnung Subordinate orientiert sich an der Rolle, die eine Komponente unter dieser Schnittstelle einnimmt. Als untergeordnete Komponente stellt sie über diese Schnittstelle ihre Dienstleistung einer übergeordneten Werkzeugkomponente zur Verfügung und empfängt entsprechende Arbeitsaufträge. Auf dieser abstrakten Ebene ist die Komponentenfunktionalität durch generische Aktionsbeschreibungen definiert. Abhängig vom inneren Komponentenzustand kann eine Werkzeugkomponente ihre Aktionen vorübergehend auch deaktivieren. Über Aktivierungen und Deaktivierungen von Aktionen informiert eine Subordinate ihre Vorgesetztenkomponente durch entsprechende Ereignisse. Das Konzept der Aktionen entspricht im wesentlichen dem von Gamma et al. beschriebenen Command-Muster [GHJV95].

Eine weitere Dienstleistung, die über die Schnittstelle Subordinate bereitgestellt wird, ist ein Undo/Redo-Mechanismus für die Widerrufung durchgeführter Aktionen. Eine Subordinate signalisiert ihrem Kontext, sobald eine widerrufbare Aktion ausgeführt worden ist.

Konkrete Ableitungen der allgemeinen Werkzeugkomponente erweitern die Subordinate-Schnittstelle i.d.R. um differenziertere Zugriffsoperationen. Hierdurch lassen sich die Dienste einer Komponente nicht bloß unter den generischen Aktionsbeschreibungen ansprechen, sondern unter einer spezialisierteren komponentenbezogenen Schnittstelle. Häufig sind diese komponentenbezogenen Schnittstellen an einem Materialaspekt des von der Werkzeugkomponente bearbeiten Materials orientiert. Arbeitet eine Werkzeugkomponente auf einem Material unter verschiedenen Aspekten, bietet es sich meistens an, die Komponente für jeden einzelnen Aspekt mit einer eigenen Schnittstelle auszustatten.

Beispiel: Abbildung 13 zeigt einer Editorkomponente, die ihre speziellen Dienste über zwei unterschiedliche Schnittstellen anbietet (Zur verwendeten Notation siehe Seite 9.). Eine PlainText-Schnittstelle, für die einfachste Form der rein zeichenorientierten Textbearbeitung und eine StyledText-Schnittstelle, die es erlaubt Texte auch mit zusätzlichen Formatierungen, wie Schriftgrößen, Fettdruck, etc. auszuzeichnen. Die beiden Schnittstellen bearbeiten ein Material unter unterschiedlichen Aspekten. Wie die Abbildung zeigt, erfüllen beide Schnittstellen das Subordinate-Protokoll. Die Editorkomponente läßt sich folglich je nach Bedarf entweder mit ihrer PlainText- oder mit ihrer SyledText-Schnittstelle in eine Komponentenhierarchie einbetten.

Neben diesen beiden editorspezifischen Schnittstellen besitzt die Komponente in Abbildung 13 eine weitere Schnittstelle *Debug*. Diese Schnittstelle ermöglicht ein Debugging der Komponente auf einer abstrakteren Ebene als es mit den Debuggern des Entwicklungssystems möglich ist. Da Debug jedoch, anders als die anderen beiden Schnittstellen, nicht das Subordinate-Protokoll erfüllt, läßt sich die Editorkomponente unter dieser Schnittstelle nicht in eine Werkzeugkomponentenhierarchie einbetten.

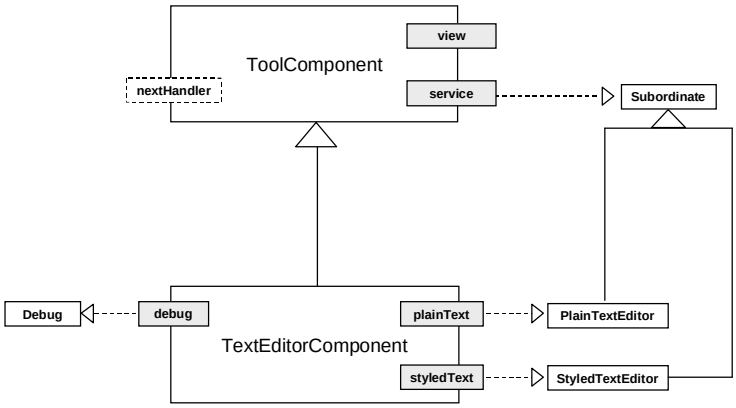

Abbildung 13: Editorkomponente

NextHandler

Innerhalb eines Werkzeugs sind Werkzeugkomponenten in einer hierarchischen Struktur angeordnet. Da aber nicht sämtliche Kommunikation über die Vorgesetztenkomponente verläuft, sondern eine Komponente auch direkt - über die graphische Benutzungsschnittstelle - vom Anwender Anforderungen erhält, kann es vorkommen, daß eine Komponente einen Auftrag selbst nicht vollständig ausführen kann. Für diese Fälle benötigt eine Werkzeugkomponente eine Bindung zu einer Komponente, an die sie die Aufträge (Requests) weiterreichen kann. Die Call-Out-Schnittstelle *NextHandler* erfordert eine Bindung an eine Schnittstelle, die das Protokoll RequestHandler realisiert. Hierüber kann eine Zuständigkeitskette entsprechend dem Muster Chain-of-Responsibility [GHJV95] aufgebaut werden.

In einer Werkzeugkomponentenhierarchie wird diese Call-Out-Schnittstelle in der Regel an die Manager-Schnittstelle (Beschreibung folgt) der übergeordneten Werkzeugkomponente gebunden. Die Zuständigkeitskette verläuft damit aufwärts in der Komponentenhierarchie. Es gibt zwei typische Fälle, die bei nahezu allen Werkzeugen auftreten, in denen die Zuständigkeitskette benötigt wird:

- *Schließen eines Werkzeugs:* Keine einzelne Werkzeugkomponente ist in der Lage vollständig selbst zu entscheiden, ob und wie ein Werkzeug geschlossen (beendet) werden kann. Letzten Endes kann diese Aufgabe nur in der Umgebung außerhalb eines Werkzeug bearbeitet werden. Schließen-Anforderungen müssen von allen Komponenten deshalb immer über die Zuständigkeitskette weitergereicht werden. Das bedeutet, daß in einem Softwarewerkzeug NextHandler auch bei der Wurzelkomponente an eine RequestHandler-Schnittstelle gebunden sein muß. Im Gegensatz zu untergeordneten Werkzeugkomponenten ist die implementierende Komponente jedoch selbst i.d.R. keine Werkzeugkomponente mehr (siehe auch Erklärungen zur Root-Schnittstelle auf Seite 40).
- *Kontextsensitive Popup-Menüs:* Moderne, benutzungsfreundliche Softwarewerkzeuge sollten dem Anwender kontextsensitive Popup-Menüs anbieten. Eine einzelne Werkzeugkomponente ist zwar in der Lage zu entscheiden,

welche ihrer eigenen Aktionen in einer gegebenen Situation, d.h. abhängig von ihrem inneren Komponentenzustand, möglich und sinnvoll sind, jedoch sollte das Popup-Menü auch die entsprechenden Aktionen der übergeordneten Komponenten enthalten. Erhält eine untergeordnete Komponente eine Benutzeranforderung für ein Popup-Menü, kann sie die für sie relevanten Aktionen zusammenstellen und diese mit einer Bitte um Ergänzung an ihre Vorgesetztenkomponente weiterleiten. Erst die Wurzelkomponente an der Spitze der Komponentenhierarchie veranlaßt dann die Visualisierung des kompletten Popup-Menüs.

View

Jede Werkzeugkomponente realisiert mindestens eine View-Schnittstelle. Über diese Schnittstelle läßt sich eine Komponente in die Benutzungsoberfläche des Werkzeugs einbetten. Die View-Schnittstelle ist für das verwendeten Toolkit des Fenstersystems bzw. für das eingesetzte GUI-Rahmenwerk spezialisiert. Die im Rahmen dieser Diplomarbeit implementierten Werkzeugkomponenten implementieren eine SwingView-Schnittstelle, die den Komponenteneinsatz in einer Java Swing-Oberfläche ermöglichen. Alternativ könnte eine Komponente aber auch View-Schnittstellen für andere Toolkits wie beispielsweise AWT, Bongo [Goodman97] oder SubArctic [Subarctic96] bereitstellen. Ein besonders flexible einsetzbare Komponente könnte durchaus auch mehrere View-Schnittstellen anbieten von denen je nach Einsatzkontext eine andere verwendet wird.

Die Vorgabe, daß jede Werkzeugkomponente über die View-Schnittstelle ihre Oberflächenrepräsentation bereitstellt, bewirkt eine relativ enge Kopplung von Darstellung und Funktionalität innerhalb der Komponente. In [Züllighoven98] werden andere Konstruktionsansätze diskutiert, welche auch bei zusammengesetzten Werkzeugen die gesamte Oberfläche in einer einzigen technischen Komponente realisieren. Dieses entkoppelt die Darstellung einer Komponente stärker von ihrer Funktionalität und kann helfen, die Werkzeugoberfläche insgesamt homogener zu gestalten und die gesamte Handhabung stärker zu vereinheitlichen. Ungeachtet der potentiellen Vorteile, die sich durch eine stärkere Trennung von Darstellung und Funktionalität ergeben, ist dieser Konstruktionsansatz in dem hier entwickelten Werkzeugkomponentenmodell nicht verwendbar. Ich gehe jedoch davon aus, daß das im allgemeinen kein großer Nachteil sein wird, da die Entkopplung von Darstellung und Funktionalität in der Praxis de facto nur selten möglich oder sinnvoll ist. Eine Beobachtung, die [Bäumer98] im Kontext der Schichtenbildung klassischer Drei-Schichten-Architekturen gemacht hat, kann auf die Struktur von Werkzeugkomponenten übertragen werden. Bäumer kritisiert, daß eine Trennung in Präsentation, Funktion und Daten drei Schichten erzeugt, die in keiner Richtung wirklich unabhängig voneinander sind. Änderungen an der Präsentationsschicht ziehen fast immer Änderungen an der funktionalen und Datenschicht nach sich und genauso führen Änderungen in der funktionalen oder der Datenschicht wiederum zu Änderungen der Darstellung.

> *"Denn neu zu präsentierende Informationen führen meistens zu neuen Attributen in den Datenobjekten, und fachlich motivierte Erweiterungen in der Handhabung des Systems, etwa durch neue Menüeinträge vergegenständlicht, bedingen Änderungen an der funktionalen Schicht. Im Gegenzug führen Änderungen an den Datenobjekten ebenso zu Anpassungen an der Präsentation, da neue*

> *Attribute i.d.R. auch an der Benutzungsschnittstelle dargestellt werden." [Bäumer98, S. 109]*

Vergleichbar ist zu erwarten, daß auch bei einer Werkzeugkomponente die Darstellung naturbedingt nicht unabhängig von der Komponentenfunktionalität sein wird und Änderungen sich gegenseitig stark beeinflussen. Eine engere Kopplung der beiden Aspekte scheint daher durchaus gerechtfertigt zu sein.

4.3.2.2 Modellelement: Zusammengesetzte Werkzeugkomponente

Durch Vererbungsbeziehungen lassen sich Generalisierungen und Spezialisierungen von Komponentenklassen modellieren. Zu den einfachen Werkzeugkomponenten gibt es die Unterklassen der zusammengesetzten Werkzeugkomponenten. Zusammengesetzte Werkzeugkomponenten sind Komponenten, die selbst wiederum aus anderen Werkzeugkomponenten zusammengesetzt sind. Sie besitzen zusätzliche Schnittstellen, die es ermöglichen, ihnen Werkzeugkomponenten unterzuordnen. Zusammengesetzte Komponenten erfüllen immer auch die Anforderungen einfacher Werkzeugkomponenten. Das heißt, sie realisieren auch alle Schnittstellen der Klasse Werkzeugkomponente. Hierdurch ist es möglich, eine zusammengesetzte Komponente ebenfalls rekursiv wieder einer anderen Komponenten unterzuordnen und dadurch Werkzeugkomponenten in einer hierarchischen Baumstruktur anzuordnen (vgl. Composite-Muster [GJHV95]). Im folgenden werden die Schnittstellen von zusammengesetzten Werkzeugkomponenten genauer betrachtet.

Subordinate

Zusammengesetzte Werkzeugkomponenten erweitern ihre eigenen Fähigkeiten um die Fähigkeiten ihrer Untergebenen. Da sie entsprechende Aufträge an die ihnen unterstellte Subordinaten jederzeit weiterdelegieren können, bietet eine zusammengesetzte Komponente an ihrer Subordinate-Schnittstelle üblicherweise eine Obermenge der Aktionen ihrer Subordinaten.

Manager

Über die Schnittstelle Manager kann eine Werkzeugkomponente die Rolle einer übergeordneten Vorgesetztenkomponente für die ihr untergeordneten Komponenten übernehmen. Die Manager-Schnittstelle wird von allen zusammengesetzten Komponenten implementiert. Da die Manager-Schnittstelle eine Erweiterung des RequestHandler-Protokolls ist, kann eine zusammengesetzte Komponente unter dieser Schnittstelle als Nachfolger innerhalb der Zuständigkeitskette agieren. Eine Komponente, die über ihre Manager-Schnittstelle eine Anforderung (Request) erhält, die sie aber selbst nicht vollständig bearbeiten kann, leitet die Anforderung über ihre eigene Call-Out-Schnittstelle *NextHandler* an die ihr übergeordnete Komponente weiter.

Im Rahmen einer Werkzeugkomponentenhierarchie ist die Manager-Schnittstelle einer übergeordneten Komponente prinzipiell bei allen untergeordneten Komponenten an deren NextHandler-Schnittstelle gebunden.

Sub <X>

Für jede Subordinate, auf deren Dienst eine zusammengesetzte Werkzeugkomponente basiert, spezifiziert eine Call-Out-Schnittstelle die konkret benötigte Dienstleistung. Zur korrekten Arbeitsweise der zusammengesetzten Komponente

müssen diese Schnittstellen zur Laufzeit an entsprechende Subordinate-Schnittstellen anderer Komponenten gebunden werden. So benötigt die Kalendereditorkomponente beispielsweise Zugriff auf eine Auflister- und eine Editor-Dienstleistung.

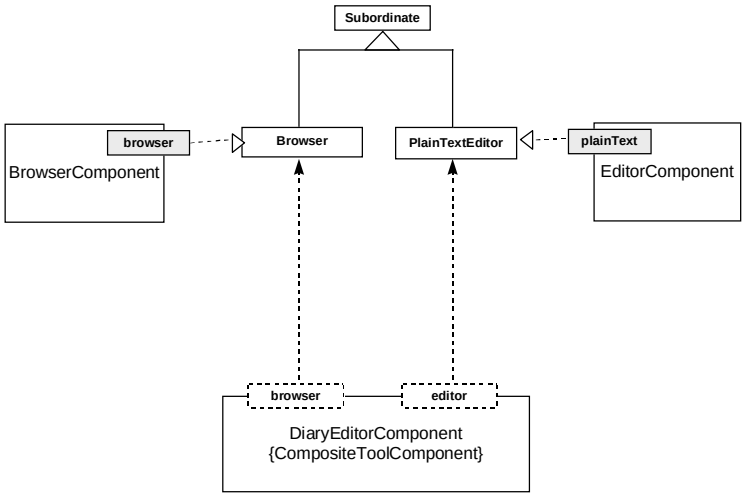

Abbildung 14: Call-Out-Schnittstellen von DiaryEditorComponent

4.3.2.3 Modellelement: Wurzelkomponente

Wurzelkomponenten sind eine weitere Spezialisierung der zusammengesetzten Werkzeugkomponenten. Sie bilden den Abschluß einer Werkzeugkomponentenhierarchie. Abbildung 15 zeigt die gesamte Vererbungsbeziehung zwischen einfachen Werkzeugkomponenten (ToolComponent), zusammengesetzten Werkzeugkomponenten (CompositeToolComponent) und Wurzelkomponenten (RootToolComponent) sowie deren Basisschnittstellen.

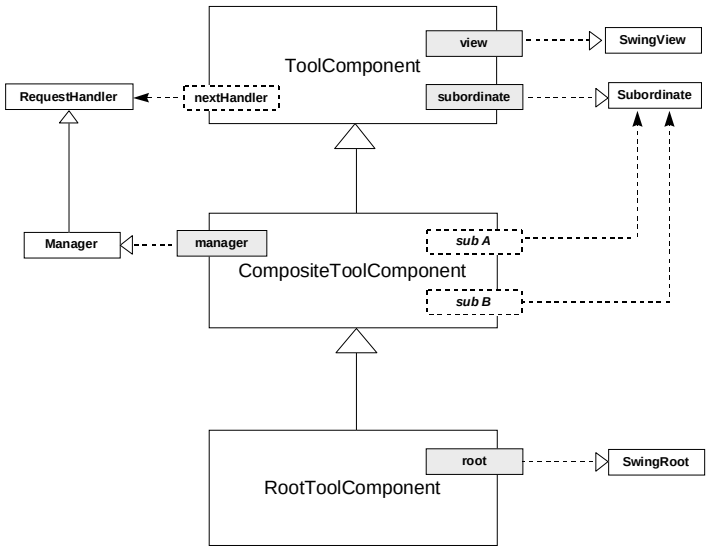

Abbildung 15: Werkzeugkomponenten

Die Schnittstellen der Wurzelkomponente besitzen die folgende Semantik:

Subordinate

Zum Aufgabenbereich einer Wurzelkomponente gehört es einen Undo-Manager zu installieren, über den sich durchgeführte Aktionen kontrolliert zurücksetzen lassen. Der Undo-Manager beobachtet alle Änderungen innerhalb seiner Komponentenhierarchie. Hierdurch definiert eine Wurzelkomponenten eine unteilbare Konsistenzeinheit, in der Folgen von Aktionsausführungen zusammenhängend betrachtet werden müssen. Ein separates, unabhängiges Rücksetzen von Aktionen einer einzelnen untergeordneten Komponente ist nicht möglich, da dieses zu einem inkonsistenten Werkzeug- und Materialzustand führen würde. Für das Kalenderwerkzeug bedeutet das beispielsweise, daß es nicht möglich ist, separat ausschließlich die verändernden Aktionen der untergeordneten Editorkomponente zurückzusetzen, da dieses nicht berücksichtigen würde, zu welchem Termin eine widerrufene Textänderung ursprünglich gehörte.

Root

Unter der Root-Schnittstelle wird eine Komponentenhierarchie als gesamtes Werkzeug aufgefaßt. Die Schnittstelle ermöglicht das kontrollierte Starten und Beenden eines Werkzeugs und aller seiner Komponenten. Ergänzend zu der View-Schnittstelle, stellt die Root-Schnittstelle zusätzlich noch eine statische Menüzeile, eine Symbolleiste (Toolbar) sowie eine Statuszeile bereit.

Das äußere Fenster eines Werkzeugs ist jedoch nicht Bestandteil der Root-Schnittstelle. Das Fenster muß prinzipiell vom Klienten der Root-Schnittstelle bereitgestellt werden. Diese Entwurfsentscheidung ermöglicht die einfache Verwendung einer Wurzelkomponente in verschiedenen Kontexten. So kann das Kalenderwerkzeug beispielsweise unter seiner Root-Schnittstelle einheitlich sowohl als eigenständige Anwendung auf der Betriebssystemebene (Abbildung 7, Seite 26), als Applet innerhalb eines Web-Browsers (Abbildung 16) oder auch als separates, aber abhängiges Subwerkzeug innerhalb einer Organizer-Anwendung verwendet werden (z.B. vergleichbar mit Programmen wie Microsoft Outlook oder Lotus Organizer).

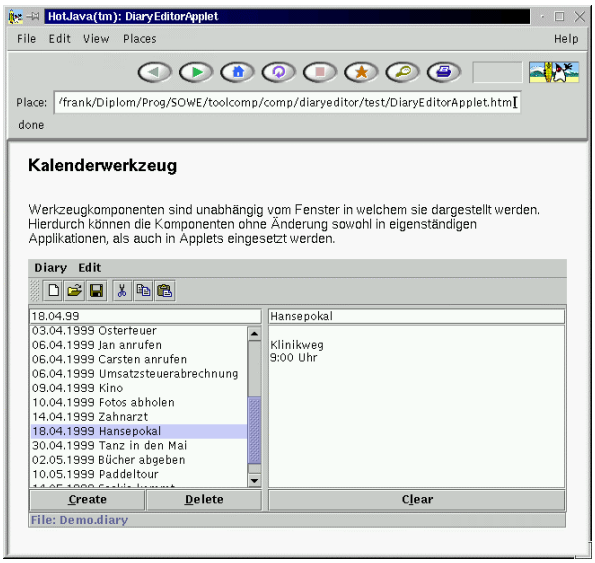

Abbildung 16: Kalenderwerkzeug als Applet

4.3.2.4 Komponenteninteraktion

Werkzeugkomponenten interagieren innerhalb eines Systems ausschließlich entlang definierter Interaktionspfade. Ein Interaktionspfad ist die Bindung einer Call-Out-Schnittstelle an eine Call-In-Schnittstelle. Eine Bindung läßt sich nur zwischen protokollkonformen Schnittstellen aufbauen. Call-In- und Call-Out-Schnittstelle müssen das gleiche Protokoll verstehen.

