Diplomarbeit

Werkzeugkonstruktion mit Manipulatoren

Februar 2004

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Autor:

Simon Ditrich

Heinrich-Schulte-Höhe 21
22117 Hamburg

Matrikelnummer: 4724588

eMail: simon@kasim.net

Betreuer:

Prof. Dr. Heinz Züllighoven
Prof. Dr. Winfried Lamersdorf

Für Katrin

Danksagung

Ich möchte an dieser Stelle allen Menschen danken, die mir bei der Erstellung dieser Arbeit geholfen haben.

Ich danke Prof. Dr. Heinz Züllighoven und Prof. Dr. Winfried Lamersdorf für die Erst- und Zweitbetreuung dieser Arbeit.

Ich danke Stefan Roock, von dem die Idee zum Thema dieser Arbeit stammt und der mir bei allen Fragen rund um diese Arbeit – insbesondere zum Aufbau der Arbeit, zum ursprünglichen Manipulator-Konzept, sowie zu WAM und JWAM – zur Verfügung stand.

Auch bei Henning Wolf und Andreas Kornstädt möchte ich mich für ihre Antworten auf meine Fragen zum Einsatz von JWAM und Manipulatoren in Softwareprojekten bedanken.

Weiterhin möchte ich mich bei Katrin Steinki, Rolf Tyzuk, sowie bei Johanna und Erdal Özkan für das Korrekturlesen bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Kontext

Softwareprogramme müssen – um nützlich zu sein – mit ihrer Umwelt interagieren. Technisch motivierte Programme wie z. B. Middleware zur Unterstützung von verteilten Anwendungen oder zum Zugriff auf Datenbanken interagieren mit anderen Programmen über eine Programmierschnittstelle (*application programming interface*, API). Interaktive Anwendungen kommunizieren mit ihren Benutzern über eine – heutzutage meist grafische – Benutzungsschnittstelle.

Für interaktive Anwendungen hat es sich als vorteilhaft erwiesen, die Benutzungsschnittstelle von der fachlichen Logik des Programms zu entkoppeln. Aus dieser Erkenntnis heraus sind zahlreiche Architekturen entstanden, die eine Trennung der Benutzungsschnittstelle vom Rest des Programms ermöglichen sollen. Unterschiedliche Architekturen verfolgen außer der Entkopplung der Benutzungsschnittstelle unterschiedliche weitere softwaretechnische Ziele.

Die vorliegende Arbeit beschäftigt sich mit Architekturen und Mustern zur Entwicklung interaktiver Anwendungsprogramme. Diese Muster und Architekturen werden auf ihre Tauglichkeit zur Entwicklung von Werkzeugen nach dem Werkzeug & Material-Ansatz (WAM, s. [Zül98]) untersucht. Es soll eine neue Architektur für WAM-Anwendungen entwickelt werden, die universeller einsetzbar ist, als die z.Z. am häufigsten verwendete Architektur nach dem Funktion/Interaktion-Muster (FK/IAK, s. Abschnitt 3.3).

1.2 Der Architekturbegriff

Der Begriff der Architektur ist in der Softwaretechnik nicht eindeutig definiert. Martin Fowler schreibt dazu in [Fow03], Seite 1:

> "Architecture" is a term that lots of people try to define, with little agreement. There are two common elements: One is the highest-level breakdown of a system into its parts; the other, decisions that are hard to change. It's also increasingly realized that there isn't just one way to state a system's architecture; rather, there are multiple architectures in a system, and the view of what is architecturally significant is one that can change over a system's lifetime.

In [RP97], Seite 652 definieren Floyd und Züllighoven den Begriff „Architektur“ wie folgt:

> Im engeren Sinne wird unter einer Architektur die Aufteilung eines Softwaresystems in seine Komponenten (meist Module), deren Schnittstellen, die Prozesse und Abhängigkeiten zwischen ihnen, sowie die benötigten Ressourcen [...] verstanden. Allgemeiner umfaßt der Architekturbegriff auch die strukturellen, formalen, nicht an anwendungsfachlichen Inhalten orientierten Prinzipien und Organisationsformen von Software. Wesentliche Eigenschaften einer Softwarearchitektur sind z. B. die Umsetzung des Prinzips der Datenkapselung, die Modul- oder Klassenbildung und die Erweiterbarkeit von Strukturen [...].

Züllighoven unterscheidet in [Zül98], Seiten 324f weiterhin zwischen den Begriffen „Softwarearchitektur“ und „Modellarchitektur“:

> Eine Softwarearchitektur bezeichnet die Modelle und die konkreten Komponenten eines Softwaresystems in ihrem statischen und dynamischen Zusammenspiel. Sie kann selbst als explizites Modell dargestellt werden. Eine Softwarearchitektur beschreibt ein konkretes System in seinem Anwendungskontext.
>
> Eine Modellarchitektur beschreibt die allgemeinen Prinzipien hinter einer Softwarearchitektur. Sie umfaßt die grundlegenden Elemente, deren Verknüpfungen und Regeln, die für eine Softwarearchitektur gelten. Eine Modellarchitektur gibt Anleitung bei der softwaretechnischen Realisierung eines Softwaresystems.

Die vorliegende Arbeit benutzt den Begriff der Architektur entsprechend Züllighovens Definition des Begriffs „Modellarchitektur“. Dabei werden allerdings nicht alle Schichten eines Softwaresystems, sondern nur die Teile, die die Werkzeugkonstruktion betreffen, behandelt (Werkzeugarchitektur). Nicht behandelt werden tieferliegende Schichten von Softwaresystemen, wie z. B. Schichten zum Zugriff auf Datenbanken, verteilte Objekte, etc.

1.3 Werkzeugtypen

Werkzeuge unterstützen den Anwender bei der Erledigung wiederkehrender Arbeitsabläufe. Das WAM-Leitbild ist das des Arbeitsplatzes für eigenverantwortliche Expertentätigkeit. Vom Benutzer eines Werkzeugs wird also vorausgesetzt, dass er sich in der unterstützten Domäne auskennt. Der Anwender bestimmt die Arbeitsabläufe und die Reihenfolge, in der Arbeitsschritte ausgeführt werden, selbst.

Ein Werkzeug macht Materialien sichtbar und ermöglicht deren Bearbeitung. Materialien wiederum sind die Arbeitsgegenstände, mit denen der Anwender in seiner Domäne umgeht.

Je nach Anwendungsfall werden unterschiedliche Werkzeugtypen benötigt. Die Art, wie mit einem Werkzeug umgegangen wird, bestimmt den Typ des Werkzeugs. Die vorliegende

Arbeit beschäftigt sich mit Werkzeugen, die lokal auf dem Computer des Benutzer installiert sind und eine Repräsentation für grafische Benutzungsoberflächen (*graphical user interface*, GUI) besitzen. Im Folgenden wird dabei – analog zu den Anwendungstypen in Microsofts Entwicklungsumgebung MS Visual C++ – zwischen zwei Werkzeugtypen unterschieden: Dialogbasierte Werkzeuge und dokumentbasierte Werkzeuge.

1.3.1 Dialogbasierte Werkzeuge

Dialogbasierte Werkzeuge werden hauptsächlich mit Hilfe von Kontroll-Elementen (*Widgets*) bedient. Widgets sind z. B. Eingabefelder, Knöpfe, Auswahllisten, Baumstrukturen, Tabellen, etc.

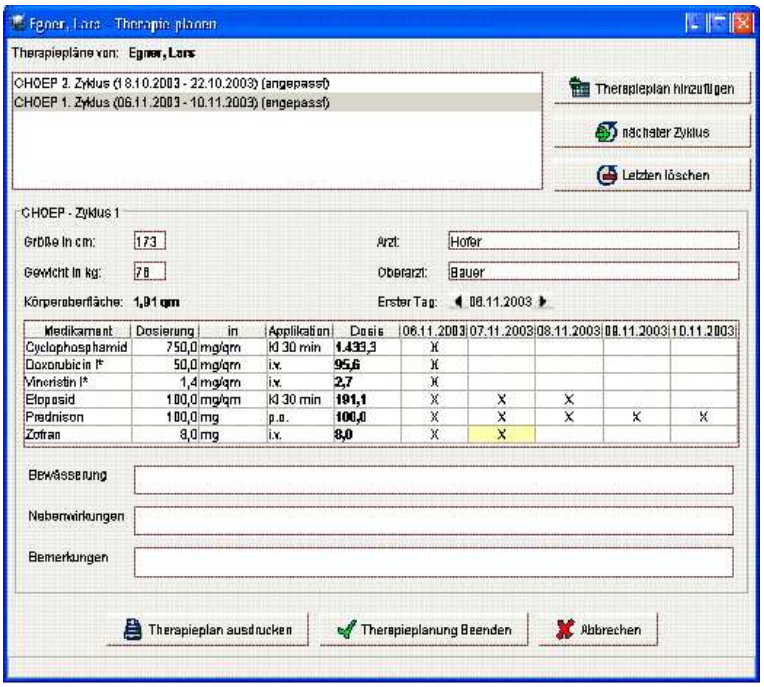

Abbildung 1.1: Das mobiVisit Werkzeug

Ein Beispiel für ein dialogbasiertes Werkzeug ist das in Abbildung 1.1 gezeigte mobiVisit. mobiVisit ist ein Werkzeug für Mediziner und dient der Erfassung von Haupt- und Nebendiagnosen. Mit Hilfe von mobiVisit können Kliniken die Qualität ihrer Behandlungsmaßnahmen belegen.

1.3.2 Dokumentbasierte Werkzeuge

Dokumentbasierte Werkzeuge dienen der Bearbeitung von Materialien, die einen Dokument-Charakter besitzen. Dies können z. B. Textdokumente oder Grafiken sein.

Zu den typischen dokumentbasierten Werkzeugen gehören u.a. Text-Editoren und Text-Verarbeitungen, Desktop-Publishing-Anwendungen, sowie Mal- und Zeichenprogramme.

Materialgeflechte können oft als Grafik dargestellt und bearbeitet werden. Ein Materialgeflecht ist eine Menge von Materialien, die miteinander in Beziehung stehen. Bearbeitbare Einheiten eines Materialgeflechts sind die Materialien selbst, sowie die Beziehungen, die zwischen den Materialien bestehen. Ein Werkzeug, mit dem ein Materialgeflecht bearbeitet werden kann, könnte z. B. ein UML-Werkzeug zur Gestaltung von Klassendiagrammen sein. In einem solchen Werkzeug entspricht ein Klassendiagramm dem Materialgeflecht, das aus Klassen (Material) sowie Beziehungen wie z. B. „Vererbung“ oder „Komposition“ besteht.

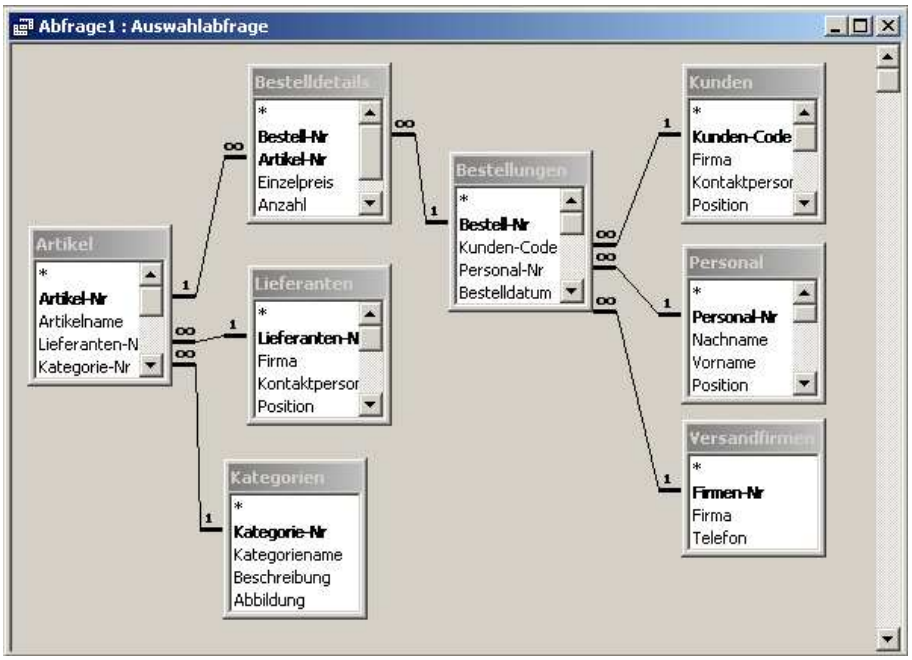

Abbildung 1.2: Grafischer Editor zum Erstellen von SQL-Anfragen in MS-Access

Ein weiteres Beispiel für ein Werkzeug, dass Materialgeflechte bearbeitbar macht, ist der in Abbildung 1.2 gezeigte Editor zum Erstellen von SQL-Anfragen in Microsoft Access. In diesem Beispiel sind die Datenbanktabellen die Materialien, die mittels WHERE-Klauseln miteinander verknüpft sind.

1.4 Problemstellung und Zielsetzung

Bei der Erstellung von WAM-Anwendungen mit Hilfe des Rahmenwerks JWAM (siehe [JWA]) wurde festgestellt, dass für grafische Werkzeuge die bisher verwendeten Entwurfsmuster und Architekturen nicht gut anwendbar sind (siehe Abschnitt 3.3).

Ein Ziel der vorliegenden Arbeit ist eine Vereinheitlichung der Werkzeugarchitekturen für

beide Werkzeugtypen im JWAM-Framework. Der Schwerpunkt soll dabei auf Architekturen für große, langlebige und interaktive Anwendungssysteme liegen, da solche Systeme in der Regel am meisten von der Verwendung einer Architektur profitieren.

Weiterhin wird untersucht, welche Architekturen es erlauben, ein Werkzeug um Funktionalität zu erweitern, ohne sehr viele Änderungen am bestehenden Programm-Code des Werkzeugs vornehmen zu müssen. Das Optimum stellt in dieser Hinsicht eine Architektur dar, die beim Hinzufügen von Funktionalität keinerlei Änderungen am bestehenden Code erfordert. Eine solche Plugin-Architektur (siehe Abschnitt 2.3) hätte den Vorteil, dass Funktionalität zu einem Anwendungssystem hinzugefügt oder ausgetauscht werden kann, ohne das System komplett neu installieren zu müssen. Plugin-Funktionalität könnte auch von Drittherstellern entwickelt werden.

1.5 Lösungsidee

Die aktuelle JWAM-Version enthält die Beispielanwendung *BasicModellingTool* (s. Abb. 1.3). Dieses Programm erlaubt es Benutzern, grafische Elemente mit der Maus frei zu verschieben und miteinander durch Verbindungspfeile in Beziehung zu setzen.