Es gibt drei mögliche Mechanismen, wie Werkzeugkomponenten auf einem Interaktionspfad miteinander kommunizieren können:

(1) Operationsaufrufe

(2) Ereignisbenachrichtigungen

(3) Generische Bearbeitungsaufträge (Requests)

Operationsaufrufe: Bei Operationsaufrufen wird unterschieden zwischen rein sondierenden Operationen (Funktionsaufrufe) und verändernden Operationen (Prozeduren). Operationsaufrufe werden nur abwärts der Komponentenhierarchie eingesetzt, wenn ein Manager Aufgaben an eine seiner untergeordnete Komponenten delegiert. Operationsaufrufe setzen voraus, daß der Aufrufer genaue Kenntnis davon hat, was eine Operation semantisch bewirkt. Der Aufrufer kann sich auch darauf verlassen, daß die gerufene Komponente in angemessener Weise auf den Operationsaufruf reagiert. Aufrufe von sondierenden Operationen erfolgen prinzipiell synchron, Aufrufe von verändernden Operation erfolgen dagegen grundsätzlich asynchron. Die Asynchronität auf der abstrakten Ebene des Architekturmodells ist aber nicht automatisch mit einer Asynchronität im Implementierungsmodell gleichzusetzen. Das bedeutet verändernde Operationen müssen nicht zwangsläufig auch asynchron implementiert werden. Die Asynchronität bei der Komponenteninteraktion beinhaltet jedoch zwei Konsequenzen:

(1) Ein Aufrufer darf keine Annahmen darüber machen, wann ein Aufruf ausgeführt wird und welche Zustandsänderungen der Komponente er bewirkt.

(2) Nachdem eine Operation ausgeführt worden ist, muß die Komponente sichtbare Zustandsänderungen durch entsprechende Ereignisse signalisieren. Der Aufrufer von asynchronen Operationen kann diese Ereignisse beobachten und dann entsprechend reagieren.

Asynchronität auf der Architekturebene wird als ein konzeptionelles Element der Komponenteninteraktion verstanden und nicht als eine technische Implementierungseigenschaft.

Ereignisse: Änderungen am Werkzeug- oder Materialzustand werden von einer Werkzeugkomponente durch Ereignisse signalisiert. Interessiert sich ein Klient für das Auftreten eines speziellen Ereignisses, kann er sich hierfür registrieren lassen. Die ereignisaussendende Komponente benötigt hierbei über die Ereignisbeobachter nur gerade soviel Wissen, daß sie in der Lage ist, diese über das Auftreten der Ereignisse zu informieren. Da Ereignisse im allgemeinen spezifisch für eine bestimmte Dienstleistung sind bzw. speziell im Kontext einer bestimmten Rolle der Komponente interpretiert werden müssen, werden sie direkt in den entsprechenden Dienstleistungsschnittstellen definiert. Es gibt folglich keine zentrale Komponentenschnittstelle an der alle möglichen Ereignisse einer Komponente definiert sind.

Ereignisbenachrichtigungen erfolgen ausschließlich aufwärts in der Komponentenhierarchie. Ereignisbenachrichtigungen sind aber immer nur als ein Angebot einer Komponente an ihren Kontext zu verstehen. Es obliegt der übergeordneten Werkzeugkomponente sich für entsprechende Ereignisse zu interessieren. Eine unbedingte Beachtung von Ereignissen kann von der aussendenden Komponente nicht erzwungen werden.

Requests: Requests bilden einen Kommunikationsmechanismus, der zwischen Operationsaufrufen und Ereignisbenachrichtigungen einzuordnen ist. Requests ermöglichen die Formulierung allgemeiner, generischer Bearbeitungsgesuche. Im Gegensatz zu Operationsaufrufen benötigt die requestaussendende Komponente sehr viel weniger Wissen darüber, wer der konkrete Request-Empfänger ist, wie dieser auf den Request tatsächlich reagieren wird und ob er den Request direkt selbst bearbeitet oder ihn an andere Stellen weiterreicht. Die Kopplung zwischen Request-Sender und Empfänger ist sehr viel schwächer als bei Operationsaufrufen. Im Gegensatz zu einer Ereignisbenachrichtigung ist ein Request jedoch mehr als lediglich ein Angebot an den Kontext. Vielmehr *fordert* die requestaussendende Komponente vom Empfänger, daß dieser auf den Request in geeigneter Weise reagiert. Der Requestmechanismus beinhaltet keine Möglichkeit, um Bearbeitungsergebnisse an einen Sender zurückzureichen.

Eine Kommunikation über Requests erfolgt ausschließlich aufwärts in der Komponentenhierarchie. Abbildung 17 zeigt die Richtungen der drei Kommunikationsmechanismen innerhalb einer Werkzeugkomponentenhierarchie.

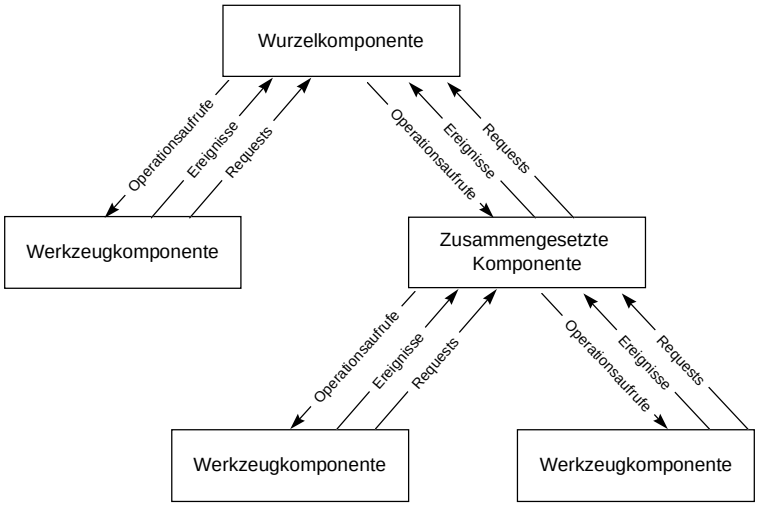

Abbildung 17: Kommunikationsrichtungen

4.3.2.5 Konfigurationsregeln

Die Werkzeugkomponenten eines Werkzeugs sind generell in einer baumförmigen Hierarchie angeordnet. Hierbei gilt, daß eine Werkzeugkomponente *S* einer Werkzeugkomponente *M* untergeordnet ist, genau dann, wenn eine Schnittstelle *Subordinate* von *S* an eine der Call-Out-Schnittstellen *Sub<x>* von *M* gebunden ist. Eine wohlgeformte Werkzeugkomponentenhierarchie ist gegeben, wenn

(1) bei allen Werkzeugkomponenten *S*, die einer Komponente *M* untergeordnet sind, ist die Call-Out-Schnittstelle *NextHandler* an die Call-In-Schnittstelle *Manager* von *M* gebunden ist und

(2) *Manager* die einzige Komponentenschnittstelle ist, die *S* von *M* verwendet und

(3) die Wurzel der Hierarchie eine Komponente vom Typ *RootToolComponent* ist.

Diese Regeln lassen noch Freiheiten in der konkreten Werkzeugarchitektur. So ist es möglich, daß zwei Werkzeugkomponenten, die auf einer Hierarchiestufe stehen, prinzipiell direkt miteinander kommunizieren ohne über ihre übergeordnete Komponente zu gehen. [Riehle98] weist aber darauf hin, daß diese *Abkürzungen* nur in begründeten Ausnahmefällen eingerichtet werden sollten, wenn sicher ist, daß die übergeordnete Komponente niemals Einfluß auf die Kommunikation zwischen den untergebenen Komponenten nehmen muß.

4.3.2.6 Einschränkungen

Die vorgestellte Modellarchitektur für komponentenorientierte Werkzeugarchitekturen basiert auf einigen einschränkenden Annahmen. Zum einen wird davon ausgegangen, daß jede Werkzeugkomponente eine graphische Repräsentation an der Benutzungsoberfläche besitzt. Ein wesentliches Merkmal der hier diskutierten Werkzeugarchitekturen ist eben gerade, daß sie eine direkte Interaktion des Benutzers mit jeder Stufe der Werkzeugkomponentenhierarchie erlauben, wodurch die speziellen bürokratischen Formen der Interaktion zwischen Komponenten überhaupt erst erforderlich werden. Werkzeuge oder Automaten ohne eine GUI-Oberfläche besitzen Architekturen, die von den hier vorgestellten abweichen. Es wird hier die Eigenschaft von Modellarchitekturen deutlich, daß sie immer speziell für eine begrenzten Einsatzkontext konzipiert werden und nicht allgemeine Architekturbeschreibungen darstellen.

Eine weitere Einschränkung ist, daß das Modell darauf basiert, daß sich ein Werkzeug nicht über Prozeßgrenzen hinweg verteilt. Wie Waldo et al. in [WWWK94] gezeigt haben, lassen sich Verteilungsaspekte - obgleich vielerorts versprochen - nicht allein auf der technischen Implementierungsebene behandeln, sondern wirken sich bis in höhere Architekturebenen aus. Die mögliche und sinnvolle Verteilung von Werkzeugen ist eine Problemstellung, die zu ihrer Lösung noch intensivere Forschungsarbeit erfordert. Die hier vorgenommene Einschränkung, daß ein Werkzeug immer vollständig in einem Adreß- und Prozeßraum abläuft, stellt eine pragmatische Vereinfachung dar.

4.3.2.7 Abhängigkeiten zum Implementierungsmodell

Eine Modellarchitektur beschreibt auf abstrakter Ebene die allgemeinen Charakteristika einer Klasse vergleichbarer Softwarearchitekturen. Obgleich auf dieser Architekturebene im wesentlichen von Fragen der Implementierung abstrahiert wird, hat die Wahl des Implementierungsmodells trotzdem immer auch eine Rückwirkung auf die Art der Systemarchitektur. Die vorgestellte Modellarchitektur für komponentenorientierte Werkzeugarchitekturen ist beeinflußt von der Wahl der Implementierungssprache Java und den praktischen Erfahrungen, die ich bei der Programmierung in Java gesammelt habe. Einige Entwurfsentscheidungen orientieren sich bewußt an Java-typischen Konzepten wie

Interfaces, Action-Objekte, EventListener, etc. Eine Umsetzung in Java ist daher ohne großen Bruch zwischen den Abstraktionsebenen möglich (siehe folgendes Kapitel). Die Frage, inwieweit diese Modellarchitektur ohne Änderung gleichermaßen auch für die Entwicklung mit anderen Programmiersprachen und Rahmenwerken geeignet ist, liegt außerhalb dieser Arbeit und müßte noch detaillierter untersucht werden.

4.4 C2- Modellarchitektur

In diesem Abschnitt wird die an der Universität Irvine in Kalifornien entwickelte Modellarchitektur C2 vorgestellt [TMA+96]. C2 ist eine Modellarchitektur, die ebenfalls speziell für Anwendungssysteme mit graphischer Benutzungsoberfläche entwickelt worden ist. C2-Architekturen sind Konfigurationen von hierarchisch angeordneter Komponenten, wobei typischerweise systemnähere Komponenten unten in der Hierarchie stehen und anwendungsnähere weiter oben[13]. Im Vergleich zu der im vorigen Abschnitt vorgestellten Hierarchie von Werkzeugkomponenten hat eine C2-Hierarchie eine stärker technische Ausrichtung. Neben Komponenten stellen im C2-Stil auch Konnektoren, die Verbindungsstücke zwischen Komponenten, explizite Modellelemente dar. Die Komponenteninteraktion erfolgt auf der Basis eines asynchronen Nachrichtenaustauschs. Komponenten und Konnektoren besitzen explizite Interaktionspunkte über welche Nachrichten übertragen werden. Abbildung 18 zeigt ein Beispiel einer C2-Architektur.

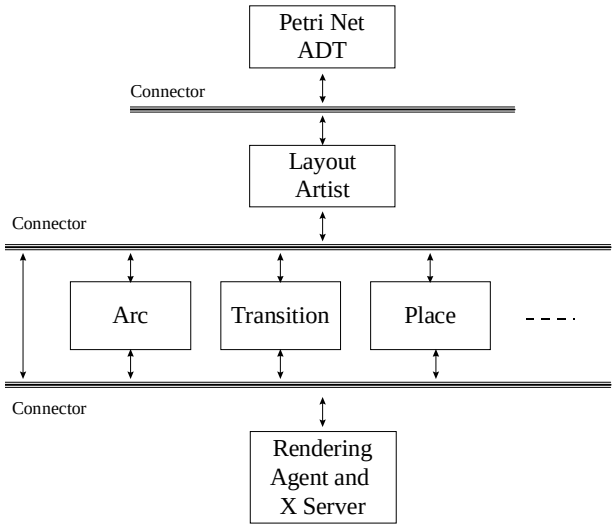

Abbildung 18: C2-Architektur eines Petri-Netz Editors

In [TMA+96] wird der C2-Stil durch fünf typische Architekturprinzipien charakterisieren:

(1) *Kontextunabhängigkeit*: eine Komponente hat kein Wissen und macht keinerlei Annahmen über die Komponenten, die in der Hierarchie unter ihr stehen. Eine Zustandsänderung einer Komponente, die für tieferliegenen Komponenten relevant sein könnte, wird prinzipiell über eine ungerichtete

[13] C2-Hierarchien sind keine strengen Baumhierarchien, sondern gerichtete Graphen, die eine Unterscheidung in *oben* und *unten* erlauben.

Benachrichtigung an alle tieferliegenden Komponenten propagiert. Taylor et al. nennen diese Eigenschaft *Substrate Independence*.

(2) *Nachrichtenbasierte Kommunikation*: sämtliche Komponenteninteraktion basiert auf asynchronem Nachrichtenaustausch. In C2 gibt es zwei Arten von Nachrichten: an tieferliegene Komponenten werden ausschließlich Ereignisbenachrichtigungen (Notifications) über Zustandsänderungen gesendet und an höhergelegene Komponenten konkrete Aufrufe zur Ausführung von Aktionen (Requests).

(3) *Multiple Threads*: jede Komponente hat ihre eigene Kontrollflußsteuerung (Thread of Control).

(4) *Keine Annahmen über gemeinsamen Adressraum*: C2-Komponenten dürfen prinzipiell nicht davon ausgehen, daß sie mit anderen Komponenten im gleichen Adreßraum agieren. Diese Einschränkung erlaubt den flexiblen Einsatz von heterogenen Komponenten auch in verteilten Systemen.

(5) *Trennung von Implementation und Architektur*: im C2-Modell wird streng zwischen der Architektur und der Implementation der Komponenten getrennt. Obwohl der Architekturstil beispielsweise verbietet, eine Architektur davon abhängig zu machen, daß ihre Komponenten sich immer einen gemeinsamen Adressraum teilen, so können in einer konkreten Implementierung die Komponenten aus Performanzüberlegungen durchaus im selben Adreßraum laufen.

4.4.1 C2-Komponenten

Alle C2-Komponenten besitzen einen Zustand und eine eigene Kontrollfluß-steuerung. Entsprechend der hierarchischen Struktur von C2-Architekturen werden die Interaktionspunkte einer Komponente in zwei Gruppen eingeteilt: (1) die sog. *Top Domain* spezifiziert Interaktionspunkte, über die Nachrichten mit hierarchisch übergeordneten Komponenten ausgetauscht werden und (2) die *Bottom Domain*, spezifiziert den Nachrichtenaustauch mit hierarchisch tiefer liegenden Komponenten. Während der Nachrichtenfluß aufwärts in der Hierarchie ausschließliche über Requests abläuft, werden hierarchieabwärts nur Ereignisbenachrichtigungen (Notifications) eingesetzt (Abbildung 19).

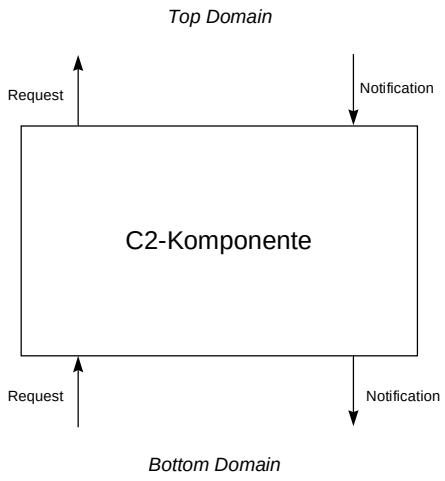

Abbildung 19: C2-Komponente

Über einen Request wird an eine Komponente eine konkrete Anforderungen formuliert. Wie auch in der obigen Modellarchitektur der Werkzeugkomponenten stellen in C2 Requests einen generischen Kommunikationsmechanismus dar. Anwendungsfachliche Requests können nur unter einem allgemeinen Request-Typ spezifiziert werden und ermöglichen lediglich dynamische Typprüfungen. Notifications haben einen reinen benachrichtigenden Charakter und informieren tieferliegende Komponenten über den Komponentenzustand. Die strikte Trennung dieser beiden Nachrichtentypen ist die Grundlage zur Umsetzung des C2-Prinzips der Substratunabhängigkeit.

Wie die Beispielimplementierungen der C2-Autoren zeigen, haben die oben aufgeführten Prinzipien des C2-Stils in verschiedener Weise Auswirkungen auf die Komponenteninteraktion:

- Die Ausführung einer Operation in einer C2-Komponente läßt sich nur über eine Request-Nachricht erreichen. Direkte, statisch typisierte Operationsaufrufe sind nicht möglich.
- Auch Funktionsaufrufe müssen über den Request-Mechanismus realisiert werden. Den Funktionswert erhält die aufrufende Komponente über eine asynchrone Ereignisbenachrichtigung.
- Da in C2-Architekturen keine Annahmen über einen gemeinsamen Adreßraum gemacht werden, dürften in Nachrichten genaugenommen niemals Referenzen von Implementationsobjekten übermittelt werden. Die gesamte Kommunikation muß wertebasiert (vgl. [MacLennan82]) erfolgen, damit die Verlegung einer Komponente in einen anderen Adreßraum nicht zum Zusammenbruch der gesamten Systemarchitektur führt. Vor allem auch bei der Realisierung der graphischen Benutzungsoberfläche läßt sich diese Anforderung nur sehr schwer verwirklichen.

Neben dem allgemeinen Systemarchitekturmodell beinhaltet das C2-Modell auch eine abstrakte Beschreibung der internen Komponentenarchitektur (Abbildung 20). Diese interne Architektur ist immer noch unabhängig von der konkreten Komponentenimplementation und kann gemäß Moriconi et al. als eine Architekturverfeinerung verstanden werden (vgl. [MQR95]).

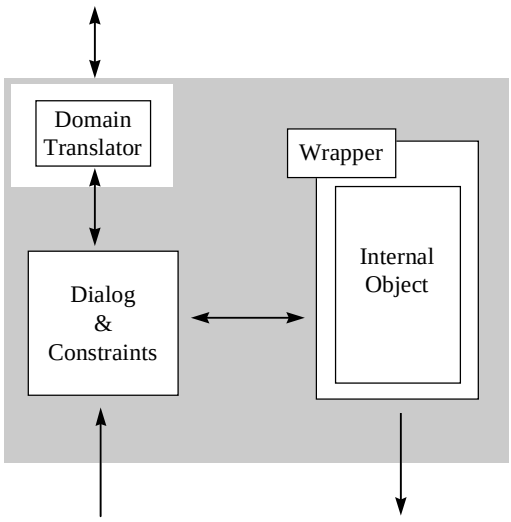

Abbildung 20: Interne Architektur einer C2-Komponente

Das dargestellte interne Objekt kann beliebig komplex sein und besteht technisch in der Regel aus mehreren Implementationsobjekten. Der Zugriff auf dieses Objekt erfolgt ausschließlich über die technische Komponente *Dialog&Constraints*. Diese technische Komponente beinhaltet die separate Kontrollflußsteuerung der C2-Komponente und reagiert auf einlaufende Requests der Bottom Domain und auf Notifications der Top Domain. Um die Wiederverwendbarkeit einer C2-Komponente zu erhöhen, kann sie einen sog. Domain Translator besitzen, der eine Anpassung an die Schnittstelle des Konnektors der Top Domain vornimmt. Ein Wrapper, der das interne Objekt umhüllt, hat die Aufgabe das Ergebnis eines Operationsaufrufs in eine Notification zu übersetzen und über die Bottom Domain zu versenden.