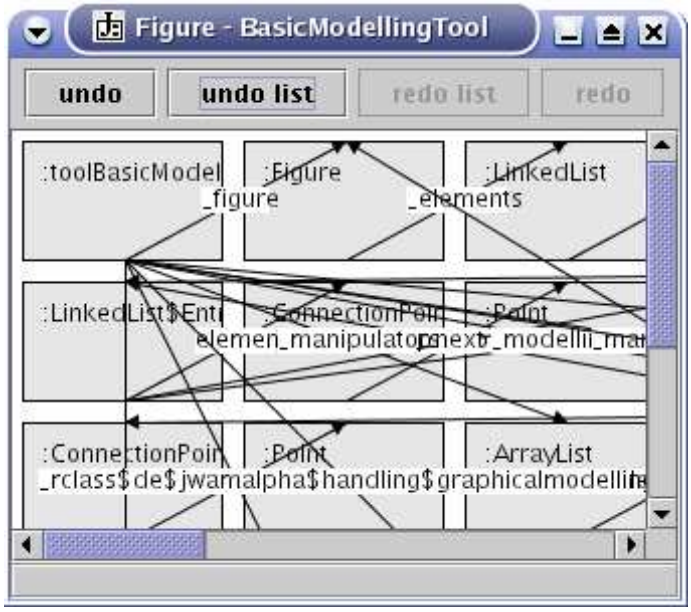

Abbildung 1.3: Das JWAM-BasicModellingTool

Anders als die meisten JWAM-Werkzeuge ist das BasicModellingTool nicht nach dem FK/IAK-Muster aufgebaut. Die Materialien werden mittels sogenannter Manipulatoren bearbeitet. Das verwendete Manipulator-Muster ist an das Befehlsmuster aus [GHJV96] angelehnt.

Ein Manipulator nimmt genau eine Art von Manipulation an genau einer Materialart vor. Für jede Manipulation (Erzeugen, Löschen, Verschieben und Verbinden von Elementen) wurde eine Manipulator-Klasse implementiert.

Das BasicModellingTool kennt Manipulatoren nur unter einer abstrakten Schnittstelle. Manipulatoren können am Werkzeug registriert werden. Wenn der Benutzer ein bearbeitbares Element mit der rechten Maustaste anklickt, zeigt das Werkzeug ein Kontextmenü mit den Namen aller registrierten Manipulatoren an, die das Element bearbeiten können.

Die Lösungsidee für eine neue Werkzeugarchitektur besteht in der Verwendung von Manipulatoren statt Funktions- und Interaktionskomponenten.

1.6 Gang der Untersuchung

Zunächst werden in Kapitel 2 die Anforderungen an eine optimale Werkzeugarchitektur aufgestellt. In unterschiedlichen Architekturen werden unterschiedliche Begriffe mit ähnlicher Bedeutung verwendet. Deshalb werden in diesem Kapitel außerdem einige im WAM-Kontext gebräuchliche Begriffe definiert. Die aus unterschiedlichen Architekturen stammenden Begriffe werden dann in den jeweiligen folgenden Kapiteln mit den hier definierten Begriffen in Beziehung gesetzt.

Daraufhin werden in Kapitel 3 bekannte Architekturmuster diskutiert, die jeweils einige der im vorangegangenen Kapitel aufgestellten Anforderungen erfüllen. Die Muster werden daraufhin untersucht, ob sie den Anforderungen einer WAM-Anwendung gerecht werden und ob sie mit anderen Architekturen derart kombiniert werden können, dass als Ergebnis eine Architektur herauskommt, die möglichst viele dieser Anforderungen erfüllt.

In Kapitel 4 werden verschiedene GUI-Toolkits und Application Frameworks auf weitere für die Zielsetzung interessante Muster untersucht. Außerdem wird untersucht, ob und wie diese Toolkits und Frameworks die in Kapitel 3 diskutierten Architekturen umsetzen.

In Kapitel 5 wird unter Verwendung der in den Kapiteln 3 und 4 erarbeiteten Erkenntnisse ein Architekturmuster erarbeitet, das den in Kapitel 2 beschriebenen Anforderungen gerecht werden soll. Als Ausgangsbasis für eine solche Architektur dient das in Abschnitt 1.5 erwähnte Manipulator-Muster.

Kapitel 6 zeigt eine Umsetzung des in Kapitel 5 konzipierten Manipulator-Musters in Java anhand eines Werkzeugs zur Erstellung von UML-Klassendiagrammen. Dieses Werkzeug wird sowohl die Eigenschaften eines dokumentbasierten Werkzeugs, als auch die eines dialogbasierten Werkzeugs besitzen. Ausserdem wird das Werkzeug in mehreren Versionen unter Verwendung verschiedener GUI-Toolkits konstruiert.

1.7 Notation

Für die Darstellung von Diagrammen wird in der vorliegenden Arbeit die UML-Notation (*Unified Modeling Language*, s. [RJB99]) verwendet. Ausnahmen sind Diagramme, die aus Publikationen übernommen wurden, die eine andere Notation als UML verwenden.

2 Anforderungen an eine Werkzeugarchitektur

Das Ziel der vorliegenden Arbeit ist es, eine einheitliche Werkzeugarchitektur für die Konstruktion von Werkzeugen nach dem WAM-Ansatz zu erarbeiten. Dieses Kapitel beschreibt und begründet die Anforderungen, die an eine solche vereinheitlichte Architektur gestellt werden.

Die folgende Aufzählung gibt einen stichwortartigen Überblick über die Anforderungen an die zu entwickelnde Architektur:

- Trennung von Funktion und Interaktion
- Modularisierung der Funktionalitäten
- Dynamischer Aufbau
- Universalität und Skalierbarkeit
- Umsetzbarkeit

Die folgenden Abschnitte erläutern und begründen diese Anforderungen.

2.1 Trennung von Funktion und Interaktion

Die Trennung von Funktion und Interaktion ist eine Anforderung, die von den meisten Architekturen berücksichtigt wird. In unterschiedlichen Architekturen werden dabei unterschiedliche Begriffe, die die gleiche oder eine ähnliche Bedeutung wie „Funktion“ und „Interaktion“ im WAM-Kontext haben, verwendet.

Definition: Funktion (nach [Zül98], S. 235)

> Die fachliche Funktionalität eines Werkzeugs Bezeichnen wir als seine *Funktion*.

Die Funktion eines Werkzeugs umfaßt nicht die gesamte Fachlogik einer Anwendung. Der vom Benutzungsmodell (s.u.) unabhängige Teil der Fachlogik befindet sich im fachlichen Modell (*Domain Model*, siehe [Fow03], Seiten 166ff) der Anwendung. Die Funktion eines Werkzeugs enthält den Teil der Fachlogik, der vom Benutzungsmodell bestimmt wird.

Definition: Benutzungsmodell (nach [Zül98], S. 71)

> Ein Benutzungsmodell ist ein fachlich motiviertes Modell darüber, wie Anwendungssoftware bei der Erledigung der anstehenden Aufgaben im jeweiligen Einsatzkontext benutzt werden kann. Das Benutzungsmodell umfaßt eine Vorstellung von der Handhabung und Präsentation der Software aber auch von den fachlichen Gegenständen, Konzepten und Abläufen, die von der Software unterstützt werden. [...]

Definition: Interaktion (nach [Zül98], S. 235)

> Die Art und Weise der Handhabung und die Form der Präsentation eines Werkzeugs bezeichnen wir als seine *Interaktion*.