4.4.2 Konnektoren

Konnektoren verbinden mehrere C2-Komponenten. Komponenten in C2 interagieren niemals direkt miteinander, sondern immer über einen dazwischengeschalteten Konnektor. An einen C2-Konnektor kann eine beliebige Menge von Komponenten gehängt werden. Die hauptsächliche Aufgabe eines Konnektors ist es, für die richtige Weiterleitung (routing) und Verbreitung (broadcasting) von Nachrichten zu sorgen. Darüber hinaus können verschiedene Konnektoren unterschiedliche Filter- und Verbreitungsstrategien realisieren. Beispiele hierfür sind:

(1) *Keine Filterung*: Der Konnektor leitet jede Nachricht in der entsprechenden Richtung (Requests aufwärts, Notifications abwärts) an alle angeschlossenen Komponenten weiter.

(2) *Notification-Filterung*: Notifications werden nur an die Komponenten weitergeleitet, die sich hierfür vorher explizit registriert haben.

(3) *Bedingte Weiterleitung*: Ein Konnektor kann eine Reihenfolge definieren, in der er eine Notification an die angeschlossenen Komponenten nur solange weiterleitet, bis eine bestimmte Bedingung eingetreten ist. Beispielsweise könnte die Auslieferung abgebrochen werden, sobald die erste Komponente darauf regiert hat.

Wie die Komponenten besitzen auch Konnektoren jeweils eine Top und eine Bottom Domain. In einer C2-Architektur läßt sich die Top Domain einer Komponente nur an die Bottom Domain eines Konnektors anschließen und die Bottom Domain einer Komponente nur an die Top Domain eines Konnektors. Der C2-Stil erlaubt es jedoch auch, daß die Bottom Domain eines Konnektors direkt mit der Top Domain eines anderen Konnektors verbunden ist, ohne daß eine Komponente dazwischen hängt. Konnektoren stellen damit eine Art Softwarebus dar, wie er sich beispielsweise mit CORBA [OMG95] realisieren ließe.

4.5 Vergleich

Während die in Kapitel 4.3 vorgestellte Modellarchitektur speziell Bezug auf die typischen Eigenschaften von Softwarewerkzeugen nimmt, stellt der C2-Stil einen allgemeineren Ansatz dar. Obgleich beide Ansätze auf hierarchischen Komponentenstrukturen basieren, unterscheiden sie sich im Kern doch entscheidend. C2-Komponenten haben eine sehr viel stärker technische Ausrichtung. Eine C2-Komponentenarchitektur ist weniger an der anwendungsfachlichen Aufgabe, die mit dem System bearbeitet werden soll, orientiert, sondern mehr an den zur

Realisierung erforderlichen technischen Komponenten. Die Modellarchitektur der Werkzeugkomponentenhierarchien in Kapitel 4.3 modelliert eine Komponente nicht in Begriffen wie systemnah und anwendungsnah. Eine Werkzeugkomponente enthält sowohl systemnahme als auch anwendungsnahe Bestandteile. Die hierarchische Ordnung von C2-Komponenten ähnelt eher der Trennung von Interaktionskomponente (= weiter unten in der Hierarchie) und Funktionskomponenten (= weiter oben in der Hierarchie).[14]

Ein sich hieraus ableitender wesentlicher Unterschied der beiden Modelle zeigt sich in der Art, wie Komponenteninteraktionen modelliert werden. Der Mechanismus des Nachrichtenaustauschs in C2 ist ein generischer Kommunikationsmechanismus auf einer sehr niedrigen Abstraktionsebene. Mit Requests und Notifications lassen sich alle erdenklichen Interaktionsprotokolle realisieren. Diese Allgemeingültigkeit hat aber den Nachteil, daß die typischen Interaktionsmuster zwischen den Komponenten im Architekturmodell nicht direkt ersichtlich sind. Die Modellarchitektur der Werkzeugkomponenten beschreibt die Komponenteninteraktion dagegen auf einem höheren Niveau. Hier werden die Rollen, die eine Komponente in einem speziellen Interaktionskontext spielt, explizit modelliert. Die Verantwortlichkeiten einer Komponente lassen sich in diesem Modell dadurch sehr klar darstellen. Ein weiterer Unterschied ist, daß C2 völlig auf statisch getypte Operationsaufrufe verzichtet. Ähnlich in beiden Modellen ist jedoch, daß interagierende Komponenten immer asymmetrisch miteinander gekoppelt sind. Die substratunabhängigkeit in C2 ist vergleichbar mit der Unabhängigkeit einer Werkzeugkomponente von ihrer übergeordneten Manager-Komponente.

Die Modellarchitektur der Werkzeugkomponenten umfaßt weitaus weniger mögliche Architekturen, als der C2-Stil. C2-Architekturen unterliegen weniger Einschränkungen als Werkzeugkomponentenarchitekturen. Die Anforderungen an eine C2-Komponente, damit sie in einem System mit anderen C2-Komponenten interagieren kann, sind sehr gering. Die Komponente muß lediglich generische Requests und Notifications senden und empfangen können. An das semantische Verhalten stellt der C2-Stil keine expliziten Anforderungen. Eine weitere Einschränkung bei Werkzeugkomponenten, die jedoch ihre Verwendung sehr vereinfacht, ist, daß alle Komponenten eines Werkzeugs im gleichen Prozeß- und Adreßraum ablaufen. Hierdurch werden einige Probleme des C2-Stils von vornherein vermieden. So ist es beispielsweise bei Werkzeugkomponenten weniger kritisch Referenzen auszutauschen als bei C2-Komponenten.

[14] Begriffe wie "höher in der Hierarchie" und "tiefer in der Hierarchie" sind beim Vergleich der beiden Modelle etwas irreführend, da die zwei Hierarchien eine gegensätzliche Ausrichtungen besitzen. Die hauptsächliche Aufrufrichtung zwischen Komponenten ist im C2-Modell von unten nach oben, im Werkzeugkomponentenhierarchien dagegen von oben nach unten.

5 Implementierung in Java

Modelle der Architekturebene stellen noch keine ablauffähigen Systeme dar. Für die Realisierung eines Softwaresystems ist deshalb immer eine Abbildung des abstrakten Architekturmodells auf ein konkretes Implementierungsmodell erforderlich. Bei objektorientiert modellierte Architekturen ist die Abbildung dank existierender objektorientierter Programmiersprachen recht einfach. Das fachliche Klassenkonzept läßt sich direkt auf das technische Klassenkonzept einer objektorientierten Programmiersprache abbilden. Jede fachlich modellierte Klasse wird durch eine programmiersprachliche Klasse realisiert.

Bei komponentenorientierter Modellierung gestaltet sich die Abbildung jedoch schwieriger. Eine Komponentenklasse läßt sich im allgemeinen gerade nicht durch eine einzelne programmiersprachliche Klasse realisieren, sondern nur durch einen Klassenverbund bzw. durch ein ganzes Rahmenwerk. Eine Komponente des abstrakten Modells wird im laufenden System nicht durch eine, sondern durch mehrere, miteinander kommunizierende Objekte realisiert.

Prinzipiell lassen sich komponentenorientiert entworfene Werkzeuge auf beliebige Implementierungsmodelle abbilden und nicht bloß auf objektorientierte. Je größer aber die konzeptuellen Unterschiede zwischen der Architekturebene und der Implementierungsebene sind, desto schwieriger gestaltet sich hierbei jedoch die praktische Umsetzung. Der Übergang zwischen Modellebenen beinhaltet immer auch einen konzeptionellen Bruch, den es möglichst klein zu halten gilt. Zwischen Komponenten und Objekten existieren eine Reihe von strukturellen Ähnlichkeiten (siehe Metamodelle, Seite 33). Hierdurch erweist sich die Realisierung einer komponentenorientierten Architektur in einer objektorientierten Programmiersprache generell einfacher als mit anderen Implementierungsmodellen. Wie Komponenten, so interagieren auch Objekte in einem System ausschließlich über definierte Schnittstellen, die den internen Objektzustand vor externen Zugriffen schützen. Ebenso wie eine Komponente besitzt auch ein Objekt immer eine eindeutige Identität, die sie von anderen Objekten unterscheidbar macht.

Im praktischen Teil dieser Arbeit wurden verschiedene Abbildungsmöglichkeiten von Werkzeugkomponenten in der Programmiersprache Java untersucht. Dieses Kapitel diskutiert die daraus gewonnenen Erkenntnisse und beleuchten unterschiedliche Aspekte der Komponentenimplementierung.

5.1 Werkzeugkomponenten in Java

Werkzeugkomponenten sind die grundlegende Bausteine der Modellarchitektur für komponentenorientierte Werkzeugkonstruktion. Im zugrundegelegten Komponenten-Metamodell (Kapitel 4.3.1, Seite 33) werden Komponenten im wesentlichen durch eine Liste von Call-In-Schnittstellen, einer Liste von Call-Out-Schnittstellen sowie einer Menge von Komponenteneigenschaften (Properties) definiert. Bei der Abbildung auf ein Implementierungsmodell in der Sprache Java müssen daher folgende Fragen untersucht werden:

(1) Wie lassen sich Call-In- und Call-Out-Schnittstellen in Java abbilden?

(2) Wie können Komponentenklassen mit Java-Klassen implementiert werden?

(3) Wie lassen sich Komponenteneigenschaften in Java realisieren?

5.1.1 Komponentenschnittstellen

Das Architekturmodell spezifiziert die Schnittstellen einer Werkzeugkomponente explizit durch einen Namen und ein Protokoll. Für eine einfache und klare Implementierungsabbildung empfiehlt es sich, auch im Implementierungsmodell die Schnittstellen, daß heißt insbesondere auch die Typen der Komponentenschnittstellen, als separate softwaretechnische Elemente zu realisieren.

Eine Komponentenschnittstelle definiert eine Menge von Operationen sowie ein Protokoll für deren Verwendung. Während sich die Operationen durch einen Datentyp hinreichend definieren lassen, wird die eindeutige Spezifikation des Protokolls in den meisten objektorientierten Sprachen nicht direkt unterstützt.[15]

In der Programmiersprache Java lassen sich Schnittstellentypen prinzipiell als Java *Interfaces* oder als Java *Classes* realisieren. Während in Java das Sprachkonstrukt Interface eine reine Typbeschreibung darstellt, beinhaltet eine Klasse neben ihrem Typ immer auch eine Form von Implementation. Da es in unterschiedlichen Kontexten durchaus sinnvoll sein kann, eine Komponente mit verschiedenen Implementationen ihrer Schnittstellen auszustatten, eignet sich die Abbildung von Komponentenschnittstellen auf Java-Klassen nur bedingt. So ist es zum Beispiel denkbar, je nachdem, ob ein Werkzeug in einem leichtgewichtigen Applet-Umfeld oder als eigenständige Applikation auf einem Desktop eingesetzt werden soll, Schnittstellenimplementationen zu verwenden, die zwar in ihrer Funktionalität identisch sind, sich aber in ihrer Geschwindigkeit oder ihrem Ressourcenbedarf unterscheiden. Wird ein Schnittstellentyp auf eine Java-Klasse abgebildet, gestaltet sich diese erwünschte Flexibilität schwierig, da immer auch die Klassenimplementation weitervererbt wird. Darüber hinaus wird, durch die Einschränkung auf Einfachvererbung, in Java der Aufbau verständlicher Klassenstrukturen erschwert, wenn sowohl Komponententypen als auch Schnittstellentypen als Java-Klassen realisiert werden.

Die Abbildung einer Komponentenschnittstelle auf ein Java-Interface erscheint hier als die geeignetere Lösung. Sie erlaubt eine strikte Trennung zwischen dem Schnittstellentyp, wie er von anderen Komponenten benötigt wird und der Schnittstellenimplementation. Hierdurch ist es sehr viel einfacher für denselben Schnittstellentyp unterschiedliche Implementationen bereitzustellen. In Kapitel 5.2 werden die Java-Interfaces der Basisschnittstellen von Werkzeugkomponenten im einzelnen betrachtet.

Im Architekturmodell werden neben den Call-In-Schnittstellen auch die Call-Out-Schnittstellen der Komponenten explizit modelliert. In der Implementierung stellt eine Call-Out-Schnittstelle eine Referenz dar, die zur Laufzeit an die typkonforme Call-In-Schnittstellen einer anderen Komponente gebunden sein muß. Call-Out-Schnittstellen manifestieren sich in der Implementierung daher lediglich durch Verknüpfungsoperationen in der jeweiligen Komponentenklasse. Es empfiehlt

[15] Weitergehende Ansätze, die darauf abzielen in einer Schnittstelle mehr als nur die Signaturen der Operationen zu definieren, finden sich beispielsweise in den Vor- und Nachbedingungen des Vertragskonzepts der Programmiersprache Eiffel [Meyer90].

sich die Verknüpfungsoperationen von Call-Out-Schnittstellen nach einem einheitlichen Namensschema aufzubauen. Die Namenskonventionen, die im Rahmen dieser Diplomarbeit verwendet wurden, beschreibt exemplarisch das folgende Beispiel der nextHandler-Schnittstelle:

Die Call-Out-Schnittstelle `nextHandler` mit dem Typ `RequestHandler` wird durch die folgen drei Operationen realisiert

```
public void connectNextHandler(RequestHandler i);
public void disconnectNextHandler();
public RequestHandler getConnectionNextHandler();
```

5.1.2 Komponentenklasse

Komponentenklassen implementieren Komponentenschnittstellen. Werden die Komponentenschnittstellen durch Java-Interfaces realisiert, so muß es Java-Klassen geben, die diese Interfaces implementieren. Wie bereits aufgeführt, müssen Komponentenklassen aber nicht zwangsläufig eins-zu-eins auf Java-Klassen abgebildet werden. Vielmehr wird in der Regel eine Komponentenklasse durch ein ganzes Klassenkonglomerat, bzw. ein Rahmenwerk realisiert. Für die Realisierung einer Komponentenklasse als Ganzes ergeben sich hieraus verschiedene Ansätze, die zu jeweils unterschiedlichen internen Komponenten-architekturen führen. Im folgenden sollen die Konstruktionsvarianten näher betrachtet werden.

5.1.2.1 Zentrale Schnittstellenimplementation

Eine geradlinige Implementierungsabbildung ergibt sich, wenn alle Schnittstellen direkt von einer zentralen Komponentenklasse implementiert werden. Im Unterschied zu Java-Klassen, bei denen es nur Einfachvererbung gibt, ist die Mehrfachvererbung und die Implementation mehrerer Interfaces in Java durchaus möglich. Die zentrale, implementierende Komponentenklasse agiert in diesem Fall als eine Fassade für die gesamte Werkzeugkomponente (vgl. Facade-Muster in [GHJV95]). Hierbei wird die Komponentenklasse im allgemeinen nicht alle Schnittstellenoperationen vollständig selbst umsetzen, sondern vielmehr wird sie die Aufrufe an interne Klassen der Komponente weiterdelegieren. Abbildung 21 zeigt beispielhaft die Architektur einer in dieser Weise konstruierten Editorkomponente.

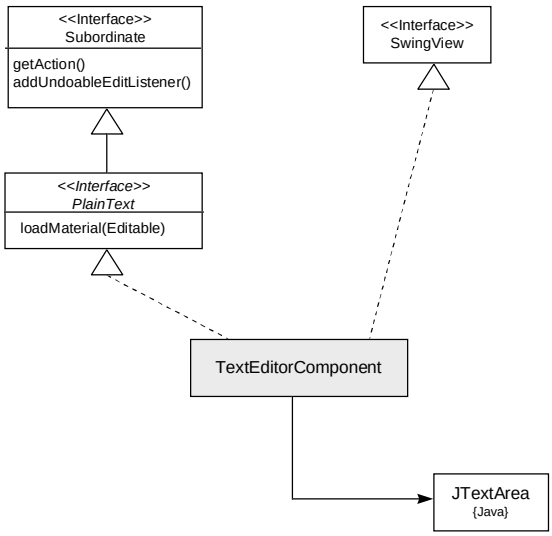

Abbildung 21: Zentrale Schnittstellenimplementation

Das Beispiel deutet an, in welchen Fällen diese Struktur sinnvoll eingesetzt werden kann: Ist die Funktionalität einer Werkzeugkomponente zum großen Teil vergleichbar mit der einer vorhandenen technischen Komponente, so bietet es sich an, daß die Komponentenimplementation sich auf dieser fertigen Lösung abstützt. Beispielsweise beinhaltet das Swing-Rahmenwerk neben einfachen GUI-Elementen, wie Schaltflächen oder Editierfelder, durchaus auch sehr komplexe Komponenten wie Tabellen, Baumdarstellungen oder verschiedene Editoren, die sich häufig als Basisbausteine für die Komponentenimplementation gut eignen. Im obigen Beispiel besitzt die Swing-Klasse `JTextArea` zum großen Teil die Funktionalität, die die Editorkomponente des Kalenderwerkzeugs realisieren soll. Eine einfache Implementierung ergibt sich dadurch, daß die Klasse `TextEditorComponent` in der Rolle eines Objekt-Adapters, entsprechend dem Adapter-Muster [GHJV95], die Schnittstellenoperationen direkt auf die Operationen der Klasse `JTextArea` abbildet.

In ähnlicher Weise ist diese Struktur auch geeignet, existierende Altsysteme als Werkzeugkomponenten zu verpacken.

5.1.2.2 Entkoppelte Schnittstellenimplementation

Ein anderer Ansatz ist die Implementation der Komponentenschnittstellen durch jeweils separate Schnittstellenklassen. Jede Implementationsklasse realisiert in großer Eigenständigkeit die von ihrer Schnittstelle definierte Rolle. Auf diese Weise läßt sich beispielsweise eine Werkzeugkomponente entsprechend der im WAM-Ansatz verwendeten Trennung von Funktion und Interaktion aufbauen (siehe S. 27). Abbildung 22 zeigt wie eine auf der FK-IAK-Trennung basierende Komponentenimplementation der Editorkomponente aussieht. Wie bereits erwähnt, wird das Muster der Trennung von Funktion und Interaktion im komponentenorientierten Ansatz dieser Diplomarbeit als *eine von verschiedenen* Möglichkeiten der Komponentenimplementation betrachtet.

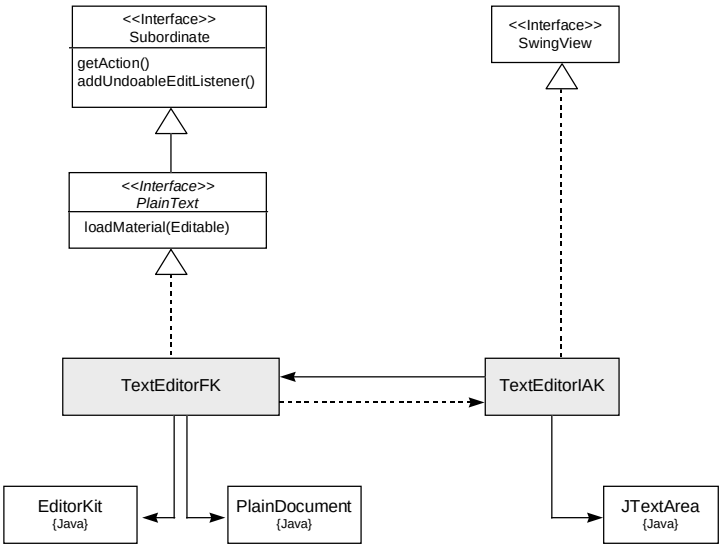

Abbildung 22: FK-IAK-Implementation

Während die dienstleistungsspezifische Schnittstelle PlainText von der Funktionskomponente implementiert wird, realisiert die IAK die View-Schnittstelle. Beide Implementationsklassen stützen sich wiederum auf existierenden Swing-Klassen ab. Von Vorteil erweist es sich hierbei, daß die GUI-Klassen des Swing-Rahmenwerks auf einer Variante des Model-View-Controller Schemas basieren. Hierdurch ist es so ohne größeren Aufwand möglich, in einer Implementation die darstellenden Bestandteile von den fachlichen zu trennen. Das GUI-Modell bildet die Implementationsgrundlage für die FK und der View-Controller-Teil die Basis für die IAK.

Es fällt bei obigem Konstruktionsansatz auf, daß keine Klasse existiert, welche die Editorkomponente als Ganzes repräsentiert. Erst FK und IAK zusammen sind ein vollständiges Abbild einer Werkzeugkomponente der Architekturebene. Eine zum Architekturmodell strukturähnlichere Implementierungsabbildung zeigt Abbildung 23.