Das bekannteste Architekturmuster, das die Funktionalität von der Benutzungsschnittstelle trennt, ist das in Abschnitt 3.1 diskutierte MVC-Muster. Das auf Windows-Plattformen am häufigsten eingesetzte Muster ist das vom MFC-Framework (Microsoft Foundation Classes) verwendete Dokument/View-Muster, das eine Vereinfachung des MVC-Musters darstellt. Im WAM-Kontext kommt meist das FK/IAK-Muster (siehe Abschnitt 3.3) zum Einsatz.

Im Folgenden wird diskutiert, warum gerade die Entkopplung von Funktion und Interaktion Bestandteil vieler Architekturen ist.

2.1.1 Unabhängiger Entwurf der Funktion

Eine Trennung von Funktion und Interaktion erlaubt es, die fachliche Funktionalität unabhängig von der Benutzungsschnittstelle zu entwerfen und zu implementieren. Dadurch kann der Entwurf der Funktion auf einer abstrakteren Ebene stattfinden, so dass die konkrete Umsetzung des in der Funktion vorgegebenen Benutzungsmodells in der Interaktion nicht bekannt sein muss.

Damit die Funktion unabhängig von der Interaktion wird, darf die Funktion die Interaktion nicht „kennen“, d.h., dass keine direkten Aufrufe von der Funktion an die Interaktion erfolgen dürfen. Stattdessen sollte die Interaktion mittels eines Reaktionsmechanismus' (vgl. z. B. Beobachtermuster in [GHJV96], oder Reaktionsmuster in [RW96]) über Zustandsänderungen der Funktion und des fachlichen Modells informiert werden.

Die Unabhängigkeit der Funktion erhöht deren Stabilität. Nach Martin (siehe [Mar97]) ist die Stabilität eines Moduls ein Maß dafür, wie häufig Änderungen an dem Modul vorgenommen werden müssen. Je seltener ein Modul verändert wird, desto stabiler ist es. Gleichzeitig ist die Stabilität eines Moduls ein Maß der Unabhängigkeit von anderen Modulen. Je weniger ein Modul von anderen Modulen abhängig ist und je mehr andere Module Abhängigkeiten zu diesem Modul besitzen, desto höher ist die Stabilität des Moduls.

In der Regel ist die Interaktion eines Werkzeugs häufigeren Änderungen unterworfen als die Funktion (siehe Abschnitt 2.1.3). Die Interaktion ist also weniger stabil als die Funktion.

Nach Martin ([Mar97], Seite 8) sollte die Funktion daher keine Abhängigkeiten zur Interaktion besitzen:

> The dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable than it is.

2.1.2 Wiederverwendbarkeit der Funktion

Die Entkopplung zweier Softwarekomponenten resultiert in einer besseren Wiederverwendbarkeit von mindestens einer der beiden Komponenten. Wenn die Funktion unabhängig von der Interaktion ist, kann die Funktion mit anderen Interaktionen wiederverwendet werden. Die Notwendigkeit der Wiederverwendung der Funktion mit einer neuen Interaktion ergibt sich, wenn ein Werkzeug auf ein anderes GUI-Toolkit – z. B. im Rahmen einer Portierung auf ein anderes System – umgestellt werden soll.

Die Möglichkeit der Wiederverwendung der Funktion eines Werkzeugs ist besonders wichtig für Werkzeuge, die in einer plattformunabhängigen Programmiersprache entwickelt werden, die keine Standardbibliothek zur Entwicklung einer GUI besitzt (z. B. C++), da bei einer Portierung eines solchen Werkzeugs oft ein neues GUI-Toolkit eingesetzt werden muss[1].

Aber auch wenn eine Programmiersprache wie Java verwendet wird, die inzwischen bereits zwei Standard-GUI-Frameworks[2] kennt und für die es mindestens eine weitere Alternative[3] gibt, kann eine solche Trennung sinnvoll sein, wenn man sich nicht auf ein Toolkit festlegen möchte. So könnte für ein Werkzeug, das zunächst das Swing-Toolkit zur Implementation seiner Interaktion verwendet, die Notwendigkeit entstehen, dass beispielsweise aus Gründen der Laufzeiteffizienz ein anderes Toolkit wie z. B. das Standard Widget Toolkit (SWT) eingesetzt werden muss.

2.1.3 Anpassbarkeit der Interaktion

Für die meisten Werkzeuge gilt, dass die Interaktion eines Werkzeugs häufigeren Änderungen unterworfen ist als die Funktion des Werkzeugs.

Martin begründet dies in [Mar03] auf Seite 359 wie folgt:

> User interfaces are volatile. They are subject to the whims of customers, marketing people, and nearly everyone else who comes in contact with the product. It seems very likely that if any part of the system suffers requirements thrashing, it will be the user interface. Therefore, we should decouple it first.

[1] In der Regel wird mehr als nur die GUI neu implementiert werden müssen. Die aktuelle C++-Standardbibliothek (s. [ISO98], S. 311ff) enthält z. B. keine APIs für Multi-Threading, verteilte Objekte oder XML-Verarbeitung. Es existieren allerdings plattformunabhängige Bibliotheken von Drittherstellern sowie aus dem Open-Source-Bereich.

[2] *Abstract Windowing Toolkit* (AWT, s. [GYJT97]) und *Java Swing*, (s. [ELW98])

[3] *Standard Widget Toolkit* (SWT, s. [Nor01])

Damit die Überarbeitung der Interaktion keine Anpassungen der Funktion erfordern, darf die Funktion keine Abhängigkeiten zur Interaktion aufweisen. Auf Seite 234 fordert Züllighoven in [Zül98]:

> Ein Werkzeug soll so in einen interaktiven und einen funktionellen Teil aufgeteilt werden, daß beide Teile softwaretechnisch weitgehend unabhängig voneinander modelliert und konstruiert werden können. Besonders wichtig ist, daß wir den interaktiven Teil verändern können, ohne den funktionellen Teil anpassen zu müssen.

Besonders bei Werkzeugen, die in einem iterativen Prozeß wie z. B. *Extreme Programming* (XP, s. [Bec99]) unter Einbeziehung des späteren Benutzers entwickelt werden, ist die Benutzungsschnittstelle häufigen Änderungen unterworfen. Cockburn schreibt in [Coc97], S. 10:

> Iterative development involves reworking a part of the system to improve the quality of the system. The user interface and the infrastructure of a system are two parts of software systems that benefit the most from the improvement. The user interface is improved to better serve the users, and the infrastructure is improved to simplify the evolution of the system.

2.1.4 Unterschiedliche Ansichten auf gleiche Informationen

Für einige Anwendungen kann es sinnvoll sein, dem Benutzer unterschiedliche Arten der Präsentation für die gleichen Informationen anzubieten. Martin Fowler schreibt in [Fow03], Seite 3 in Bezug auf große Geschäftsanwendungen:

> It's not unusual to have hundreds of distinct screens. Users of enterprise applications vary from occasional to regular, and normally they will have little technical expertise. Thus, the data has to be presented lots of different ways for different purposes.

Durch eine Entkopplung von Funktion und Interaktion können mehrere Interaktionen – auch gleichzeitig – von einer Funktion „bedient“ werden.

2.2 Modularisierung der Funktionalitäten

Die einzelnen fachlichen Funktionalitäten eines Werkzeugs sollen unabhängig voneinander implementiert werden können. Sie sollen im Code derart voneinander getrennt sein, dass die Funktion eines Werkzeugs nicht aus einer einzelnen monolithischen Klasse besteht, sondern dass jede Funktionalität in einer eigenen Klasse abgebildet wird.