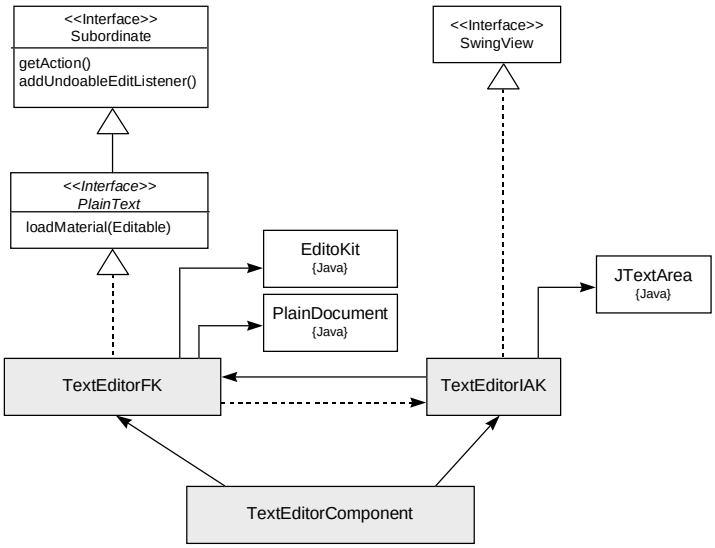

Abbildung 23: FK-IAK-Implementation mit Komponentenklasse

Die Klasse `TextEditorComponent` hat hier vor allem die Aufgabe, den Erzeugungsprozeß für die Funktions- und die Interaktionskomponente zu konfigurieren und zu steuern und die Schnittstellen dem Umfeld zugänglich zu machen. An der Interaktion innerhalb der Werkzeugkomponentenhierarchie ist die Klasse nicht beteiligt.

Solange eine Werkzeugkomponente lediglich eine Subordinate- und eine View-Schnittstelle besitzt, läßt sich die Komponente prinzipiell immer auch mit einer zentralen Schnittstellenimplementation realisieren. Bietet eine Werkzeugkomponente jedoch mehrere alternative Schnittstellen einer Art, so muß die Komponentenimplementation zwangsläufig auf einer separaten Schnittstellenimplementation basieren. Als Beispiel soll hier eine erweiterte Version der Editorkomponente betrachtet werden, die neben einer einfachen PlainText-Schnittstelle alternativ auch unter einer StyledText-Schnittstelle in ein Werkzeug eingebunden werden kann (Abbildung 24). Die StyledText-Schnittstelle ermöglicht es einen Text mit zusätzlichen Formatierungsinformationen auszustatten.

Da sowohl PlainText als auch StyledText Spezialisierungen der allgemeinen Subordinate-Schnittstelle sind, ist eine Implementation der Schnittstellen in separaten Klassen nötig, um in einer Editorkomponente diese beiden Schnittstellen auseinanderhalten zu können. Nur so kann beispielsweise die Operation `getActions()` des Subordinate-Interface je nach Schnittstelle eine unterschiedliche Menge von Aktionsobjekten zurückliefern.

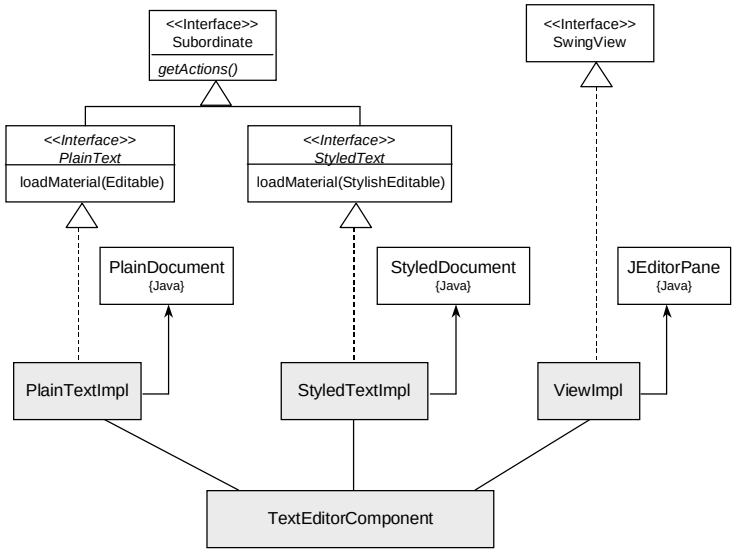

Abbildung 24: Komponentenklasse als Mediator

Die dargestellte Architektur unterscheidet sich aber noch in einer weitereren Hinsicht von der Architektur in Abbildung 23. Anders als bei der FK-IAK-Architektur existieren in Abbildung 24 keine direkten Beziehungen zwischen den Schnittstellenklassen. Die einzelnen Implementationen kennen sich nicht gegenseitig und arbeiten im wesentlichen unabhängig voneinander. Da aber trotzdem immer eine Art des Austauschs zwischen den Schnittstellenimplementationen nötig ist, übernimmt die Klasse `TextEditorComponent` die Rolle eines Mediators entsprechend dem Mediator-Muster [GHJV95].

Bei der entkoppelten Schnittstellenimplementation muß die Frage geklärt werden, wie ein Schnittstellenobjekt zur Laufzeit verfügbar gemacht wird, damit es in einer Komponentenhierarchie verwendet werden kann. Während sich bei der zentralen Implementierung Exemplare der Komponentenklasse polymorph direkt unter dem Typ einer Schnittstelle verwenden lassen, geht dieses bei der separaten Schnittstellenimplementation nicht. Eine einfache Lösung um Schnittstellenimplementationen erreichbar zu machen ist, für jede Schnittstelle in der Komponentenklasse eine explizite Zugriffsoperation einzurichten. Die Komponentenklasse `TextEditorComponent` aus den obigen Beispielen könnte entsprechend die folgenden drei Operationen besitzen:

```
public PlainText getPlainTextInterface();
public StyledText getStyledTextInterface();
public SwingView getSwingViewInterface();
```

Einen alternativen Ansatz für Verwaltung von Schnittstellenimplementationen beschreibt das Muster *Extension Object* [Gamma98]. Hier werden alle Schnittstellenobjekte über eine einzelne, generische Zugriffsoperation zugänglich gemacht. Bekannt ist diese Struktur u.a. aus Microsofts COM-Architektur (siehe Kapitel 4.1). Um auf die Schnittstellenimplementation eines COM-Objektes zuzugreifen muß ein Klient sich an die `IUnknown`-Schnittstelle wenden und das Schnittstellenobjekt über die Operation `queryInterface()` ermitteln. Anschließend ist eine expliziten Typumwandlung (Down Cast) erforderlich um die erhaltene Referenz unter dem gewünschten konkreten Schnittstellentyp verwenden zu können. Wie bei Werkzeugkomponenten eine Architektur nach dem Muster Extension Object aussieht, zeigt Abbildung 25.

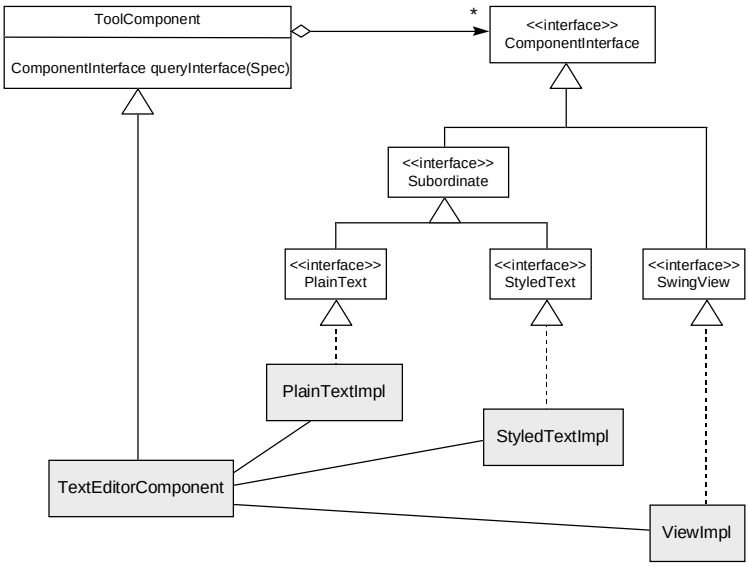

Abbildung 25: Extension Object

Diese Struktur bietet einen hohen Grad an Flexibilität. Das Muster erlaubt es Komponenten flexibel für einen konkreten Einsatzkontext zu konfigurieren und jeweils nur mit den Schnittstellenimplementationen auszustatten, die auch tatsächlich benötigt werden. Auch müssen konkrete Schnittstellenimplementationsklassen erst bei Bedarf in das Laufzeitsystem geladen werden, wodurch der gesamte Erzeugungsprozeß des Werkzeugs beschleunigt wird. Darüber hinaus

erlaubt diese Struktur die Erweiterung bzw. Spezialisierung einer Werkzeugkomponente ohne, daß dafür von der Komponentenklasse geerbt werden muß. Eine speziellere Komponente muß lediglich zusätzliche Schnittstellenimplementationen über die `queryInterface()`-Operation verfügbar machen. Es ist sogar prinzipiell möglich, einer Werkzeugkomponente je nach Bedarf erst zur Laufzeit zusätzliche Schnittstellen hinzuzufügen oder während der Laufzeit zu entfernen. Im Kontext von Werkzeugkomponenten ist diese Dynamik aber i.d.R. nicht gewünscht, da dies das weiter unten (Seite 76) behandelte Prinzip der Architekturkonformität verletzen würde.

Eine Konsequenz dieser flexiblen Schnittstellenverwaltung ist, daß ein Teil der statischen Typprüfung verloren geht. Einer Komponentenklasse ist an ihrer Klassenschnittstelle nicht anzusehen welche Schnittstellen die Komponente als Ganzes unterstützt. Setzt ein Klient bei einer Werkzeugkomponente eine bestimmte Schnittstelle voraus, die aber von der Komponente gar nicht unterstützt wird, so fällt diese Diskrepanz erst zur Laufzeit auf.

5.1.2.3 Aufgabenzentrierte Implementation

Wie [Züllighoven98] aufführt, zeichnet sich ein interaktives Softwarewerkzeugs durch die Aspekte Funktion, Präsentation und Handhabung aus. Im WAM-Ansatz wird ein Werkzeug deshalb jeweils aus den zwei technische Klassen FK und IAK konstruiert. Während die gesamte Werkzeugfunktionalität in der FK implementiert ist, realisiert die IAK sowohl die Präsentation, als auch die Werkzeughandhabung. Diese Trennung ermöglicht Änderungen am Werkzeug, die sich ausschließlich auf die Präsentation oder Handhabung beziehen, ohne Auswirkung auf die Funktionskomponente vorzunehmen.

Der WAM-Konstruktionsansatz geht davon aus, daß generell die Handhabung und die Präsentation untrennbar miteinander verbunden sind. In anspruchsvollen Programmsystemen kann diese enge Kopplung eine beachtliche Komplexität der IAK bedingen. Im Gegensatz zu WAM differenziert die "Mutter aller GUI-Architekturen", das aus Smalltalk bekannte Model-View-Controller-Muster (MVC) [KP88], beim interaktiven Teil eines Systems konsequent zwischen der reinen Präsentation (View) und der Handhabung (Controller).

Die FK-IAK-Trennung ist im wesentlichen technisch motiviert. Sie berücksichtigt nicht die konkreten fachlichen Aufgaben und die Handhabung eines Werkzeugs bei deren Bearbeitung. Änderungswünsche ergeben sich aber häufig gerade durch die Arbeit mit einem Werkzeug und orientieren sich direkt an den zu bearbeitenden fachlichen Aufgaben. Aus diesen Überlegungen heraus haben Robert Martin et al. eine Architektur entwickelt, die sie *Taskmaster* nennen und die sich stärker an den zu bearbeitenden Aufgaben anlehnt [MNR97]. Die Taskmaster-Struktur eignet sich auch zur Implementierung von Werkzeugkomponenten.

Im Kern dieses Musters wird die reine Präsentation von der Handhabung getrennt. Anstatt einer zentralen Controller-Klasse wird aber jede von dem System zu bearbeitende Aufgabe in einer separaten Klasse, die *Task* genannt wird, implementiert. Fachlich bedingte Programmänderungen betreffen dadurch meist jeweils nur eine Task-Klasse und lassen sich dadurch besser lokal einschränken. Eine Task ist eine aus mehreren Einzelschritten bestehende Transaktion, die, einmal angestoßen, durchläuft, bis sie ihre Aufgabe erfüllt hat oder bis sie explizit abgebrochen wird. Der Start einer Task erfolgt durch eine explizite

Benutzeraktion, beispielsweise einem Mausklick oder einem Tastenanschlag. Während der Bearbeitung kann es zu mehr oder weniger ausgeprägten Dialogen mit dem Benutzer kommen. Ein interaktives Programm besitzt i.d.R. mehrere Tasks, die häufig auch unabhängig voneinander ablaufen können.

Abbildung 26 zeigt schematisch eine aufgabenzentrierte Werkzeugkomponenten-architektur.

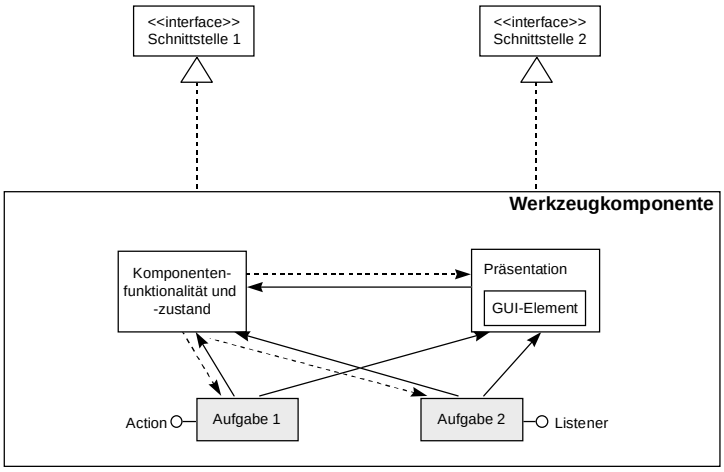

Abbildung 26: Aufgabenzentrierte Komponentenarchitektur

Neben den Task-Klassen enthält die Werkzeugkomponente eine technische Komponente für die Oberflächenpräsentation sowie eine Komponente, die den funktionalen Kern und den Komponentenzustand, das Komponentengedächtnis, repräsentiert. Entsprechend dem MVC-Muster kann diese funktionale Komponente auch als das Modell einer Werkzeugkomponente bezeichnet werden. Eine Task ruft direkt Operationen der funktionalen und der darstellenden Komponenten auf. Über eine Änderung am Komponentenzustand kann sich eine Task über den Ereignismechanismus benachrichtigen lassen. Im Gegensatz zu einer IAK kapselt die Oberflächenkomponente jedoch nur die reine Präsentation. Die Handhabung ist Bestandteil der einzelnen Tasks. Damit eine Task angestartet werden kann, implementiert sie ein oder mehrere Listener-Interfaces. Ein Großteil der Komponententasks implementiert hierbei das Action-Interface und läßt sich dadurch an Menüeinträge und Schaltflächen binden.

Eine Komponententask kann als die Implementierung eines einzelnen Use Case [BJR99] bzw. einer einzelnen Handhabungsvision [Züllighoven98] verstanden werden. Technisch läßt sich eine Task mit dem Command-Muster realisieren. Wie Gamma et al. anführen, können Commands unterschiedlich *intelligent* gestaltet sein. Es ist im Einzelfall abzuwägen, ob eine Task lediglich eine Bindung zwischen einer Aktion und der ausführenden Modellkomponente darstellt oder ob die Task einen Teil der Funktionalität selbst implementiert. Im Extremfall repräsentiert die Modellkomponente nur noch den Komponentenzustand während sich sämtliche Funktionalität auf die Tasks verteilt.

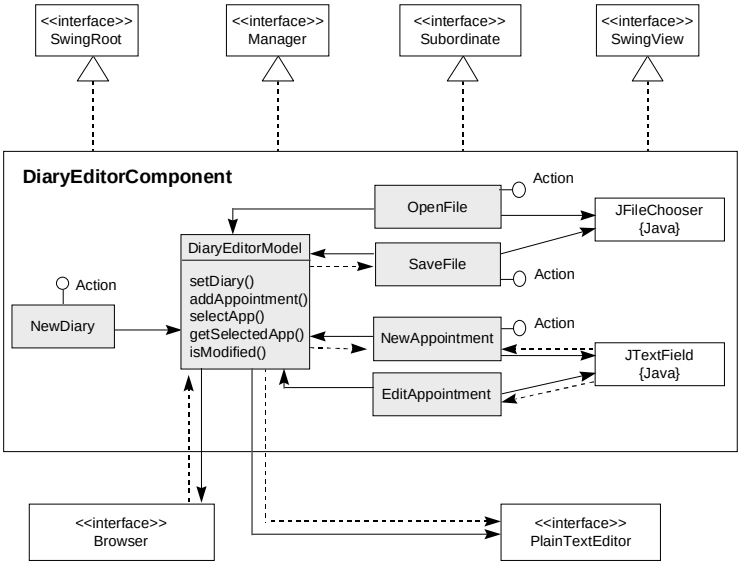

Abbildung 27: DiaryEditorComponent

Abbildung 27 zeigt eine nach den fachlichen Aufgaben ausgerichtete Architektur der Kalendereditorkomponente.[16] Die komplexe Interaktion der Komponente verteilt sich auf mehrere Task-Klassen. Die Tasks interagieren während ihrer Bearbeitung dabei unterschiedlich stark mit dem Anwender. Beispielsweise benötigt das Anlegen eines neuen Kalenders über `NewDiary` keine weitere Interaktion. Nachdem die Task einmal über die entsprechende Schaltfläche oder den Menüeintrag angestoßen worden ist, läuft Bearbeitung selbständig ab. Dagegen ist das Erfassen eines neuen Termins über `NewAppointment` eine Aufgabe mit einem intensiven Benutzerdialog, wie der folgende szenarische Ablauf verdeutlicht[17]:

> *"Der Anwender möchte einen neuen Termin in den Kalender einfügen. Er klickt mit der Maus auf die Schaltfläche "Erzeugen" worauf die aktuelle Markierung in der Terminliste erlischt und die Schreibmarke im Editierfeld für das Datum plaziert wird. Die "Erzeugen"-Schaltfläche ist deaktiviert und ausgegraut. Dann tippt er 1.77.99 und drückt die Tabulatortaste. Es ertönt ein Warnton und in der Statuszeile am unteren Fensterrand erscheint eine Meldung, daß die Eingabe ungültig sei. Im Editierfeld ist der Monat des Datum markiert. Der Anwender korrigiert das Datum zu 1.7.99 und drückt erneut die Tabulatortaste. Im nächsten Feld erfaßt er die Terminbezeichnung und drückt abermals die Tabulatortaste. Der eingegebene Termin ist jetzt in die Terminliste chronologisch einsortiert und markiert. Die Schreibmarke steht im Bereich des Texteditors und der Anwender macht ergänzende Angaben zum Termin. Die "Erzeugen"-Schaltfläche ist wieder aktiviert. "*

[16] Der Übersichtlichkeit wegen sind in dem Diagramm nicht alle Task aufgeführt.

[17] Die, diesem Szenario zugrunde gelegte Werkzeugoberfläche ist auf Seite 26 in Abbildung 7 abgebildet.

Die Task beginnt mit dem Mausklick auf die "Erzeugen"-Schaltfläche und ist erst beendet, nachdem Datum und Bezeichnung des neuen Termins korrekt erfaßt worden sind und der neue Termin in den Kalender eingefügt oder die Erfassung durch das Drücken der Escape-Taste abgebrochen worden ist. Die Eingabeprüfungen, die Fokussteuerung zwischen den Eingabefeldern, das Beobachten von Tastatur- und Mausereignissen während der Erfassung, das Deselektieren vor der Erfassung und Selektieren in der Terminliste danach sowie das Einfügen des Termins in den Kalender gehören insgesamt zu dieser Aufgabe (das Editieren des ergänzenden Textes jedoch nicht).

Am Beispiel der Terminerfassung wird deutlich, daß die fachliche Entscheidung, die Erfassung zukünftig nicht mehr direkt über die zwei Eingabefelder, sondern über einem komfortableren, mausgesteuerten Eingabedialog durchzuführen, nur eine begrenzte Task-lokale Änderung in `NewAppointment` bedeuten würde. Andere Aufgabenimplementationen blieben von dieser Änderung unberührt.

5.1.2.4 Diskussion der Konstruktionsvarianten

Die aufgezeigten Konstruktionsvarianten haben alle unterschiedliche Vor- und Nachteile. Die Stärke des komponentenorientierten Ansatzes bei der Werkzeugkonstruktion liegt nun genau darin, daß die Frage der internen Komponentenimplementierung unabhängig von der Frage der Komponenteninteraktion innerhalb des Werkzeugs beantwortet werden kann. Im Komponentenmodell liegt die Betonung auf der expliziten Modellierung der Komponentenschnittstellen und der Art der Interaktion zwischen den beteiligten Komponenten. Die Wahl der geeignetsten Komponentenimplementierung hat jedoch keine Auswirkung auf die Gesamtarchitektur eines Softwarewerkzeugs und kann sich rein an Implementierungsaspekten orientieren. Es ist entsprechend auch nicht erforderlich, daß alle Werkzeugkomponenten innerhalb einer Komponentenhierarchie in der gleichen Art und Weise konstruiert sind. Solange die vereinbarten Schnittstellenprotokolle eingehalten werden, haben auch nachträgliche Änderungen einer Implementation nur komponentenlokale Auswirkungen und beeinflussen nicht die Stabilität des Gesamtsystems.

Im folgenden werden noch einmal die wesentlichen Stärken und Schwächen der unterschiedlichen Konstruktionen gegenübergestellt.

Zentrale Schnittstellenimplementation

Stärken

- einfache Struktur, da nur eine Klasse
- einfache Kommunikation zwischen den Schnittstellenimplementationen, da alle Schnittstellen zusammen in einer Klasse realisiert sind
- statisch prüfbare Subtypbeziehungen zwischen Komponenten
- einfache Adaption an vorhandene Lösungen
- Komponentenidentität == Objektidentität des Komponentenklassenobjekts

Schwächen

- monolithisch; Änderungen der Präsentation oder Handhabung wirken sich immer auf gesamte Implementation aus

- unterstützt nicht mehrere Schnittstellen vom gleichen Typ, aber mit unterschiedlichem Verhalten

Einsatz

- einfache Komponenten
- Adaption an eine existierende Komponente oder ein existierendes System, welches schon einen Großteil der gewünschten Funktionalität besitzt

FK-IAK-Implementation

Stärken

- Funktion und Interaktion können unabhängig voneinander konstruiert und weiterentwickelt werden
- Änderungen der Interaktion wirken sich nicht auf die Funktionskomponente aus

Schwächen

- technisch motivierte Trennung, keine softwaretechnische Abbildung der anwendungsfachlichen Aufgabenbearbeitung

Einsatz

- Komponenten, bei denen Änderungen eher in der Interaktion als in der Funktion zu erwarten sind
- Komponenten mit mehreren View-Schnittstellen
- Komponenten, bei denen Handhabung und Präsentation sehr eng miteinander gekoppelt sind

Separate Schnittstellenimplementation mit Mediator

Stärken

- große Unabhängigkeit und Entkopplung zwischen den Schnittstellenimplementationen
- unterstützt auch mehrere Schnittstellen vom gleichen Typ mit unterschiedlichem Verhalten

Schwächen

- aufwendige Kommunikation zwischen Schnittstellenimplementationen über Mediator, kein direkter Austausch zwischen den Schnittstellen

Einsatz

- Komponenten mit vielen und unabhängigen Schnittstellen

Aufgabenzentrierte Interaktion

Stärken

- anwendungsfachlich bedingte Änderungen in der Bearbeitung einer Aufgabe wirken sich nur auf eine Task-Klasse aus
- einfache Realisierung von Nebenläufigkeit bei der Aufgabenbearbeitung

Schwächen

- komplexe Struktur durch viele Klassen
- nicht geeignet, wenn erst der Abschluß einer Folge von Benutzeraktionen definiert, welche Aufgabe bearbeitet wurde

Einsatz

- Komponenten mit klar abgrenzbaren Einzelaufgaben
- nebenläufige Bearbeitung von Aufgaben

Es existiert aus softwaretechnischer Sicht eine gewisse konzeptionelle Ähnlichkeit zwischen der in WAM beschriebenen Werkzeug-Material-Kopplungen über Aspekte und der Kopplung von Werkzeugkomponenten über Schnittstellen, wie sie Gegenstand dieser Diplomarbeit ist. Sowohl Aspekte als auch Komponentenschnittstellen kapseln jeweils einen Verwendungszusammenhang unter dem ein Material resp. eine Werkzeugkomponente eingesetzt werden kann. Entsprechend gibt es interessante Parallelen zwischen den in [Züllighoven98], S. 209ff diskutierten Konstruktionsansätzen für Materialaspekte und den hier angestellten Überlegungen zur Implementierung von Komponentenschnittstellen.

5.1.3 Komponenteneigenschaften

Komponenteneigenschaften (Properties) ermöglichen es eine Komponente flexible für einen konkreten Kontext zu konfigurieren. Zum großen Teil handelt es sich hierbei um Einstellungen, die sich auf das Erscheinungsbild der Komponente an der Benutzungsoberfläche auswirken, auf Anpassungen an die jeweilige Landessprache (Lokalisierung) oder auf besondere Handhabungseinstellungen wie Tastaturkürzel (Shortcuts) zur schnelleren Navigation durch Menübäume.

Nicht alle Konfigurationseinstellungen sollten fest im Programmcode „verdrahtet“ sein. Dieses würde eine Komponente unnötig inflexible gegenüber Anpassungen machen. So ist es beispielsweise nicht praktikabel, wenn die Unterstützung einer neuen Sprache bedeuten würde, daß die Komponente neu programmiert werden müßte. Die Einstellung, ob auf einer Schaltfläche *Cancel* oder *Abbruch* steht, ist für die Programmlogik ohne Belang und rechtfertigt keine Programmänderung. Aus diesem Grund ist es bei der Konstruktion von GUI-Anwendungssystemen zur gängigen Praxis geworden, die konfigurierbaren und für die Ablauflogik nicht relevanten Einstellungen in separaten Ressourcedateien abzulegen. Diese externen Ressourcedateien werden erst zur Laufzeit vom Programm eingelesen und interpretiert. [18]

Konfigurationseinstellungen, die typischerweise über Ressourcedateien gemacht werden, sind:

- Beschriftungen von Interaktionselementen wie Schaltflächen oder Menüeinträge
- allgemeine sprachabhängige Texte der Benutzungsoberfläche

[18] Klassische Anwendungen von Ressourcedateien aus dem Nicht-Java-Umfeld finden sich beispielsweise in den GUI-Rahmenwerken von Microsoft (MFC) oder Star Division (StarView) [Krüger93] [StarDivision93].

- Meldungstexte bei Fehlermeldungen
- Icons und Bilder an der Benutzungsoberfläche
- Tastaturkürzel zur schnelleren Navigation (Shortcuts, Accelerators)
- Aufbau der statischen Menüs
- Aufbau von Symbolleisten (Toolbars)

Ressourcedateien stellen damit ein flexibles Instrument zur Konfigurierung von Komponenteneigenschaften dar.

Auch Java bietet Klassen, welche die Verwendung separater, externer Ressourcen unterstützen. Kernstück ist die Klasse `java.util.ResourceBundle.` Ein ResourceBundle ist im wesentlichen eine hierarchisch organisierte Menge von Verzeichnissen, die jeweils ein Schlüsselwort auf einen beliebigen Text abbilden (sog. Key-Value Paare). Jedes einzelne dieser Verzeichnisse ist hierbei spezifisch für eine konkrete Sprache. Auch zwischen verschiedene Sprachvarianten kann mit `ResourceBundle` differenziert werden. So läßt sich beispielsweise zwischen britischen und amerikanischen Englisch unterscheiden. Die Unterklasse `PropertyResourceBundle` realisiert ein `ResourceBundle` basierend auf einer oder mehreren einfachen Dateien. Die Dateinamen bilden sich hierbei aus einem Basisnamen sowie zusätzlichen Angaben zu Land und Sprache. Wird zur Laufzeit vom Programm ein Ressourcentext angefordert, durchsucht `ResourceBundle` nacheinander die Dateien mit den folgenden Namen (sofern vorhanden):

```
<Basisname>_<Sprache>_<Land>_<Variante>.properties
<Basisname>_<Sprache>_<Land>.properties
<Basisname>_<Sprache>.properties
<Basisname>.properties
```

In der praktischen Arbeit hat es sich als vorteilhaft erwiesen, jeder Werkzeugkomponentenklasse ein eigenes `ResourceBundle` zuzuweisen. Über das `ResourceBundle` lassen sich allgemeine Komponenteneigenschaften sprachsensitiv konfigurieren. Für Komponentenklassen, welche die Root-Schnittstelle implementieren, wird im `ResourceBundle` auch der Aufbau der statischen Menüs sowie der Symbolleiste definiert. Die konkrete Komponentenzusammenstellung eines Softwarewerkzeug sowie differenzierte Einstellungen einzelner Komponentenexemplare innerhalb einer Werkzeugkomponentenhierarchie wird jedoch nicht über den `ResourceBundle`-Mechanismus realisiert. Diese Einstellungen lassen sich besser mit graphischen Builder Tools zur Entwurfszeit (siehe Java Beans, Kapitel 3.2) vornehmen. In Java Beans-Modellen werden die, in einem Builder Tool erzeugte und konfigurierte Komponentennetzwerke üblicherweise über den Java-Serialisierungsmechanismus persistent gemacht. Im Rahmen dieser Diplomarbeit wurde der Einsatz von graphischen Konstruktionswerkzeugen nicht weiter untersucht. Eine Behandlung des Themas und die Ausarbeitung eines flexiblen Mechanismus aus dem WAM-Umfeld ist beispielsweise in [Lippert97] und [FLL+98] zu finden.

Abbildung 28 zeigt einen Auszug aus der Properties-Datei der Wurzelkomponente `DiaryEditorComponent`.

```
# Base Resource strings for DiaryEditor Component

# frame title
# ===========
title=Diary Editor: {0}

# menubar definition
# ==================
menubar=diary edit

# diary menu definition
# =====================
diary=new open save saveas - exit
diaryLabel=Diary
newAction=new
openAction=open
saveAction=save
saveasAction=saveas
exitAction=quit

# edit menu definition
# ====================
edit=cut copy paste
editLabel=Edit
cutAction=cut-to-clipboard
copyAction=copy-to-clipboard
pasteAction=paste-from-clipboard

# toolbar definition
# ==================
toolbar=new open save - cut copy paste

# Action properties
# =================
newDiaryLabel=New
newDiaryMnemonic=n
newDiaryIcon=/toolcomp/comp/diaryeditor/images/new.gif
newDiaryTip=Create a new diary
...
```

Abbildung 28: ComponentResource.properties

Aufgrund der hierarchischen Anordnung der Properties-Dateien einer Komponente müssen in den länderspezifischen Dateien allein die veränderten Werte aufgeführt zu werden. Der Aufbau der Menüs, beispielsweise, braucht in den sprachspezifischen Resourcedateien nicht mehr angegeben zu werden. Abbildung 29 zeigt einen Ausschnitt aus der entsprechenden deutschsprachigen Ressourcedatei. Welche Sprachversion der Ressourcen zur Laufzeit verwendet wird, ist durch die entsprechende Ländereinstellung in den System-Properties sowie der Klasse `java.util.Locale` festgelegt.

```
# Resource strings for DiaryEditor Component
# Deutsch

title=Kalender Editor: {0}

diaryLabel=Kalender
editLabel=Bearbeiten

newDiaryLabel=Neu
newDiaryMnemonic=n
newDiaryTip=Erzeugt einen neuen Kalender

saveDiaryLabel=Speichern
saveDiaryMnemonic=s
saveDiaryTip=Speichert den bearbeiteten Kalender

...
```

Abbildung 29: ComponentResource_de.properties

5.2 Umsetzung der Basisschnittstellen

In diesem Abschnitt werden Java-Realisierungen der Standardschnittstellen von Werkzeugkomponenten vorgestellt, wie sie im Rahmen dieser Diplomarbeit spezifiziert wurden.

5.2.1 Subordinate

Die wesentliche Aufgabe einer Subordinate ist es, die Komponentenaktionen verfügbar zu machen und eine Schnittstelle zum Undo-Mechanismus zu definieren. Abbildung 30 zeigt die Java-Definition der Schnittstelle.

```
public interface Subordinate
  extends ComponentInterface
{
  /*
   * Generic Actions
   */
  public Action[] getActions();
  public Action getAction(String actionName)
        throws MissingActionException;

  /*
   * Undo Eventing
   */
  public void addUndoableEditListener(UndoableEditListener l);
  public void removeUndoableEditListener(UndoableEditListener l);
}
```

Abbildung 30: Subordinate

Action-Objekte

Die Komponentenschnittstelle Subordinate bietet über die generische Operation `getAction()`Zugriff auf die Komponentenfunktionalität. Action-Objekte sind ein Konzept, welches durch das Swing-Rahmenwerk in Java Einzug erhalten hat. Ein Action-Objekt kapselt die elementare, parameterlose Aktion einer Komponente und ist eine Anwendung des in [GHJV95] beschriebenen Command-Musters. Das wichtigste Einsatzgebiet von Action-Objekten in Java ist ihre Verwendung als ausführende Befehle von Tastaturkürzeln, Schaltflächen, Menü-

oder Symbolleisteneinträgen. Die Rückrufmechanismen (Callbacks) basieren in Swing-Komponenten ausschließlich auf dem allgemeinen Java Ereignismechanismus. Um beispielsweise im Programm auf das Drücken einer Schaltfläche reagieren zu können, muß an der Schaltfläche ein entsprechender `ActionListener` registriert werden, welcher über das Drücken-Ereignis informiert wird. Das Swing-Interface `Action` ist eine Erweiterung von `ActionListener` und kann damit direkt zur Beobachtung von `ActionEvent`s verwendet werden.

Abweichend vom klassischen Command-Muster besitzt ein Action-Objekt jedoch einen Zustand, der die Aktion selbst sowie ihre momentane Anwendbarkeit beschreibt. So ermöglicht die Operation `setEnabled()` beispielsweise eine Aktion explizit zu aktivieren bzw. deaktivieren. Hierbei informiert das Action-Objekt über ein Ereignis vom Typ `PropertyChangeEvent` interessierte Beobachter über jede Zustandsänderung. Das allgemeine Konstruktionsmuster bei der Verwendung von Action-Objekten in Swing sieht nun vor, daß ein GUI-Element direkt oder indirekt ebenfalls das bei ihm registrierte Action-Objekt beobachtet, um seinen eigenen Zustand mit diesem zu synchronisieren. Wird eine Aktion deaktiviert, führt das automatisch auch zu einer Deaktivierung des entsprechenden GUI-Elements. Ist die Beziehung zwischen Komponentenaktionen und Action-Objekten eineindeutig, so lassen sich mehrere GUI-Elemente, die dieselbe Aktion anstoßen, auf einfache Weise konsistent halten.[19]

Das Objektdiagramm in Abbildung 31 zeigt den Ausschnitt eines Werkzeugs, bei dem dieselbe Aktion sowohl über einen Knopf als auch über einen Menüeintrag aktiviert werden kann. Die dargestellten Struktur zeigt eine Form, bei der die GUI-Elemente JButton und JMenuitem das Action-Objekt nicht direkt beobachten. Zwischen GUI-Element und Action-Objekt ist jeweils ein Adapter-Objekt geschaltet, welches die PropertyChangeEvents der Aktion beobachtet und das für die Synchronisation zuständig ist. Diese gebräuchliche Struktur vermeidet die Bildung von Unterklassen der GUI-Elemente. Häufig werden die Adapter-Objekte über Inner Classes realisiert.

Der Einsatz von Action-Objekten entkoppelt in einem Programm die funktionale Logik von den GUI-Elementen. Ist eine Komponentenaktion zu einem bestimmten Zeitpunkt nicht möglich bzw. fachlich nicht sinnvoll, so muß lediglich das entsprechende Action-Objekt deaktiviert werden. Durch die Synchronisation der GUI-Elemente mit ihren Action-Objekten werden automatisch auch alle betroffenen Schaltflächen, Menüeinträge etc. deaktiviert.

[19] Hier wird deutlich, daß in Java der Ereignismechanismus als ein sehr (und vielleicht zu) allgemeines Mittel der Objektinteraktion verwendet wird. Während die Beobachtung der PropertyChangeEvents durch ein oder auch mehrere GUI-Elemente dem klassischen Observer-Muster entspricht, handelt es sich bei der Registrierung der Aktion als ActionListener eher um eine Parametrierung des GUI-Elements mit einem Command-Objekt. Obgleich das Konzept der Event Listeners es prinzipiell zulassen würde, wird es im allgemeinen nicht sinnvoll sein, an einer Schaltfläche gleichzeitig mehrere Action-Objekte zu registrieren.

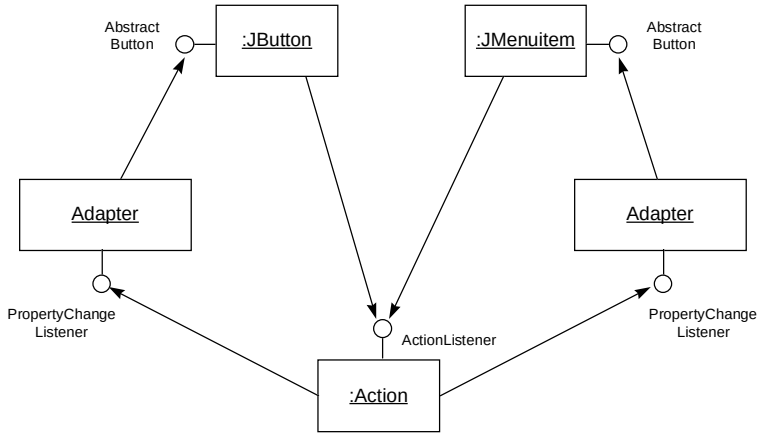

Abbildung 31: GUI-Action-Kopplung

Das Konzept der Action-Objekte wurde im Rahmen dieser Arbeit für Werkzeugkomponente noch etwas erweitert. Die Klasse `ComponentAction` implementiert das Action-Interface und kapselt neben der auszuführenden Aktivität die folgenden Attribute einer Komponentenaktion:

- textuelle Beschreibung
- Icon zur bildlichen Repräsentation
- Tastenkombination zum Aufruf (Shortcut)
- Hilfetext (Tooltip)

Die entsprechenden Daten bezieht eine ComponentAction aus dem Resource-Bundle der Werkzeugkomponente. Hierdurch lassen sich die Attribute von Action-Objekten innerhalb einer Anwendung automatisch einheitlich länder- und sprachspezifisch eingestellen.

Alle Aktionen stehen über die Komponentenschnittstelle Subordinate zur Verfügung. Bei zusammengesetzte Komponenten enthält die Schnittstelle die Summe aller Aktionen der enthaltenen Subkomponenten.

Undo/Redo-Mechanismus

Über die Komponentenschnittstelle Subordinate ist festgelegt, daß jede Werkzeugkomponente prinzipiell einen Undo-Mechanismus unterstützt. Für moderne interaktive Softwarewerkzeuge ist das heute zu einer unverzichtbaren Eigenschaft geworden. In dieser Diplomarbeit wurde die Undo-Funktionalität auf Basis des Swing Undo-Package realisiert. Bei jeder Zustandsänderung wird hierbei ein Effekt-Objekt vom Typ `UndoableEdit` erzeugt, das in der Lage ist, die durchgeführte Zustandsänderung wieder zurückzunehmen. Über Ereignisse werden die Effekt-Objekte in der Werkzeugkomponentenhierarchie aufwärts propagiert. Erst die Wurzelkomponente enthält einen Undo-Manager, welcher die Ereignisse beobachtet und die Effekt-Objekte für ein potentielles Rücksetzen

sammelt. Untergeordnete Komponenten besitzen keinen eigenen Undo-Manager.[20]

Bei der Anwendung des Undo-Mechanismus im Rahmen von Werkzeugkomponentenhierarchien müssen einige Konstruktionsaspekte beachtet werden.

Trennung von Aktion und Effekt

Das Konstruktionsschema trennt streng zwischen der Ausführung einer Komponentenaktion und den daraus resultierendem Zustandsänderungen am Material oder Werkzeug. Die Ausführung einer Aktion ist in Action-Objekten realisiert während die resultierenden Zustandsänderungen sowie deren Rückgängigmachung in UndoableEdit-Objekten gekapselt sind. Um nicht endlose Folgen von Effekt-Objekten zu erzeugen, dürfen beim Ausführen einer Undo- bzw. Redo-Operation selbst keine weiteren Effekt-Objekte mehr propagiert werden. So einleuchtend diese Anforderung klingt, so schwierig erweist sich in der Praxis teilweise ihre Umsetzung. Obgleich beispielsweise die Ausführung eines Redo (nach vorherigem Undo) dieselbe Zustandsänderung innerhalb einer Komponente bewirkt wie das ursprüngliche Action-Objekt, müssen beide Operationen zum Teil unterschiedlich implementiert sein, damit beim Redo nicht ebenfalls ein Effekt-Objekt erzeugt wird.

Zusammenfassen von Änderungen

Es ist nicht immer gewünscht, daß ein Undo-Vorgang dieselbe Granularität wie die vorangegangene Aktion aufweist. So bietet es sich beispielsweise an, im Texteditor alle über die Tastatur ohne eine Unterbrechung eingetippten Zeichen zu einer einzigen Änderung zusammenzufassen, welche auch nur über einen einzigen Undo-Aufruf vollständig zurückgenommen werden kann. Diese Form des Undo hat sich heute bei den meisten Editoren durchgesetzt. Im Undo-Package von Swing wird dieses dadurch realisiert, daß der Undo-Manger jedem Effekt-Objekt beim Einfügen in die Schlange der durchgeführten Änderungen die Möglichkeit gibt, das vor ihm stehende Objekt zu ersetzen. Hierzu ist ein Effekt-Objekt natürlich nur in der Lage, wenn es den konkreten Typ und die konkrete Wirkung des anderen Objekts ermitteln kann.