Diese Anforderung ergibt sich schon aus dem Prinzip der maximalen Kohäsion (s. [Zül98], S. 43) bzw. dem *Single-Responsibility Principle* (SRP, s. [Mar03], S. 95ff). Ein Werkzeug, dessen Funktion nicht modularisiert ist, besteht meist aus einem Funktions-Objekt mit Methoden für die einzelnen Funktionalitäten. Bei einer Änderung irgendeiner Funktionalität wird immer die gleiche Klasse geändert. Dies wiederspricht dem SRP, das Martin in [Mar03] auf Seite 95 wie folgt zusammenfasst:

> A class should have only one reason to change.

Die Funktionalitäten eines Werkzeugs sollen so auf Funktions-Klassen aufgeteilt werden, dass sich zwischen den Methoden innerhalb einer Funktions-Klasse eine maximale Kohäsion ergibt.

Wenn die Funktionalitäten so weit modularisiert werden, dass für jede einzelne Funktionalität ein eigenes Objekt im Werkzeug existiert, das die jeweilige Funktionalität ausführt, kann das Werkzeug recht einfach mit einer Undo-/Redo-Funktionalität ausgestattet werden (vgl. z. B. Befehlsmuster, *command pattern*, in [GHJV96] und Rückgängigmachen/Wiederholen (*undo/redo*, in [Ost03]).

2.3 Dynamischer Aufbau

Ein Werkzeug sollte dynamisch aus Funktionalitäten aufgebaut werden können. Eine Werkzeug-Architektur soll es also ermöglichen, ein Werkzeug zur Laufzeit um Funktionalität zu erweitern. Diese Anforderung bedingt die Modularisierung der Funktionalitäten. Eine solche Architektur wird auch als *Plugin*-Architektur bezeichnet (siehe [Fow03], Seite 499ff).

Bekannte Programme mit Plugin-Architektur sind z. B. Adobe Photoshop und das Eclipse-Projekt. Photoshop ist eine Bildbearbeitungs-Software, der mittels Plugins weitere Bildbearbeitungs-Filter hinzugefügt werden können. Eclipse ist eine Plattform zur Ausführung von Plugins. Theoretisch sind alle möglichen Anwendungen als Eclipse-Plugin denkbar. Eclipse wird z. Z. mit einem Plugin, das eine Java-IDE realisiert, sowie einem Plugin zur Entwicklung von weiteren Eclipse-Plugins geliefert. In Eclipse können Plugins aufeinander aufbauen. So besteht das Java-IDE-Plugin aus vielen weiteren Plugins für den Code-Editor, den Compiler, den Debugger u. s. w. Sowohl für das kommerzielle Photoshop als auch für das Open-Source-Projekt Eclipse existieren mittlerweile viele freie und kommerzielle Erweiterungen in Form von Plugins. Die Plugin-Fähigkeit dieser beiden Produkte ist ein wesentlicher Bestandteil ihres Erfolgs.

Ein Werkzeug, dem dynamisch Funktionalität hinzugefügt werden kann, hat also den Vorteil, dass es von Entwicklern, die nicht im Besitz des Quelltextes des Werkzeugs sind, erweitert werden kann. Aber auch das Entwickler-Team des Werkzeugs, dem der Werkzeug-Quelltext vorliegt, profitiert von einer Plugin-Architektur, da eine solche Architektur die Abhängigkeiten zwischen den Komponenten des Werkzeugs minimiert.

2.4 Universalität und Skalierbarkeit

Diese Anforderung ergibt sich aus der Zielsetzung, eine einheitliche Werkzeugarchitektur für WAM-Anwendungen zu finden. WAM-Anwendungen haben keine typische Größe. Eine allgemeine Werkzeugarchitektur sollte daher sowohl für mittel-große, als auch für große Werkzeuge sinnvoll einsetzbar sein. Das heisst, dass einerseits sehr große Werkzeuge mit Hilfe der Architektur erstellt werden können müssen. Andererseits darf der Mehraufwand, den die Architektur erfordert, um ein kleineres Werkzeug damit zu erstellen, nicht so groß sein, dass in der Praxis für kleinere Werkzeuge auf diese Architektur verzichtet wird.

Unterhalb einer gewissen Größe lohnt es sich für kurzlebige Werkzeuge wahrscheinlich niemals, eine Architektur zu verwenden, die auch und gerade für große Anwendungssysteme entworfen wurde. Sehr kleine und kurzlebige Anwendungen profitieren aber auch kaum von den in diesem Kapitel aufgestellten Anforderungen. So bringt z. B. die Trennung von Funktion und Interaktion kaum Vorteile, wenn das Programm wegen seiner Kurzlebigkeit nach der ersten Version weder geändert, noch auf eine andere Plattform portiert werden muss.

Oft werden kleine und kurzlebige Programme mit Werkzeugen zur schnellen Anwendungsentwicklung (*Rapid Development Tools, RAD Tools*) erstellt. Zu diesen Werkzeugen gehören u.a. Microsoft Visual Basic und Borland Delphi. Diese Entwicklungswerkzeuge enthalten z. B. keine explizite Unterstützung der losen Kopplung von Benutzungsoberfläche, Fachlogik und Persistenz. So enthält Delphi z. B. GUI-Elemente, die direkt mit einer Datenbank-Tabelle verknüpft werden können. Diese GUI-Elemente lesen bzw. schreiben direkt aus bzw. in die Datenbank. Aus softwaretechnischer Sicht ist dies ein sehr unsauberes Vorgehen. Für die Entwicklung bestimmter Anwendungstypen ist die Verwendung eines RAD Tools aber die praktikabelste Lösung.

Die vorliegende Arbeit beschäftigt sich mit Software, die groß und langlebig genug ist, um von den hier aufgestellten Anforderungen an eine Werkzeugarchitektur profitieren zu können. Bis zu welcher Größe eine Software als klein gilt, wie die Größe einer Software gemessen wird und ab welcher Nutzungsdauer eine Software nicht mehr als kurzlebig gilt, soll hier nicht diskutiert werden. Diese Dinge liegen meist im Ermessen der Entwickler.

2.5 Umsetzbarkeit

Eine Werkzeugarchitektur sollte für den Werkzeugentwickler keine Hürde, sondern eine Hilfe darstellen. Deshalb ist es wichtig, dass die Architektur leicht benutzbar ist. Benutzbarkeit ist gegeben, wenn die anzuwendenden Muster mit geringem Aufwand implementiert werden können, oder wenn ein Großteil des Aufwandes in einem wiederverwendbaren Rahmenwerk implementiert werden kann.

Oft können die in einem Muster teilnehmenden Klassen nicht generisch und wiederverwendbar implementiert werden, ohne statische Typisierung und damit die Möglichkeit der Fehlerentdeckung durch den Compiler aufzugeben. Als Gradmesser, ob Teile der Architektur als Rahmenwerk implementiert werden können, soll die Programmiersprache Java die-

nen, auch wenn diese Programmiersprache noch[4] keine für die Frameworkentwicklung oft vorteilhafte generische Programmierung (s. z. B. [CE00]) erlaubt. Für die Programmiersprache C++ hat Alexandrescu in [Ale01] gezeigt, dass sich viele Muster aus [GHJV96] mit Hilfe der generischen Programmierung als wiederverwendbare und typsichere Klassenbibliothek implementieren lassen. Da die vorliegende Arbeit aber zum Ziel hat, eine einheitliche Architektur für die Werkzeugentwicklung mit dem Java-Rahmenwerk JWAM zu finden, wird hier die Umsetzbarkeit einer Architektur an der Programmiersprache Java gemessen.