Insignifikante Änderungen

Nicht alle verändernden Aktionen, müssen gleichermaßen rückgängig gemacht werden können. Aktionen, die ausschließlich die momentane Werkzeugdarstellung an der Benutzungsoberfläche verändern, haben einen anderen Stellenwert als Aktionen, die eine dauerhafte Veränderung bewirken. Als minimale Anforderung kann sicher gelten, daß alle Veränderungen am aktuell bearbeiteten Material jederzeit über mehrere Stufen zurück widerrufbar sein sollten. Dagegen wird es im allgemeinen aber nicht erforderlich sein, die Positionsveränderungen des Mauszeigers über den Undo-Mechanismus abzusichern. Es gibt jedoch auch Aktionen, die sich zwar zuerst nur auf die Darstellung auswirken, deren Auswirkungen aber trotzdem über Effekt-Objekte rückgängig gemacht werden können müssen, sobald eine andere, frühere Aktion widerrufen wird. Ein Szenario aus der Verwendung des Kalenderwerkzeugs soll dies verdeutlichen:

[20] Der Swing Undo-Mechanismus sowie eine alternative Implementierungsvariante wird später in Kapitel 5.6.2 noch einmal genauer betrachtet.

> *"Das Kalenderwerkzeug zeigt eine Auswahlliste von Terminen. Per Mausklick wählt der Anwender nacheinander verschiedene Termine aus und betrachtet jeweils die auf der rechten Seite angezeigten dazugehörigen Notizen."*

Die Auswahl eines Termins in der Auflisterkomponente hat zuerst einmal nur eine Auswirkung auf die Darstellung des gerade bearbeitenden Kalenders. Der ausgewählte Termin wird farbig hinterlegt und die dazugehörigen Notizen werden in die Editorkomponente geladen und angezeigt (siehe Abbildung 32). Für eine sinnvolle Handhabung des Werkzeugs ist es nicht erforderlich, daß eine Terminauswahl über den Menüpunkt "Widerrufen" rückgängig gemacht werden kann. Hat der Anwender versehentlich einen falschen Termin ausgewählt, so kann er schließlich mit einem erneuten Mausklick die Auswahl problemlos korrigieren.

> *"Beim ausgewählten Termin löscht der Anwender jetzt im Editor eine Textzeile. Anschließend blättert er weiter im Kalender und wählt er einen neuen Termin aus. Beim genauen Betrachten des aktuellen Termins stellt er fest, daß er die eben gemachte Löschung wohl versehentlich beim falschen Eintrag vorgenommen hat. Er klickt auf den Menüpunkt Bearbeiten-Widerrufen um die Textlöschung wieder rückgängig zu machen."*

Die Textlöschung im Editor ist eine Aktion, die eine Auswirkung auf das bearbeitete Material hat und welche über den Undo-Mechanismus abgesichert sein muß. Zwischen der Textlöschung und Aufruf des Widerrufenkommandos hat der Anwender jedoch einen neuen Termin ausgewählt. Das heißt, der Widerruf bezieht sich nicht auf den aktuell ausgewählten und im Editor angezeigten Termin, sondern auf eine frühere Auswahl. Ein schlichtes Rückgängigmachen der Löschung brächte zwar das bearbeitete Material in den gewünschten Zustand, jedoch bliebe der Widerruf für den Anwender erst einmal unsichtbar. Erst durch eine erneute Auswahl des Widerrufenen Termins könnte er sich vom Ergebnis des Undo überzeugen. Eine solche Handhabung wäre sicher nicht aktzeptabel. Eine sehr viel intuitivere und einfachere Handhabung ergibt sich, wenn mit dem Widerruf auch automatisch wieder der korrigierte Termin ausgewählt wird.

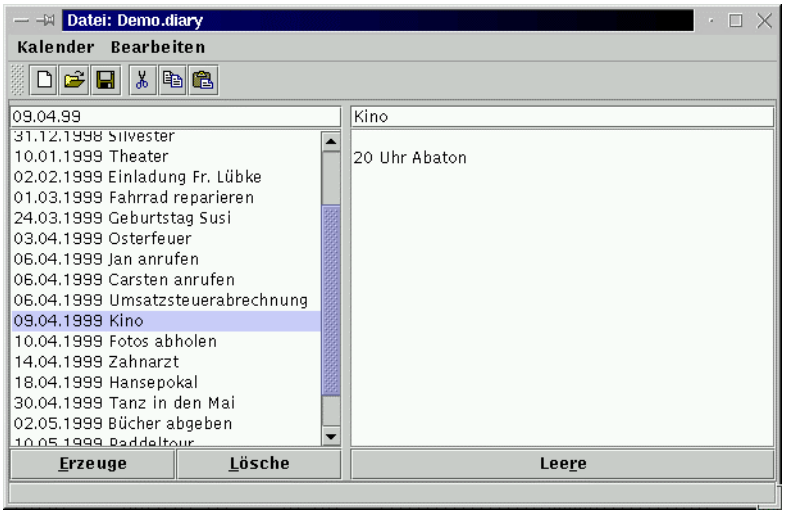

Abbildung 32: Terminbearbeitung

Prinzipiell könnte das gewünschte Verhalten des Kalenderwerkzeugs dadurch erreicht werden, daß das Effekt-Objekt der Textlöschung in seiner Undo-

Realisierung auch dafür sorge trägt, daß der dazugehörige Termin ausgewählt wird. Dieser Ansatz verträgt sich jedoch nicht mit der komponentenorientierten Architektur des Werkzeugs. Eine Stärke der Architektur ist ja gerade, daß die Aufgabenbereiche der einzelnen Komponenten klar voneinander getrennt sind und daß eine Komponente möglichst keine Annahmen über den Kontext in dem sie eingesetzt wird machen muß. Unter diesem Gesichtspunkt sollte die Editorkomponente also gar nicht wissen, daß sie in einem Kalenderwerkzeug eingesetzt wird und daß der von ihr editierte Text zu einem Termin gehört, der in einer anderen Werkzeugkomponente ausgewählt worden ist. Die Aufgabe der Editorkomponente ist lediglich dafür Sorge zu tragen, daß ein beliebiges Material unter dem Aspekt *Editierbar* vom Anwender sinnvoll bearbeitet werden kann.

Der Undo-Mechanismus des Swing-Rahmenwerks ermöglicht eine im Kontext von Werkzeugkomponenten bessere Lösung für Probleme dieser Art. Effekt-Objekte des Typs `UndoableEdit` können als *insignifikant Änderungen* markiert werden (`isSignificant()`). Insignifikante Änderungen werden vom Undo-Manager genauso zurückgesetzt wie normale Änderungen, jedoch zeigt der Undo-Manager sie nicht explizit als widerrufbare Änderungen an seiner Schnittstelle an. Wird ein Undo-Manager beauftragt die letzte Änderung zurückzunehmen, bedeutet das konkret, daß der Manager alle Änderungen bis einschließlich der letzten signifikanten Änderung zurücksetzt. Als Seiteneffekt eines Undo werden also auch alle insignifikanten Änderungen zurückgenommen. Mit Hilfe dieses Verfahrens läßt sich das oben beschriebene Problem sehr einfach lösen. Auch Terminenselektionen in der Auflisterkomponente müssen Effekt-Objekte erzeugen. Anders, als die Effekt-Objekte von Textänderungen im Editor, handelt es sich bei ihnen jedoch um insignifikante Änderungen. Die Arbeitsweise des standard Undo-Manager des Swing-Rahmenwerks gewährleistet somit automatisch das gewünschte Werkzeugverhalten.

5.2.2 View

Werkzeugkomponenten besitzen immer eine Repräsentation an der graphischen Benutzungsoberfläche. Innerhalb einer Werkzeugkomponentenhierarchie sind auch diese Komponentenoberflächen im allgemeinen hierarchisch miteinander verbunden (Widget-Hierarchie). Das Protokoll, wie die einzelnen Bestandteile miteinander interagieren ist durch das verwendete GUI-Rahmenwerk festgelegt. Während in der Vergangenheit noch recht viel Programmieraufwand erforderlich war um die Koordination und das Zusammenspiel der einzelnen GUI-Elemente innerhalb der Hierarchie zu realisieren, übernehmen modernere GUI-Rahmenwerke diese Aufgaben heute zum größten Teil selbst. So steuert das Rahmenwerk beispielsweise in welcher Reihenfolge der Bildaufbau erfolgt, welche Bereiche der Oberfläche neugezeichnet werden müssen oder wie Tastatureingaben und Mausaktionen dem Programm zugänglich gemacht werden.

Die Abhängigkeit der Benutzungsoberfläche von dem implementierenden GUI-Rahmenwerken ist der Grund, daß im Werkzeugkomponentenmodell hierfür jeweils eigene Schnittstellen definiert werden. Eine allgemeine generische View-Schnittstelle erscheint nicht sinnvoll, da unterschiedliche GUI-Protokolle in der Regel nicht kompatibel zueinander sind. Selbst im engen Rahmen der standard Java-Klassen gibt es mittlerweile zwei konkurrierende Ansätze für die Realisierung graphischer Benutzungsschnittstellen. Zum einen existiert von Beginn an das AWT-Rahmenwerk (Abstract Windowing Toolkit) und zum anderen ist seit JDK1.2 das Swing-Rahmenwerk hinzugekommene. Das AWT-

Rahmenwerk basiert auf schwergewichtigen GUI-Klassen, die jeweils eins-zu-eins auf native GUI-Elemente des Toolkits der ausführenden Systemplattform abgebildet werden. Swing-Klassen stellen dagegen leichtgewichtigere und plattformunabhängigere Komponenten dar. Ihr Look-and-Feel ist unabhängig von der ausführenden Systemplattform in eigenen Klassen definiert und außer den Top-level Fenstern werden die GUI-Elemente in Swing nicht direkt auf native Widgets abgebildet.

Obgleich die GUI-Klassen in Swing generell Unterklassen der entsprechenden AWT-Klassen sind, handelt es sich doch um zwei sehr unterschiedliche Techniken und es wird in der Dokumentation dringend davor abgeraten, in einem Fenster AWT- und Swing-Klassen zu mischen. Ein Mix von schwergewichtigen und leichtgewichtigen GUI-Komponenten auf einer Werkzeugoberfläche ist nicht ohne weiteres möglich und kann im Detail zu technischen Schwierigkeiten führen (beispielsweise kann ein leitgewichtiges GUI-Element nie ein schwergewichtiges AWT-Element überdecken). Dadurch ist es auch nicht möglich, zwei Werkzeugkomponenten ohne weiteres in einer Komponentenhierarchie zu integrieren, wenn die Oberfläche der einen Komponente mit dem AWT implementiert ist und die Oberfläche der anderen Komponente mit Swing. Die Verwendung von separaten View-Schnittstellen für AWT-Oberflächen und Swing-Oberflächen vereinfacht die Wiederverwendung von Werkzeugkomponenten sehr, da damit die beschriebenen Abhängigkeiten explizit schon im Architekturmodell beschrieben werden können. Das Konzept gestattet es auch, daß eine Komponente mehrere View-Schnittstellen unterstützt. Je nachdem, in welchem Kontext die Komponente eingesetzt wird, kommt dann entweder ihre AWT- oder ihre Swing-View zum Einsatz.

Das im Rahmen dieser Arbeit verwendete Swing-View-Interface ist sehr einfach gehalten (Abbildung 33). Es definiert die Oberflächenrepräsentation einer Komponente schlicht als ein Objekt vom allgemeinen Typ `JComponent`. Da die View-Schnittstelle ausschließlich zur Bildung der View-Hierarchie verwendet wird, ist es nicht von Belang, von welcher konkreten Klasse das View-Objekt abstammt. Neben ihrem Darstellungsbereich kann eine Komponente an ihrer SwingView-Schnittstelle außerdem noch zusätzlich ein Objekt für die Verwendung in der Statuszeile des Hauptfensters anbieten.

```
public interface SwingView
 extends ComponentInterface
{
 public JComponent getSwingComponent();
 public JPanel getSwingStatusPanel();
}
```

Abbildung 33: SwingView

Um eine möglichst große Flexibilität gegenüber Änderungen zu erhalten, ist eine Separierung der reinen Oberflächenrepräsentation von der übrigen Implementation im Inneren einer Komponente aber durchaus sinnvoll. Gerade die Gestaltung von Benutzungsoberflächen erfolgt heute in der Regel werkzeugunterstützt mit sog. GUI- bzw. Interface-Buildern. In der Regel erzeugen die Interface-Builder hierbei neue Ableitungen vorhandener GUI-Klassen evtl. ergänzt um externe Ressourcebeschreibungen. Bei der Konstruktion von Werkzeugkomponentenoberflächen mit entsprechenden, allgemein verfügbaren Entwicklungswerkzeugen stößt man jedoch schnell auf das bereits an anderer

Stelle aufgeführte Problem des Architectural Mismatch [GAO95]. Viele Builder tendieren dazu, neben der reinen Oberflächengestaltung auch mehr oder minder große Teile der Ablaufsteuerung zu generieren. Nur im Ausnahmefällen wird dieses Coding jedoch mit dem hier verwendeten Werkzeugkomponentenmodell problemlos verträglich sein. Im Rahmen der Entwicklung von Werkzeugkomponenten wird der Einsatz von Interface-Buildern daher meistens auf die reine Oberflächengestaltung beschränkt bleiben. Die technische Trennung der reinen Darstellung und der eigentlichen, komplexeren Interaktion im Komponenteninneren wird im allgemeinen den Einsatz von Entwicklungswerkzeugen erleichtern.

Im praktischen Teil dieser Arbeit wurden keine Interface-Builder eingesetzt. Komponentendarstellungen werden meist als einfach gehaltene direkte Ableitung von standard Swing-Klassen realisiert. Die Erzeugung der View-Klassen erfolgt mit dem JavaBeans Erzeugungsmechanismus `Beans.instantiate()`. Das bedeutet, eine View kann auch in serialisierter Form vorliegen.

5.2.3 Manager

Die Hauptaufgabe der Manager-Schnittstelle liegt im Aufbau einer Zuständigkeitskette aufwärts der Komponentenhierarchie. Hierfür erfüllt die Manager-Schnittstelle auch die allgemeine Schnittstelle `RequestHandler`. Darüber hinaus ermöglicht Manager den Zugriff auf alle angeschlossenen untergeordneten Komponenten unter einer einheitlichen Subordinate-Schnittstelle.

```
public interface RequestHandler
{
 public void handleRequest(Request r);
}

public interface Manager
 extends ComponentInterface, RequestHandler
{
 public Subordinate[] getSubordinates();
}
```

Abbildung 34: Manager

5.2.4 SwingRoot

In einem Swing-Umfeld kann eine Wurzelkomponente unter der SwingRoot-Schnittstelle als eigenständiges Werkzeug verwendet werden. Wie schon an früherer Stelle aufgeführt, gehört das äußere Fenster eines Werkzeugs selbst nicht mit zur Wurzelkomponente. Dieses entspricht der Architektur von Swing-Anwendungen. Auch innerhalb des Swing-Rahmenwerks wird zwischen Fenstern und Widget-Hierarchien unterschieden. Ein Fenster bilden den Rahmen für eine Widget-Hierarchie, gehört aber selbst nicht direkt dazu. Die Wurzel einer Widget-Hierarchie wird in Swing von der Klasse `JRootPane` realisiert. Diese Wurzel läßt sich in einen Container einbetten, der die Schnittstelle `JRootPane-Container` implementiert. Zu den Standardklassen, die dieses Interface implementieren, gehören zum einen die schwergewichtigen Fensterklassen `JWindow`, `JFrame` und `JApplet`, zum anderen aber auch die leichtgewichtige Fensterklasse `JInternalFrame`.

```
public interface SwingRoot
 extends ComponentInterface
{
 public JRootPane getSwingRootPane();

     public void startup();
     public void shutdown();
}
```

Abbildung 35: SwingRoot

Für die Einbettung in einen beliebigen Fensterkontext bietet eine Werkzeugkomponente an ihrer SwingRoot-Schnittstelle eine Widget-Wurzel vom Typ `JRootPane`. `JRootPane` beinhaltet neben der, über die SwingView verfügbaren Oberflächenkomponente, auch die Menüzeile, die Symbolleiste sowie die Statuszeile. Durch die Trennung von Fenster und Widget-Hierarchie läßt sich eine Werkzeugkomponente unter der `SwingRoot`-Schnittstelle ohne Änderung in einem `JFrame` einer Applikation, einem `JApplet` eines Applets oder auch in einem `JInternalFrame` eines eigenen Desktop verwenden.

5.2.5 Komponentenspezifische Serviceschnittstellen

Komponentenspezifische Serviceschnittstellen, die in eine Werkzeugkomponentenhierarchie eingebettet werden können, erweitern das Subordinate-Interface um weitere Operationen. In der Regel orientieren sich diese Schnittstellen an einem Aspekt, unter dem die Werkzeugkomponente ein Material bearbeiten kann.

```
public interface Browser
 extends Subordinate
{
  /*
   * Service
   */
  public void loadMaterial(Browseable mat);
  public void reloadMaterial(Browseable mat);
  public void unloadMaterial();
  public Browseable getLoadedMaterial();

  public int getSelection();
  public void select(int i);
  public void clearSelection();
  public boolean isSelected(int i);
  public boolean isAnySelected();


  /*
   * Events
   */
  public void addMaterialListener(MaterialListener l);
  public void removeMaterialListener(MaterialListener l);

  public void addSelectionListener(ListSelectionListener l);
  public void removeSelectionListener(ListSelectionListener l);
}
```

Abbildung 36: Browser-Schnittstelle

Abbildung 36 zeigt beispielhaft die Schnittstelle des im Kalenderwerkzeugs verwendeten Auflisters, der Materialien unter dem Aspekt auflistbar (Browseable) verwendet.

5.3 Komponentenrahmenwerke

Die Entwicklung objektorientierter Systeme erfolgt heute meistens rahmenwerkbasiert. Rahmenwerke sind nicht nur ein Mittel zur Wiederverwendung von Programmcode. Vor allem erlauben Rahmenwerke auch die Wiederverwendung bewährter Entwurfsentscheidungen innerhalb eines spezifischen Anwendungsbereichs. Im Kontext der objektorientierten Implementierung komponentenorientierter Systeme können Rahmenwerke an zwei Stellen sinnvoll eingesetzt werden. Zum einen bietet es sich an, den einbettende Kontext, welcher die Infrastruktur und das Zusammenspiel zwischen verschiedenen Komponenten realisiert, in einem eigenen Rahmenwerk zu kapseln. Für alle Komponentenimplementationen, die auf diesem Basisrahmenwerk aufbauen, ist so gewährleistet, daß sie problemlos zusammenarbeiten. Zum anderen sind Rahmenwerke auch ein geeignetes Mittel um die Entwicklung neuer Komponenten zu vereinfachen (vgl. [Johnson97]).

Gerade größere Softwaresysteme basieren nicht nur auf einem einzelnen Rahmenwerk, sondern auf mehreren Rahmenwerken, die in geordneter und kontrollierter Form zusammenarbeiten. Entsprechend den Erfahrungen und Regeln der Modulbildung sollten die Rahmenwerke hierbei nur in asymmetrischen Abhängigkeiten zueinander stehen. [Bäumer98] empfiehlt Rahmenwerke in Schichten zu strukturieren. Einen anderen Ansatz beschreibt [Willamowius97], der Partitionen als Strukturierungstechnik verwendet. Entsprechend wurden auch in dieser Diplomarbeit verschiedene Rahmenwerke entwickelt, um das Werkzeugkomponentenmodell zu implementieren. Während die Komponenteninfrastruktur in einem separaten Basisrahmenwerk implementiert ist, basiert jede Komponentenimplementation auf einem eigenen, unabhängigen Rahmenwerk. Die speziellen Komponentenrahmenwerke sind hierbei abhängig vom Basisrahmenwerk, das Basisrahmenwerk ist jedoch vollständig unabhängig von den konkreten Komponentenrahmenwerken.

In Kapitel 2.3 wurde bereits auf strukturelle Ähnlichkeiten zwischen den Komponentenklassen der Modellierungsebene und Rahmenwerke der Implementationsebene hingewiesen. Ein Merkmal von Komponenten ist ihre ausgeprägte Eigenständigkeit und Abgeschlossenheit. Innerhalb einer Werkzeugkomponentenhierarchie besitzt jede Komponente ihren eigenen Aufgabenbereich für den sie zuständig ist und in dessen Rahmen sie Aufträge, soweit es ihr möglich ist, selbständig bearbeitet. Genauso ist einem Rahmenwerk eine große Eigenständigkeit inhärent, da es nicht nur unabhängige Klassen enthält, sondern immer auch einen Kontrollfluß zwischen den Objekten der Klassen festlegt.