[4] Die z. Z. aktuelle Version von Suns Java 2 ist Version 1.4

3 Architekturmuster

Dieses Kapitel untersucht einige bekannte Architekturmuster daraufhin, welche der in Kapitel 2 aufgestellten Anforderungen an eine Werkzeugarchitektur sie erfüllen. Die Ergebnisse dieser Untersuchungen werden in Kapitel 5 dazu dienen, eine neue, einheitliche Architektur zu finden, die möglichst viele dieser Anforderungen erfüllt.

Ausser des Funktion/Interaktions-Musters (FK/IAK) ist keines der hier vorgestellten Architekturmuster in Hinblick auf die WAM-Metaphern entworfen worden. Auch galt beim Entwurf dieser Muster nicht das Leitbild des Arbeitsplatzes für eigenverantwortliche Expertentätigkeit. Die folgenden Abschnitte verwenden jeweils die Begrifflichkeiten, die üblicherweise in den untersuchten Architekturmustern verwendet werden. Gleichzeitig wird aber auch versucht, jeweils einen Bezug zu den im WAM-Kontext verwendeten Begriffen herzustellen.

3.1 Model/View/Controller (MVC)

3.1.1 Überblick

Das Model/View/Controller-Muster (MVC) wurde in den 70er Jahren bei Xerox Park entwickelt[1]. Seine erste Anwendung fand es vermutlich in Smalltalk-80. Vor [KP88] gab es praktisch keine Dokumentation zu diesem Muster. Da MVC lange vor der Prägung des Musterbegriffs in der Softwaretechnik entstanden ist, wird das Muster auch als MVC-Paradigma oder MVC-Idiom bezeichnet.

MVC ist wahrscheinlich das älteste und bekannteste Muster zur Konstruktion von interaktiven Anwendungen. Die *Gang of four*[2] erklärt in [GHJV96] anhand des MVC-Architekturmusters ihre Definition des Musterbegriffs, indem sie innerhalb von MVC verwendete Entwurfsmuster identifizieren.

Als das MVC-Muster entstand, war der Begriff des Werkzeugs noch nicht geprägt. Statt von der Konstruktion eines Werkzeugs wird bei der Beschreibung des MVC-Musters deshalb oft von der Konstruktion einer Anwendung gesprochen. Das MVC-Muster stellt allerdings keine Architektur für alle Schichten eines kompletten Softwaresystems dar. Vielmehr beschreibt es – analog zu einer Werkzeugarchitektur – nur die obersten Schichten eines solchen

[1] Der „Erfinder“ von MVC ist wahrscheinlich Krygve Reenskaug (s. [Mod02]).
[2] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides

Systems. Die oberste Schicht, die vom MVC-Muster beschrieben wird, ist die Präsentationsschicht. Die unterste beschriebene Schicht enthält die Anwendungslogik. Tieferliegende Schichten beschreibt das Muster nicht.

Das MVC-Muster trennt die Funktionalität einer Anwendung von seiner Interaktion. Dazu wird die Anwendung in die Komponenten *Model*, *View* und *Controller* unterteilt. Das Kollaborationsdiagramm in Abbildung 3.1 zeigt das Zusammenspiel der drei Komponenten.

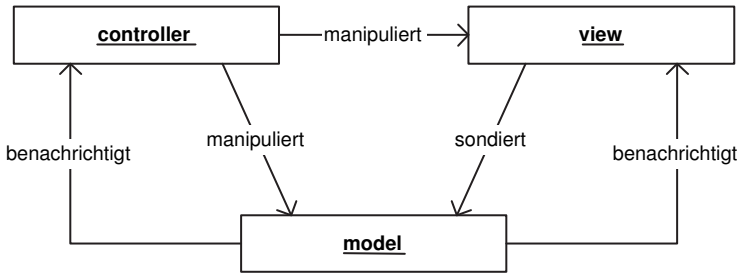

Abbildung 3.1: Das MVC-Architekturmuster

Die folgenden Abschnitte beschreiben die Aufgaben der einzelnen MVC-Komponenten.

Model

Das Model enthält die fachlichen Elemente einer Anwendung. Dazu gehören das Domain-Model sowie die Funktionalität eines Werkzeugs. Der fachliche Status einer Anwendung kann über das Model gesteuert werden. Dazu stellt das Model entsprechende sondierende und modifizierende Operationen bereit.

Zustandsänderungen teilt das Model angemeldeten Komponenten über einen Reaktionsmechanismus mit. Komponenten, die sich am Model für Benachrichtigungen über Zustandsänderungen anmelden, sind typischerweise Views und Controller, die dem Model nur unter einer abstrakten Schnittstelle bekannt sind.

Das Model ist keine reine Werkzeugkomponente, da es nicht nur die Funktionalität eines Werkzeugs, sondern die gesamte Anwendungslogik enthält.

View

Die View ist für die Repräsentation der Benutzungsschnittstelle verantwortlich. Sie stellt die Informationen des Models sowie die Interaktionsmöglichkeiten des Controllers grafisch dar.

Controller

Der Controller ist für die Interaktion mit dem Benutzer verantwortlich. Er nimmt Benutzereingaben entgegen und wandelt sie in entsprechende Methodenaufrufe am Model bzw.

an der View um. Benutzereingaben, die den fachlichen Status der Anwendung beeinflussen sollen, werden an das Model weitergeleitet. Benutzereingaben, die nur die Ansicht verändern, werden an die View weitergeleitet.

View und Controller entsprechen zusammen der Interaktion im WAM-Kontext. Sie können also als Werkzeugkomponenten bezeichnet werden.

3.1.2 Ziele

Trennung von Funktion und Interaktion

Das Hauptziel des MVC-Musters ist die in Kapitel 2 diskutierte Entkopplung von Funktion (Model) und Interaktion (View und Controller) eines Werkzeugs.

Mehrere Ansichten

Ein Ziel soll durch die Entkopplung von Funktion und Interaktion realisierbar werden: Zu einem Model kann es gleichzeitig mehr als eine Benutzungsschnittstelle geben. Dabei kann jede View andere Aspekte des Models sichtbar machen, oder die gleichen Daten auf andere Weise repräsentieren.

Abbildung 3.2 zeigt ein Model und drei Views. Die Views zeigen die Informationen des Models durch Verwendung unterschiedlicher Diagrammtypen an.

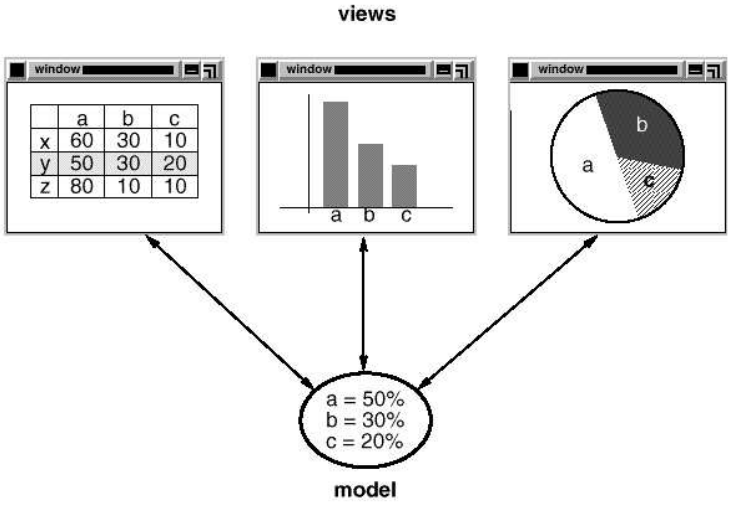

Abbildung 3.2: Ein Model mit mehreren Views (aus [GHJV96], Seite 5)

Das Kollaborationsdiagramm in Abbildung 3.3 zeigt, wie zwei View/Controller-Paare mit dem gleichen Model arbeiten.

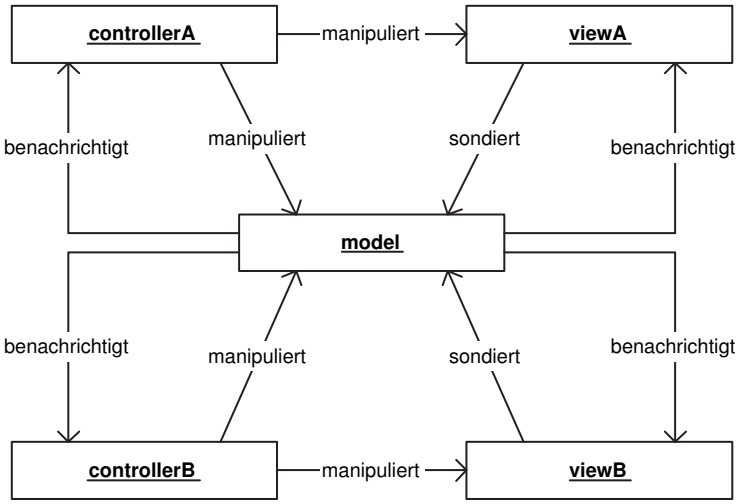

Abbildung 3.3: Zusammenspiel der MVC-Komponenten bei zwei Ansichten

Aufteilung der Interaktion in Darstellung und Kontrolle

MVC trennt ausserdem die Benutzungsschnittstelle in die Komponenten Controller und View. Dadurch wird die Interaktion (Controller) von der Darstellung (View) entkoppelt, so dass z. B. Änderungen am Layout der Benutzungsschnittstelle nur die View, aber nicht den Controller betreffen.

3.1.3 Probleme

Das Model einer Anwendung lässt sich nicht immer völlig unabhängig von der Benutzungsschnittstelle entwickeln. Oft muss antizipiert werden, auf welche Weise das Model sondiert werden wird, um dessen Informationen auf eine bestimmte Art darzustellen. Soll z. B. eine GUI entwickelt werden, die eine Baumstruktur darstellt, ist dies einfacher, wenn das Model Operationen bereitstellt, die auf diese Art der Darstellung abgestimmt sind.

Wenn die Operationen des Models schlecht zu der von der GUI verwendeten Darstellungsart passen, ist die Entwicklung der View und des Controllers schwieriger und damit fehlerträchtiger und das Programm wird wahrscheinlich eine geringere Laufzeiteffizienz aufweisen.

Wenn das Model nicht unabhängig von der Darstellung implementiert werden kann, können nicht alle in Abschnitt 3.1.2 aufgeführten Ziele des MVC-Musters erreicht werden.

Das Model kann nur in gleicher oder ähnlicher Art wieder verwendet werden. Eine solche Wiederverwendung tritt z. B. bei einem Austausch des GUI-Toolkits (z. B. bei Portierung auf eine andere Plattform) auf, da hierbei in der Regel zwar eine andere, aber trotzdem gleichartige GUI entwickelt wird.

3.1.4 Domain- und Application-Model

Ein Lösungsansatz für diese Probleme besteht darin, zwischen Funktion und Interaktion eine weitere Abstraktionsschicht einzufügen. Krasner und Pope schreiben in [KP88]:

> Using intermediary objects between views and "actual" models is a common way to further isolate the viewing behavior from the modeling application.

Diese weitere Abstraktionsschicht kann z. B. durch Aufteilung des Models in die Komponenten *Domain-Model* und *Application-Model* hergestellt werden.

Das Domain-Model ist das von Krasner und Pope zitierte „eigentliche Model". Es besteht aus den Fachobjekten der Anwendung. Fachobjekte sind die Objekte, die das fachliche Modell eines Anwendungsbereichs bilden. Zu ihnen zählen z. B. Materialien und fachliche Behälter. Fachobjekte können sondiert und manipuliert werden. Einen Reaktionsmechanismus besitzt das Domain-Model nicht.

Das Application-Model bietet der Benutzungsschnittstelle auf deren Art der Verwendung abgestimmte Operationen für den Zugriff auf die Fachobjekte an. Es benachrichtigt Views und Controller über Zustandsänderungen des Models.

Abbildung 3.4 zeigt ein Kollaborationsdiagramm mit Domain- und Application-Model. Das Domain-Model wird hier vereinfacht als einzelnes Objekt dargestellt. In der Praxis wird das Domain-Model meistens aus einer Vielzahl von Objekten bestehen.

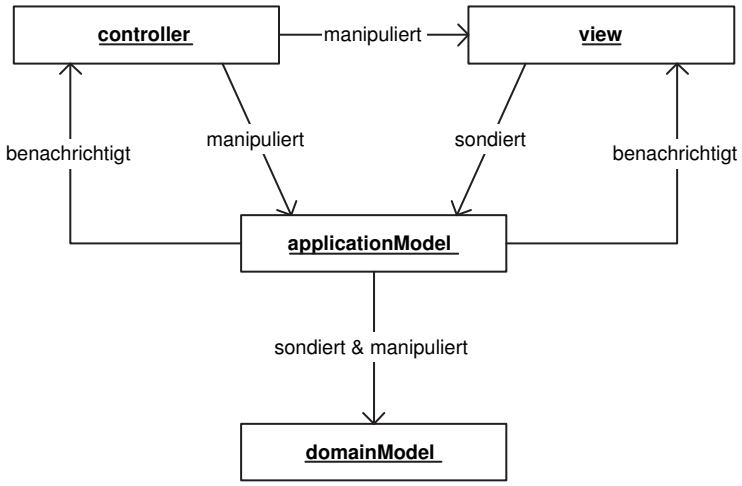

Abbildung 3.4: MVC-Muster mit verfeinertem Model

Deacon schlägt in [Dea00] „M_dM_aVC" als Akronym für eine solche Architektur vor. M_d steht hier für das Domain-Model und M_a für das Application-Model. Im Folgenden wird dieses Akronym verwendet, wenn eine MVC-Architektur mit Domain- und Application-Model gemeint ist.

Es stellt sich die Frage, wo im M_dM_aVC-Muster die Trennlinie zwischen Benutzungsschnittstelle und fachlichem Code verläuft. Im klassischen MVC-Muster bilden die Views mit ihren Controllern die Benutzungsschnittstelle. Im M_dM_aVC-Muster wird das Application-Model für eine bestimmte Art der Darstellung und für eine bestimmte Art des Umgangs mit dem Domain-Model entworfen. Wir können das Application-Model daher der Benutzungsschnittstelle zuordnen.

Theoretisch könnte man argumentieren, dass das Application-Model durch den Einsatz eines Reaktionsmechanismus nur lose mit Views und Controllern gekoppelt ist und damit eher zur Fachlogik gehört als zur Benutzungsschnittstelle. Weiterhin muss ein Application-Model keinen Plattform- oder GUI-Toolkit-spezifischen Code enthalten. In der Praxis ist es aber so, dass GUI-Toolkits wie z. B. Java-Swing zu ihren Widgets passende Application-Models enthalten. Die Swing-Klasse `JTable` arbeitet z. B. auf einem Model, das das Swing-Interface `TableModel` implementieren muss. Damit sind Application-Models abhängig vom verwendeten GUI-Toolkit und müssen bei einem Wechsel des Toolkits zusammen mit den Views und Controllern einer Anwendung ausgetauscht oder angepasst werden.