In der Literatur (vgl. [JF88]) werden verschiedene Typen von Rahmenwerken unterschieden. Die Verwendung von White-Box Rahmenwerken basiert primär darauf, daß von speziellen Klassen des Rahmenwerks Unterklassen abgeleitet werden, von denen sich dann für die weitere Verarbeitung Objekte erzeugt lassen. White-Box Rahmenwerke erfordern meist ein nicht unerhebliches Verständnis der internen Abläufe und Strukturen der verwendeten Rahmenwerksklassen. Im Gegensatz dazu lassen sich Black-Box Rahmenwerke ohne detaillierte Kenntnisse der Interna verwenden. In einem Black-Box Rahmenwerk basiert die Verwendung primär auf der direkten Benutzung der Rahmenwerksklassen ohne den Vererbungsmechanismus einzusetzen. Black-Box Rahmenwerke sind dadurch in ihrer Verwendung abgeschlossenere Einheiten als ihre White-Box Gegenstücke. Da auch Komponenten eine starke Abgeschlossenheit aufweisen und für ihren

Einsatz in Werkzeugkomponentenhierarchien kein Wissen über ihre Implementationsdetails erforderlich sein darf, ist es wichtig, daß sich die Komponentenimplementationen ohne Detailkenntnisse über internen Strukturen, d. h. ohne Vererbungsmechanismen, verwenden lassen. Komponentenimplementationen werden deshalb durch Black-Box Rahmenwerke realisiert. Dagegen ist das Basisrahmenwerk der Komponenteninfrastruktur in der Regel ein White-Box Rahmenwerk, deren Klassen erst in den Komponentenrahmenwerken konkretisiert werden.

Bei einer genauen Betrachtung können bei der Verwendung von Komponenten zwei Bereiche unterschieden werden. Zum einen werden Komponenten verwendet, um in einem größeren Kompositum als abgeschlossene Einzelbestandteile eingesetzt zu werden. Zum anderen kann eine Komponentenklasse die Grundlage für die Entwicklung einer neuen Komponentenklasse bilden. Während bei der Komponentenkomposition allgemein keine Vererbungsbeziehungen zwischen den implementierenden Rahmenwerken benötigt werden, ist das bei der Entwicklung neuer Komponenten nicht unbedingt der Fall. Die wesentliche Bedeutung der Vererbung in der Objektorientierung liegt in der Modellierung fachlicher Gemeinsamkeiten zwischen verwandten Begriffen und Konzepten. Mit dem Mechanismus der Vererbung lassen sich Begriffshierarchien von Generalisierungen und Spezialisierungen beschreiben und verdeutlichen (vgl. [Züllighoven98]). Auch Komponentenklassen lassen sich in Begriffshierarchien beschreiben. Neue Komponentenklassen können sich als Spezialisierungen aus existierenden Komponentenklassen ableiten. In diesem Fall kann es durchaus hilfreich sein, auch Vererbungsbeziehungen zwischen den implementierenden Rahmenwerken einzusetzen. Das bedeutet, daß für die Entwicklung neuer Komponenten die implementierenden Rahmenwerke teilweise doch white-box artig verwendet werden. Bäumer nennt Rahmenwerke mit dieser Eigenschaft *offen verwendbare Black-Box-Rahmenwerke* [Bäumer98].

5.4 Packages

Die Programmiersprache Java besitzt ein Package-Konzept, mit dem sich hierarchische Namensräume bilden lassen. Es bietet sich an, unterschiedliche Rahmenwerke in unterschiedlichen Packages zu organisieren um so eine gute Kapselung der Rahmenwerke zu erhalten.

Für die Strukturierung von Rahmenwerken wird in [Bäumer98] vorgeschlagen, ein Rahmenwerk jeweils in seinen Konzeptteil und ein oder mehrere Realisierungsteile aufzugliedern. Diese Struktur fördert allgemein eine bessere Modularisierung. Entsprechend wird in dieser Diplomarbeit ein Komponentenrahmenwerk prinzipiell in mindestens zwei Packages aufgeteilt. Für jede Werkzeugkomponente existiert ein Konzept-Package, deren wichtigste Bestandteile die Interfaces der Komponentenschnittstellen sind sowie ein oder mehrere Realisierungs-Packages mit den implementierenden Klassen. Die Packages stehen in einer hierarchischen Schichtenordnung zueinander. Für die Benutzung einer Komponentenklasse innerhalb einer Werkzeugkomponentenhierarchie genügt die Abhängigkeit zum entsprechenden Konzept-Package. Von den Realisierungs-Packages sollten Klienten nicht abhängig sein.

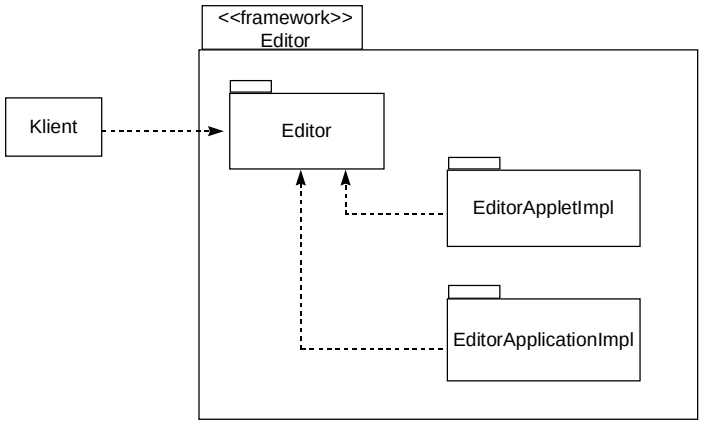

Abbildung 37: Package-Struktur

5.5 Komponentenbibliotheken

Für einen harmonischen Übergang zwischen den Abstraktionsebenen (siehe Kapitel 2) empfiehlt es sich, Werkzeugkomponenten auch auf der Ausführungsebene als eigenständige Komponenten zu realisieren. Innerhalb einer Java-Umgebung bieten sich hierfür JAR-Archive an. Außer den übersetzten Klassen können in einem JAR-Archiv auch alle von der Komponente sonst noch benötigten Ressourcen wie Bilder oder sprachabhängige Ressourcedateien gebündelt werden.

JAR-Archive ähneln den bekannten dynamisch ladbaren Bibliotheken (DLL). Sie enthalten Objekt-Dateien mit noch nicht aufgelösten Referenzen. Da JAR-Archive aber ein integraler Bestandteil von Java sind, stellen sie im Vergleich ein mächtigeres Bibliothekskonzept dar. Während DLLs gemeinhin ausschließlich vom Laufzeitsystem (Linker) verwendet werden um fehlende Referenzen aufzulösen, werden JAR-Archive zur Laufzeit von mächtigen Class Loadern bearbeitet. Da ein JAR-Archiv neben den einzelnen Objekt-Dateien auch zusätzliche Meta-Informationen enthält, hat der Class Loader mehr Möglichkeiten die Objekt-Dateien zuzuordnen und in das laufende System einzuhängen. Der Java Download Extensions-Mechanismus ermöglicht es beispielsweise in der Datei `META-INF/MANIFEST` innerhalb eines JAR-Archivs Abhängigkeiten zu anderen Archiven zu beschreiben. Auf diese Weise ist es nicht erforderlich alle potentiell erforderlichen Objekt-Dateien von vornherein in einem einzigen Archiv zusammenzufassen. Auch müssen nicht alle potentiell relevanten Bibliotheken schon zu Systemstart direkt lokal verfügbar sein. Der Class Loader kann beispielsweise zusätzlich benötigte Archive bei Bedarf dynamisch zur Laufzeit über das Netzwerk nachladen und in das System einbinden. Eine statische Angabe aller benötigten JAR-Archive im Classpath der ausführenden Maschine ist dafür nicht erforderlich.

Eine weitere nützliche Eigenschaft von JAR-Archiven ist die Möglichkeit Versionsnummern für die in ihnen enthaltenen Packages zu definieren. Diese Versionsnummern sind im Programm über spezielle Klassen zugänglich, so daß mit ihnen gewährleistet werden kann, daß auch Packages aus nachgeladenen Archiven immer einen zum Gesamtsystem kompatiblen Stand besitzen.

In dieser Diplomarbeit wurden die Komponenten der Ausführungsebene so organisiert, daß jede (Architektur-)Komponentenklasse durch ein eigenes JAR-Archiv repräsentiert wird. Zusammengesetzte Komponenten, die für ihre Arbeit auf andere Subkomponenten zurückgreifen müssen, besitzen in der `Class-Path` Angabe ihrer Manifest-Datei lediglich ein Verweis auf die Archive der benötigten Komponenten. Auf diese Weise bleiben Werkzeugkomponenten auch auf der Ausführungsebene als separate Systemelemente erhalten.

5.6 Implementierungsentscheidungen

Bei der praktischen Realisierung habe ich einige prinzipielle Implementierungsentscheidungen treffen müssen. Obgleich es sich um Implementierungsaspekte handelt, die konkret mit der Programmiersprache Java zusammenhängen, lassen sich hieraus Erkenntnisse allgemeinerer Natur ableiten.

5.6.1 Architekurkonformität

Ähnlich wie Shaw/Garlan beschreiben Moriconi et al. Softwaresysteme als Hierarchien abstrakter und immer konkreter werdender Architekturen [MQR95]. Die Implementation definiert dabei eine sehr konkrete und detaillierte Systemarchitektur. Innerhalb dieser Hierarchie ist es wichtig, daß jede konkrete Architektur immer eine korrekte Verfeinerung der abstrakteren Architektur darstellt. Wann ist aber eine Implementation eine korrekte Verfeinerung der abstrakten Softwarearchitektur?

Ein gutes Architekturmodell zeigt alle architekturrelevanten Annahmen, die dem System zugrunde liegen und hilft damit Konsequenzen zukünftiger Änderungen besser abschätzbar zu machen. Die Komponentenorientierung betont die explizite Modellierung der Komponentenschnittstellen sowie die Komponenteninteraktion über diese Schnittstellen. Diese expliziten Modellelemente beschreiben gewissermaßen die *tragenden Wände* einer Systemarchitektur (vgl. [PW92]) und ermöglichen schon auf der Ebene des Komponentenmodells eine Abschätzung, ob und mit welchem Aufwand Komponenten eines Werkzeugs ausgetauscht oder verändert werden können. Das erfordert, daß schon im Komponentenmodell alle Interaktionskanäle zwischen Werkzeugkomponenten beschrieben sind. Eine korrekte Verfeinerung in Sinne von Moriconi et al. bedingt daher, daß auf der Implementationsebene keine zusätzlichen, im Architekturmodell nicht aufgezeigten, Interaktionskanäle zwischen Komponenten aufgebaut werden dürfen.

Prinzipiell besteht eine potentielle Gefahr, daß durch Weitergabe von Objektreferenzen zusätzliche Interaktionskanäle aufgebaut werden, die im abstrakteren Komponentenmodell nicht explizit durch Komponentenschnittstellen modelliert sind. Vor allem die explizite Typumwandlung (casting) von Objektreferenzen führt häufig zu zusätzlichen, unerlaubten Interaktionskanälen, da sie Operationen zugänglich macht, die in keiner Komponentenschnittstelle explizit angegeben sind. Aus diesem Grund ist es ratsam, wann immer möglich, die Interaktion zwischen Komponenten wertebasiert zu implementieren.

5.6.2 Architectural Mismatch

Java ist mehr als nur eine objektorientierte Sprache. Untrennbar mit der Sprache verbunden ist eine große Anzahl von Rahmenwerken des Java Development Kit (JDK). Diese Rahmenwerke stellen eine Basisinfrastruktur für Java-Anwen-

dungen bereit. Strenggenommen benötigt ein Java-Programm aus dem JDK lediglich die Klassen des Package `java.lang`. Alle außerdem erforderlichen Klassen könnten selbstgeschrieben oder von einem Dritthersteller bezogen werden. Das führt in der Praxis aber zu Problemen abstrakterer Natur. Grundsätzlich ist einem Rahmenwerk immer eine gewisse Vorstellung über die Struktur der auf ihm basierenden Programme und der Art des Einsatzkontextes inhärent. Sei es, daß das Rahmenwerk selbst auf Klassen anderer Packages basiert oder daß es feste Annahmen über den Daten- und Kontrollfluß macht. Werden unterschiedliche Rahmenwerke innerhalb einer Anwendung verwendet, kann das Problem auftreten, daß die Annahmen der verschiedenen Rahmenwerke miteinander kollidieren. Garlan et al. haben für Probleme dieser Art den Begriff *Architectural Mismatch* geprägt [GAO95].

Auch die Packages des JDK implizieren konkrete Architekturvorstellungen und besitzen vielfältige Abhängigkeiten zu anderen JDK-Packages. Ersetzt man die Funktionalität eines JDK-Package durch eine eigene Implementation, die einem in einem konkreten Kontext geeigneter erscheint, muß auch immer kritisch geprüft werden, welche Auswirkungen des Architectural Mismatch dies hat. Ein Beispiel soll dieses Problem verdeutlichen:

Das Java-Swing Rahmenwerk enthält bereits Klassen zur Realisierung einer Undo/Redo-Funktionalität. Das Prinzip des Undo-Mechanismus funktioniert dabei folgendermaßen: Nach dem Ausführen einer verändernden, aber widerrufbaren Aktion informiert das für die Veränderung verantwortliche Objekt interessierte Beobachter (`UndoableEditListener`) über den Java-Ereignismechanismus. Als ein Parameter des Ereignisobjekts übergibt es hierbei ein Objekt der Klasse `UndoableEdit`. `UndoableEdit` beschreibt den Effekt einer durchgeführten Änderung und besitzt die für das Rückgängigmachen erforderlichen Operationen `Undo()` und `Redo()`. Ein spezieller Beobachter der `UndoableEditEvents` ist der Undo-Manager. Der Undo-Manager sammelt die empfangenen Effektobjekte und bietet eine Schnittstelle für ein kontrolliertes Rücksetzen der Aktionen. Das Interaktionsdiagramm in Abbildung 38 zeigt das Verhalten an einer Änderung in einem Objekt der Klasse `Document`.

Einen anderer gebräuchlichen Ansatz zur Realisierung der Undo/Redo-Funktionalität beschreibt das Entwurfsmuster *Command Processor* [Sommerlad95]. Hier werden widerrufbare Aktionen in Befehlsobjekten (Commands) gekapselt, welche zur Ausführung an einen Befehlsprozessor übergeben werden. Ein Befehlsobjekt kapselt sowohl die eigentlich verändernde Aktion (`doIt()`-Operation), als auch ihr Rückgängigmachen (`undo()`-Operation). Ein kontrolliertes Rücksetzen erfolgt durch einen Auftrag an den Befehlsprozessor bei dem zuletzt ausgeführten Befehlsobjekt die undo-Operation aufzurufen. Im Unterschied zu einem `UndoableEdit` beinhaltet ein Befehlsobjekt hier also, nicht den Effekt einer bereits durchgeführten Aktion, sondern die Aktion selbst.

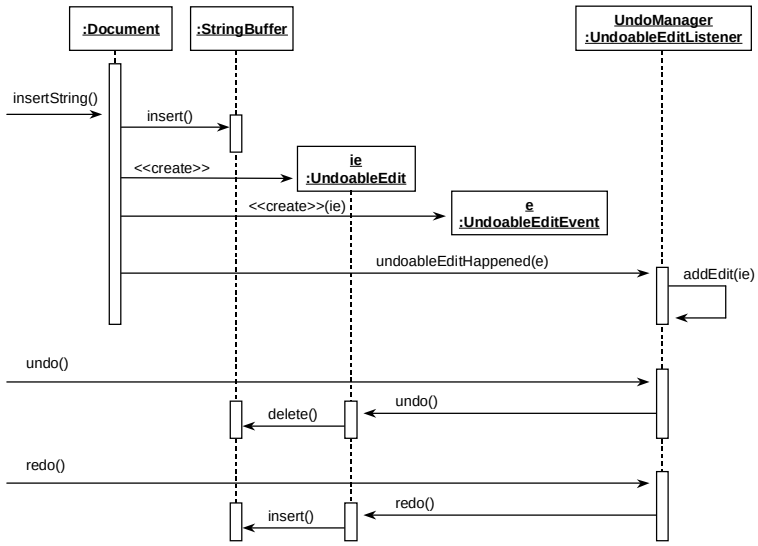

Abbildung 38: Swing Undo-Package

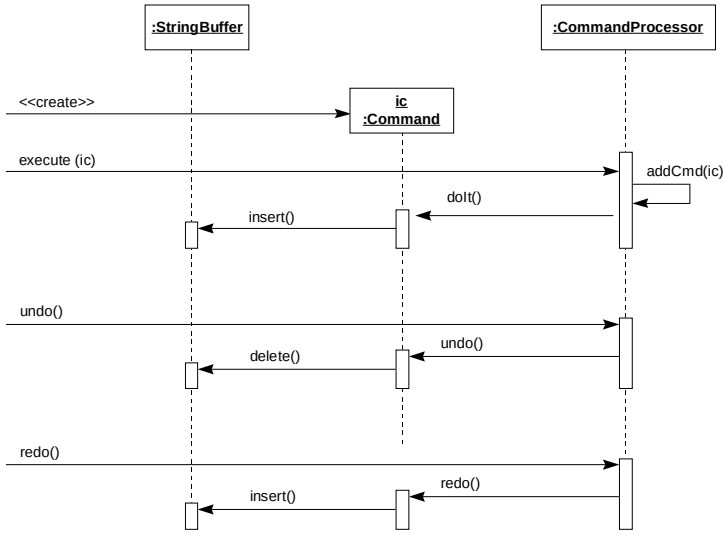

Abbildung 39: Undo Command Processor

Welchen der beiden Ansätze man dem Vorzug gibt, ist letztendlich eine Frage des softwaretechnischen Geschmacks. Für neu zu entwickelte Klassen unterscheidet sich auch der Programmieraufwand nicht nennenswert. Werden in der Anwendung aber u.a. auch die GUI-Klassen des Swing-Rahmenwerks eingesetzt, führt der zweite Ansatz zu einem nicht unerheblichen Architectural Mismatch, da keine der Swing-Klassen direkt die erforderlichen Befehlsobjekte unterstützt. Effektobjekte sind dagegen ein integraler Bestandteil der Swing-Klassen. Die Realisierung eines Undo-Mechanismus in einer Swing-Umgebung gestaltet sich mit Effektobjekten sehr viel einfacher als über einen Befehlsprozessor. In Rahmen dieser Diplomarbeit basiert der Undo-Mechanismus daher auf Effektobjekten.

Grundsätzlich wurde in dieser Arbeit das Prinzip verfolgt, den Architectural Mismatch zu den Standardklassen des JDK möglichst gering zu halten. Wann immer möglich, wurden die Konzepte der JDK-Klassen übernommen oder adaptiert.

5.6.3 Abgrenzung zum Java Beans-Standard

Komponentenorientiertung in Java wird meist direkt mit dem Java Beans-Modell gleichgesetzt (siehe Kapitel 3.1). Die hier vorgestellte Art Werkzeugkomponenten zu implementieren unterscheidet sich aber in einigen Punkten von typischen Java Beans-Implementierungen. Der wichtigste Unterschied ist, daß jede Komponentenschnittstelle mit einem separaten Java-Interface realisiert wird. Auch wenn es nicht der Java Beans-Spezifikation [JavaSoft96] widersprechen würde, so ist es doch unüblich Java Beans-Komponenten *ausschließlich* über explizite Java-Interfaces anzusprechen.

Ein weiterer Unterschied besteht in der Form, wie Komponenten zusammenarbeiten. Sowohl Werkzeugkomponenten, als auch Java Beans-Komponenten lassen sich in Hierarchien anordnen. Werkzeugkomponenten interagieren dabei entsprechend dem Bürokratiemuster [Riehle98]. Das bedeutet, die einzige Annahme, die eine Werkzeugkomponente über ihre übergeordnete Komponente macht ist, daß diese das `RequestHandler`-Protokoll versteht. Java Beans-Komponenten können dagegen von ihrer übergeordneten Komponente (`BeanContext`) dynamisch zur Laufzeit erfragen, welche Dienste diese anbietet. Hierdurch können untergeordnete Komponenten mit ihrer übergeordneten Komponente durchaus über mehreren verschiedenen Schnittstellen kommunizieren.

Im Java Beans-Standard gibt es zwei unterschiedliche Anwendungen des Java-Ereignismechanismus. Zum einen werden Ereignisse verwendet, damit eine Komponente über durchgeführte Zustandsänderungen informieren kann. Werkzeugkomponenten verwenden diese Form des Java-Ereignismechanismus um Ereignisse aufwärts in der Komponentenhierarchie zu propagieren.

Im Java Beans Komponentenmodell gibt es noch eine andere Anwendung des Ereignismechanismus: Wie in Kapitel 3.2 erwähnt, sind Beans-Eigenschaften (Properties) ein wesentliches Element des Modells. Properties sind diskrete, benannte Attribute einer Java Bean, deren Werte eine, wie auch immer geartete, Auswirkung auf das Verhalten oder Aussehen der Komponente haben. Es werden drei Klassen von Properties unterschieden: einfache, gebundene (bound properties) und beschränkte Eigenschaften (constraint properties). Alle Properties besitzen generische get- und/oder set-Operationen mit denen auf ihren Wert zugegriffen werden kann. Während ein Aufruf der set-Operation bei den einfachen Eigenschaften lediglich den inneren Zustand der Bean verändert, propagieren gebunden und begrenzte Eigenschaften eine Wertänderung zusätzlich über Ereignisse (`PropertyChangeEvent`). Hierbei informieren Ereignisse von gebundenen Eigenschaften registrierte Beobachter, wie oben beschrieben, *nachdem* die Zustandsänderung durchgeführt worden ist. Bei der Wertänderung eines beschränkten Property, sendet die Bean dagegen ein Ereignisobjekt bereits *vor* der Änderung an die registrierten Beobachter vom Typ `VetoableChangeListener`. Jeder der Beobachter hat darauf hin die Möglichkeit sein Veto gegen die bevorstehende Änderung auszusprechen indem er eine `PropertyVetoException` wirft. In diesem Fall wird die Änderung von der Bean nicht durchgeführt.

Diese Anwendung des Ereignismechanismus unterscheidet sich grundlegend von der ersten. Hier werden Ereignisse verwendet um eine gezielt Reaktion beim Beobachter auszulösen. Während im ersten Anwendungsfall die ereignisaus-

sendende Bean-Komponente unabhängig von eventuellen Reaktionen der Beobachter agiert, basiert im zweiten Fall ihre Verarbeitung maßgeblich auf den Reaktionen der Beobachter. Nach dem Verständnis des Bürokratiemusters ist das ein Mißbrauch des Ereignismechanismus. Gedacht ist der Veto-Mechanismus für Anwendungsfälle, in denen eine Bean-Komponente nicht in der Lage ist, vollständig zu entscheiden, ob eine Wertänderung augenblicklich sinnvoll und zulässig ist oder nicht. Der ausführenden Bean fehlt es offensichtlich an Wissen und Überblick um die volle Verantwortung für eine Änderung zu übernehmen. Auch im Kontext von Werkzeugkomponentenhierarchien gibt es Situationen, in denen eine Komponente einen Auftrag erhält, dessen Ausführung sie nicht vollständig selbst verantworten kann. Für diese Fälle sind die Verantwortlichkeiten jedoch klar definiert: Jeden Auftrag den eine Komponente nicht selbst ausführen kann, delegiert sie über die Zuständigkeitskette an die ihr hierarchisch übergeordneten Komponente zur Ausführung weiter. Die strenge hierarchische Anordnung von Werkzeugkomponenten fördert ein klares Verständnis der Verantwortlichkeiten. Veto-Ereignisse wurden deshalb bei der Implementierung in dieser Arbeit nicht verwendet.

Die Semantik von Veto-Ereignissen ist schwieriger, als es auf den ersten Blick erscheint. Es muß die Frage eindeutig geklärt sein, wie eine ereignisaussendende Komponente auf ein Veto zu reagieren hat. Darf sie ihren Änderungswunsch ohne Konsequenzen einfach streichen? Muß sie ihren eignen Auftraggeber über das Veto informieren? Wer trägt die Verantwortung für die korrekte Fortsetzung der Verarbeitung: die ändernde Komponente oder die Komponente, die das Veto ausgesprochen hat? Wie sieht eine alternative Bearbeitung aus? Darf ein Änderungsauftrag nach einem Veto wiederholt werden? Wie oft? Spätestens, sobald es mehr als einen Beobachter auf ein Veto-Ereignis gibt, kann das den Kontrollfluß des Systems schwer nachvollziehbar werden lassen. Auf der anderen Seite bietet der Veto-Mechanismus die Möglichkeit, eine Komponente, abhängig von dem Kontext, in dem sie eingebettet ist, sehr flexibel agieren zu lassen. Ohne einen Veto-Mechanismus müssen die Fähigkeiten und Befugnisse einer Komponente immer a priori genau und vollständig festgelegt werden. Es wurde im Rahmen dieser Diplomarbeit nicht weiter untersucht, inwieweit bei der Interaktion innerhalb einer Werkzeugkomponentenhierarchie eine Kombination aus Veto-Ereignissen und Zuständigkeitskette möglich wäre. Durch die Kombination der beiden Mechanismen ließe sich evtl. die Flexibilität des Veto-Mechanismus nutzen, ohne die klare Zuordnung der Zuständigkeiten aufzugeben. Ein Interaktionsszenario könnte wie folgt aussehen: Eine Komponente ist in Begriff eine Aktion durchzuführen, von der sie nicht absolut sicher sein kann, daß sie in ihrem aktuellen Einsatzkontext auch befugt dazu ist. Sie propagiert ein Veto-Ereignis. Erhält sie dann ein Veto, delegiert sie die Aktion über die Zuständigkeitskette an ihre übergeordnete Komponente weiter.

6 Zusammenfassung und Ausblick

Komponentenorientierte Ansätze erlangen zunehmendes Interesse auch im Bereich der Softwareentwicklung. In dieser Diplomarbeit wurde gezeigt, was Komponenten sind und daß eine komponentenorientierter Sichtweise sich nicht auf die Ebene des binären Codes beschränkt bleiben muß, sondern auch auf der abstrakten Architekturebene sinnvoll einsetzt werden kann. Softwarearchitekturen als Ganzes lassen sich gut als Systeme miteinander interagierender Komponenten modellieren. Die komponentenorientierte Modellierung abstrahiert hierbei von implementationstechnischen Details und betont die Komponentenschnittstellen und die abstrakte Interaktion über diese Schnittstellen. Eine komponentenorientierte Herangehensweise zeigt deutlich die *tragenden Wände* eines Softwaresystems und erleichtert dadurch die Abschätzung der Auswirkung von nachträglichen Systemänderungen. Die Betonung der Außenansicht von Komponenten und die Abstraktion von der inneren Implementation erleichtern die Wiederverwendbarkeit und das Plug-and-Play von Komponenten.

Auch für die Entwicklung von Softwarewerkzeugen ist ein komponentenorientierter Ansatz gut geeignet. Es wurde eine Modellarchitektur für komponentenorientierte Werkzeugarchitekturen vorgestellt, die auf hierarchisch angeordneten Werkzeugkomponenten basieren, die gemäß des Bürokratiemusters [Riehle98] miteinander interagieren. Die Modellarchitektur spezifiziert welche Schnittstellen eine Werkzeugkomponente erfüllen muß, damit sie auf einfache Weise in einem Softwarewerkzeug eingesetzt werden kann. Es wurde gezeigt, daß es sehr unterschiedliche Möglichkeiten gibt, Werkzeugkomponenten in der Programmiersprache Java zu implementieren. Die Stärke des komponentenorientierten Ansatzes liegt darin, daß die Komponentenimplementation hierbei weitgehend unabhängig von der Werkzeugarchitektur betrachtet werden kann. Das im WAM-Ansatz beschriebene Prinzip, der Aufgliederung eines Softwarewerkzeugs in eine Funktions- und eine Interaktionskomponente wird unter diesem Blickwinkel zu einer von mehreren Implementierungsmöglichkeiten. Kann eine Werkzeugkomponente beispielsweise durch einfache Adaption an eine existierende Komponente realisiert werden, ist durchaus auch eine einfachere Implementation gerechtfertigt, die nicht zwischen Interaktion und Funktion trennt.

Komponentenorientierung und Objektorientierung weisen an vielen Punkten konzeptionelle Ähnlichkeiten auf. Trotzdem muß die Komponentenorientierung als ein eigenes Konzept verstanden werden. So stellen in der Objektorientierung Klassen das Mittel zur Modellierung abstrakter Konzepte und Begriffe und gleichzeitig auch Elemente des Implementierungsmodells dar. Eine Klasse läßt sich dadurch nicht generell vollständig von Implementierungsaspekten trennen. Komplexere Einheiten, wie die hier behandelten Werkzeugkomponenten, werden jedoch nicht durch eine einzelne Klasse implementiert, sondern durch ein ganzes Klassenkonglomerat. Klassen sind häufig zu feingranular um diese komplexen Einheiten als Ganzes zu modellieren. In dieser Diplomarbeit wurde gezeigt, wie sich Komponentenorientierung und Objektorientierung ergänzen können. Komponentenorientiert modelliert Systeme lassen sich harmonisch mit objektorientierten Mitteln und Techniken implementieren.

Ein Fazit, daß ich aus dieser Arbeit ziehe ist, daß ein komponentenorientierter Ansatz sich gut zur Konstruktion von Softwarewerkzeugen eignet. Komponenten als eine zusätzliche Abstraktion über Klassen erlauben eine gute Modellierung von Werkzeugkomponenten als Ganzes. Auch die Umsetzung mit der Programmiersprache Java war ohne großen Modellbruch möglich. Vor allem die in Java vorgenommene Differenzierung von Klasse und Interface erlaubt eine harmonische Abbildung der Konzepte Komponentenklasse und Komponentenschnittstelle.

Auch wenn eine Modellarchitektur prinzipiell unabhängig vom verwendeten Implementierungsmodell sein sollte, hat sich gezeigt, daß sich Rückwirkungen von der Sprache zur Architektur nicht vollständig vermeiden lassen. Als Beispiele hierfür wurde der verwendete Undo-Mechanismus diskutiert. Es bleibt zukünftigen Arbeiten vorbehalten, die Frage zu klären, inwieweit sich die vorgestellte Modellarchitektur ohne Probleme auf andere Implementierungsmodelle abbilden ließe.

7 Literatur

[AG96] K. Arnold, J. Gosling; *The Java Programming Language;* Reading, Massachusetts: Addison-Wesley, 1996

[AGI98] R. Allen, D. Garlan, J. Ivers; *Formal Modeling and Analysis of the HLA Component Integration Standard;* ACM SIGSOFT Software Engineering Notes, Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering FSE-6, S. 70, 1998

[Bäumer98] D. Bäumer; *Softwarearchitekturen für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme;* Dissertation an der Universität Hamburg im Fachbereich Informatik, 1998

[BRJ99] G. Booch, J. Rumbauch, I. Jacobsen; *The Unified Modeling Language User Guide;* Reading, Massachusetts: Addison-Wesley, 1999

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal; *Pattern-Oriented Software Architecture - A System of Patterns;* Chichester, New York: Wiley & Sons Ltd, 1996

[BCK97] L. Bass, P. Clements, R. Kazman; *Software Architecture in Practice;* Reading, Massachusetts: Addison-Wesley, 1997

[Cox87] B. Cox; *Building malleable Systems from Software 'Chips';* Computerworld, März 1987

[CW85] L. Cardelli, P. Wegner; *On Understanding Types, Data Abstraction, and Polymorphism;* Computing Surveys, 17(4), Dezember 1985

[Denning97] A. Denning; *ActiveX Controls Inside Out;* Redmond, Washington: Microsoft Press, 1997

[DK76] F. DeRemer, H. Kron; *Programming-in-the-Large Versus Programming-in-the-Small;* IEEE Transaction on Software Engineering SA-2,2, Juni 1996

[DW98] D. F. D'Souza, A. C. Wills; *Objects, Components, and Frameworks with UML - The Catalysis Approach;* Reading, Massachusetts: Addison-Wesley, 1998

[FLL+98] N. Fricke, C. Lielienthal, M. Lippert, S. Roock, H. Wolf; *Operating and Window Systems will never strike back or Independence day for Java developers;* in: R. Nigel Horspool (Ed): Systems Implementation 2000, Chapman & Hall, Proceedings of SI2000 (IFIP Working Group 2.4), London, New York, 1998

[Gamma98] E. Gamma; *Extension Object;* in: Pattern Languages of Program Design 3; Hrsg.: R. C. Martin, D. Riehle, F. Buschmann; Reading, Massachusetts: Addison-Wesley, Kapitel 1, S. 79-88, 1998

[GAO95] D. Garlan, R. Allen, J. Ockerbloom; *Architectural Mismatch or Why it's hard to build systems out of existing parts;* Proceedings of the Seventeenth International Conference on Software Engineering, 1995

[Garlan95] D. Garlan; *What is Style?;* Proceedings of First International Workshop Software Architecture, April 1995

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides; *Design Patterns: Elements of Reusable Design;* Reading, Massachusetts: Addison-Wesley, 1995

[GMW95] D. Garlan, R. Monroe, D. Wile; *ACME: An Architectural Interconnection Language;* Technical Report CMU-CS-95-219; Carnegie Mellon University; 1995

[Goldberg90] A. Goldberg; *Information Models, Views, and Controllers;* Dr. Dobb's Journal, Juli 1990, S. 54-60

[Goodman97] D. Goodman; *Official Marimba Guide to Bongo;* sams.net, 1997

[Griffel98] F. Griffel; *Componentware: Konzepte und Techniken eines Softwareparadigmas;* Heidelberg: dpunkt-Verlag, 1998

[JF88] R. E. Johnson, B. Foote; *Designing Reusable Classes;* The Journal of Object-Oriented Programming, 1(2), Juni/Juli 1988, S. 22-35

[JavaSoft96] JavaSoft; *JavaBeans 1.0 API Specification, Version 1.00-A;* http://java.sun.com/beans; Dezember 1996

[JavaSoft98] JavaSoft; *Enterprise JavaBeans Specification, Version 1.0;* http://java.sun.com/products/ejb/docs.html; März 1998

[Johnson97] R. E. Johnson; *Frameworks = (Components + Patterns);* Communications of the ACM, Vol. 40, No. 10, S. 39-42, Oktober 1997

[KGZ93] Klaus Kilberth, Guido Gryczan, Heinz Züllighoven; *Objektorientierte Anwendungsentwicklung*; Braunschweig/Wiesbaden: Vieweg, 1993

[KP88] G. E. Krasner, S. T. Pope; *A Cookbook for Using the Model-View-Controler User Interface Paradigm in Smalltalk80*; Journal of Object-Oriented Programming, 1(3), 1988

[Krauß98] T. O. Krauß; *Softwarekonstruktion nach WAM unter Nutzung des MFC Rahmenwerks*; Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1998

[Krueger92] C. W. Krueger; *Software Reuse;* ACM Computing Surveys, 24(2), Juni 1992

[Lehman80] M. Lehmann; Programs, *Life Cycles, and Laws of Software Evolution; Proceedings of the IEEE*, 68/9, 1980

[Lippert97] M. Lippert; *Konzeption und Realisierung eines GUI-Frameworks in Java nach der WAM Metapher;* Studienarbeit am Fachbereich Informatik, Arbeitsbereich Softwaretechnik der Universität Hamburg, 1997

[Liskov88] B. Liskov; *Data Abstractopn and Hierarchy;* OOPSLA '87 Addendum to the Proceedings, ACM SIGPLAN Notices, 23(5), 1988, S. 17-34

[LKA+95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, W. Mann; *Specification and Analysis of System Architecture Using Rapide;* IEEE Transactions of Software Engineering, S. 336-355, 1995

[McLennan82] B. J. MacLennan; *Values and Objects in Programming Languages;* ACM SIGPLAN Notices, 17(12), 1982

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, J. Kramer; *Specifying Distributed Software Architecture;* Proceedings of the Fifth European Software Engineering Conference (ESEC 95), 1995

[MDK93] J. Magee, N. Dulay, J. Kramer; *Structuring Parallel and Distributed Programs;* IEEE Software Engineering Journal, S. 73-82, 1993

[Medvidovic97] N. Medvidovic; *A Classification and Comparison Framework for Software Architecture Description Languages*, Department of Information and Computer Science, University of California, Irvine, Technical Report UCI-ICS-97-02, 1997

[Meyer90] B. Meyer; *Objektorientierte Softwareentwicklung;* München: Hanser, London: Prentice Hall, 1990

[MKMG97] R. T. Monroe, D. Kompanek, R. Melton, D. Garlan; *Architecture Styles, Design Patterns, and Objects;* IEEE Software, 14(1), Januar 1997, S. 43-52

[MNR97] R. C. Martin, J. W. Newkirk, B. Rao; *Taskmaster: An Architecture Pattern for GUI Applications;* C++ Report, März 1997

[MQR95] m. Moriconi, X. Qian, R. A. Riemenschneider; *Correct Architecture Refinment;* IEEE Transaction on Software Engineering, 21(4), 1995

[MT97a] N. Medvidovic, R. N. Taylor; *A Framework for Software Architecture Description Languages;* in Proceedings of the Sixth European Software Engineering Conference together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering, S. 60-76, Zürich, 1997

[MT97b] N. Medvidovic, R. N. Taylor; *Exploiting Architectural Style to Develop a Family of Applications;* IEEE Proceedings Software Engineering, 1997

[MTW96] N. Medvidovic, R. N. Taylor, E. J. Whitehead; *Formal Modeling of Software Architectures at Multiple Levels of Abstraction;* in Proceedings of the California Software Symposium 1996, S. 28-40; Los Angeles, Carlifornia, 1996

[ND95] O. Nierstrasz, L. Dami; *Component-Oriented Software Technology;* in O. Nierstrasz, D. Tsichritzis: Object-Oriented Software Composition; London: Prentice Hall, 1995

[NT95] O. Nierstrasz, D. Tsichritzis; *Object-Oriented Software Composition;* London: Prentice Hall, 1995

[OHE96] R. Orfali, D. Harkey, J. Edwards; *The Essential Distributed Object Survival Guide;* New York: John Wiley & Sons Inc., 1996

[OMG95] Object Management Group; *Common Object Request Broker: Architecture and Specification. Revision 2.0;* OMG technical document 96-03-04, 1995

[OMT98] P. Oreizy, N. Medvidovic, R. T. Taylor; *Architectur-Based Runtime Software Evolution;* Proceedings of the International Conference on Software Engineering 1998 (ICSE98); Kyoto, Japan, 1998

[Parnas72] D. L. Parnas; *On the Criteria to be Used in Decomposing Systems into Modules;* Communication of the ACM, 5(12), S. 1053-1058, 1972

[PW92] D. E. Perry, A. L. Wolf; *Foundations for the Study of Software Architectur;* ACM SIGSOFT Software Engineering Notes, 17(4), S. 40-52; 1992

[RJB99] J. Rumbauch, I. Jacobsen, G. Booch; *The Unified Modeling Language Reference Manual;* Reading, Massachusetts: Addison-Wesley, 1999

[Riehle95] D. Riehle; *Muster am Beispiel der Werkzeug und Material Metapher;* Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1995

[Riehle98] D. Riehle; *Bureaucracy;* in: Pattern Languages of Program Design 3; Hrsg.: R. C. Martin, D. Riehle, F. Buschmann; Reading, Massachusetts: Addison-Wesley, Kapitel 3, S. 163-185, 1998

[RMRS98] J. E. Robbins, N. Medvidovic, D. F. Redmiles, D. S. Rosenblum; *Integrating Architecture Description Languages with a Standard Design Method;* Proceedings of the International Conference on Software Engineering 1998 (ICSE98), Kyoto, Japan, 1998

[SDK+95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, G. Zelesnik; *Abstractions for Software Architecture and Tools to Support Them;* IEEE Transaction on Software Engineering, S. 314-335, 1995

[SDZ96] M. Shaw, R. DeLine, G. Zelesnik; *Abstraction and Implementations for Achitectural Connections;* in Proceedings of the Third International Conference on Configurable Distributed Systems, 1996

[SEI97] Software Engineering Institute, Carnegie Mellon University; *Software Technology Review*; verfügbar unter http://www.sei.cmu.edu/str, 1997

[SG96] M. Shaw, D. Garlan; *Software Architecture - Perspectives on an Emerging Discipline;* London: Prentice Hall, 1996

[Sommerlad95] P. Sommerlad; *Command Processor;* in: Pattern Languages of Program Design 2; Hrsg.: J. M. Vlissides, J. O. Coplien, N. L. Kerth; Reading, Massachusetts: Addison-Wesley 1995, Kapitel 2, S. 63-74;

[StarDivision93] Star Division: *Star View C++ Klassenbibliothek Version 2.0 Benutzerhandbuch*, 1993

[Stroustrup91] B. Stroustrup; *The C++ Programming Language;* Second Edition, Reading, Massachusetts: Addison-Wesley, 1991

[Subarctic96] http://www.cc.gatech.edu/gvu/ui/sub_arctic/sub_arctic/doc/users_manual.html

[Szyperski98] C. Szyperski; *Component Software, Beyond Object-Oriented Programming;* Reading, Massachusetts: Addison-Wesley, 1998

[TMA+96] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, D. L. Dubrow; *A Component- and Message-Based Architectural Style for GUI Software;* IEEE Transaction on Software Engineering 22, 6, S. 390-406, 1996

[Udell94] J. Udell; *Componentware;* in Byte, Mai 1994

[WWWK94] J. Waldo, G. Wyant, A. Wollrath, S. Kendall; *A Note on Distributed Computing;* Sun Microsystems Laboratories, Technical Report SMLI-TR-94-29, 1994

[Willamowius97] J. Willamowius; *Framework-Evolution am Beispiel eines Frameworks zur Steuerung von TK-Anlagen;* Diplomarbeit am Fachbereich Informatik, Arbeitsbereich Softwaretechnik der Universität Hamburg, 1997

[Züllighoven98] H. Züllighoven; *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- & Material-Ansatz;* Heidelberg: dpunkt-Verlag, 1998