3.1.5 Bewertung

Im Folgenden wird das MVC-Muster aufgrund der in Kapitel 2 aufgestellten Anforderungen an eine Softwarearchitektur bewertet.

Trennung von Funktion und Interaktion

Das MVC-Architekturmuster unterteilt eine Anwendung in zwei technisch motivierte Schichten. Die untere Schicht enthält die Fachlogik (Domain-Model). Die obere Schicht enthält die Benutzungsschnittstelle (Application-Model, Views und Controller). Eine reale Anwendung wird meist aus mehr als diesen beiden Schichten bestehen. So wird es meistens weitere technische Schichten unterhalb der Fachlogik geben, die z. B. für Persistenz zuständig sind. Aber für das Ziel, die Funktion von der Interaktion zu entkoppeln, ist die in MVC gemachte Vereinfachung zulässig.

Die Trennung von Funktion und Interaktion ist das Hauptziel des MVC-Musters. Dieser Anforderung wird das Muster durch die Trennung der Benutzungsschnittstelle vom Rest des Softwaresystems gerecht.

In der Literatur wird das MVC-Muster meistens wie in Abschnitt 3.1.1 ohne weitere Unterteilung des Models beschrieben (vgl. z. B. [BMR+98]). Die Nachteile, die eine direkte Umsetzung dieses Architekturmusters mit sich bringt, wurden in Abschnitt 3.1.3 beschrieben. Das in Abschnitt 3.1.4 beschriebene M_dM_aVC-Muster verbessert die Entkopplung der Fachlogik einer Anwendung von seiner Benutzungsschnittstelle.

Modularisierung der Funktionalitäten

MVC bietet keine vollständige Beschreibung einer Architektur. Insbesondere bei größeren Anwendungen wird eine Schicht der MVC-Architektur nicht aus einem monolithischen Ob-

jekt bestehen. Vielmehr wird es viele Tripel aus Model-, View- und Controller-Objekten geben, die jeweils einen Teilaspekt der Anwendung repräsentieren. Die MVC-Architektur beschreibt nicht, wie diese Objekte zueinander in Beziehung stehen und wie sie miteinander kommunizieren. Auch die Granularität der MVC-Tripel wird vom MVC-Muster nicht vorgegeben. Eine Modularisierung bis auf die Ebene einzelner Funktionalitäten sollte aber möglich sein.

Dynamischer Aufbau

Eine Funktionalität eines Softwarewerkzeugs muss für den Benutzer sichtbar und ausführbar sein. Daher betrifft eine Funktionalität immer beide Schichten eines MVC-Programmes. Wenn man sich die technisch motivierten Schichten einer Anwendung als horizontale Schichten vorstellt, verlaufen Funktionalitäten vertikal durch mehrere dieser Schichten. Dadurch muss für jede neue Funktionalität jede dieser Schichten entsprechend erweitert werden. Daher eignen sich Architekturen mit technisch motivierten Schichten wie MVC nicht gut als Grundlage einer Plugin-Architektur.

Universalität und Skalierbarkeit

Um eine Aussage über die universelle Anwendbarkeit und Skalierbarkeit treffen zu können, ist die MVC-Architektur nicht detailliert genug spezifiziert. Ein nach MVC entwickeltes System wird in Model-, View- und Controller-Komponenten aufgeteilt. Über die Granularität der MVC-Komponenten-Tripel und deren Zusammenspiel wird keine Aussage gemacht.

Um skalierbar zu sein, muss ein größeres System aus mehr als einem Komponenten-Tripel bestehen können. Das MVC-Muster verbietet dies nicht. Ein wesentlicher Faktor zur Beurteilung der Skalierbarkeit ist allerdings die zu schaffende Infrastruktur für die Kommunikation der Komponenten-Tripel untereinander. Da das MVC-Muster keine Kommunikations-Infrastruktur beschreibt, hängt die Skalierbarkeit sehr stark von der konkreten Implementation des Musters ab.

Es besteht die Gefahr, dass für ein Softwaresystem anfänglich eine Kommunikations-Infrastruktur entwickelt wird, die zunächst ausreichende Funktionalität bietet, aber dann später, wenn das System wächst, nicht entsprechend ausbaufähig ist und somit neu entwickelt werden muss.

Umsetzbarkeit

Es existieren zahlreiche in der Praxis eingesetzte Frameworks und von MVC abgeleitete Architekturen für die Entwicklung von interaktiven Anwendungen. Das Java-Swing-Framework (s. Abschnitt 4.2) benutzt z. B. eine MVC-ähnliche Architektur. Auf Microsoft Windows Systemen wird oft das MFC-Framework (Microsoft Foundation Classes) eingesetzt, das die MVC-ähnliche Architektur Document/View unterstützt. Mit Hilfe der MFC sind bisher zahlreiche große, kommerziell erfolgreiche Softwaresysteme entwickelt worden. Zu diesen Systemen gehören u. a. Microsoft Word und Microsoft Excel.

Java-Swing und MFC ist gemein, dass sie auf von der View getrennte Controller-Komponenten verzichten. Die Aufteilung der Benutzungsschnittstelle in Controller und View scheint in der Praxis keinen genügend großen Nutzen zu haben, als dass sich der Aufwand, den diese Trennung mit sich bringt, lohnen würde.

Die Umsetzbarkeit des MFC-Musters hängt – genau wie die im vorherigen Abschnitt diskutierte Universalität und Skalierbarkeit – wieder von seiner konkreten Implementation ab. Die Praxis hat gezeigt, dass sich sowohl große, als auch kleine interaktive Softwaresysteme mit MFC-ähnlichen Architekturen entwickeln lassen.

Zusammenfassung

Tabelle 3.1 enthält zusammenfassend zu jeder der in Kapitel 2 aufgestellten Anforderungen stichwortartige Argumente, die für bzw. gegen die Verwendung des Musters zur Konstruktion von WAM-Werkzeugen sprechen.

Anforderung		Model/View/Controller
Trennung von Funktion und Interaktion	+	Ziel des Musters
	+	Interaktion in eigener Schicht
Modularisierung der Funktionalitäten	?	Aufbau des Models nicht spezifiziert (Aufteilung eines Werkzeugs in mehrere M/V/C-Tripel theoretisch möglich)
Dynamischer Aufbau	-	Aufteilung in technische Schichten
Universalität & Skalierbarkeit	?	nicht detailliert genug spezifiziert
	+	viele praktische Umsetzungen
Umsetzbarkeit	?	nicht detailliert genug spezifiziert
	+	viele praktische Umsetzungen

Tabelle 3.1: Zusammenfassung der Bewertung des MVC-Musters

Positive Bewertungen sind mit einem „+“ markiert. Negative Bewertungen sind entsprechend mit einem „-“ gekennzeichnet. Das MVC-Muster gibt nur sehr grob den Aufbau eines Werkzeugs vor. Anforderungen, zu denen aufgrund mangelnder Detailliertheit des Musters keine Aussage getroffen werden kann, erhalten eine mit „?“ markierte Bewertung.

3.2 Presentation/Abstraction/Control (PAC)

3.2.1 Überblick

Das Presentation/Abstraction/Control-Muster (PAC) wurde zum ersten Mal 1987 von Joëlle Coutaz in [Cou87] beschrieben. Eine aktuellere Beschreibung des Musters befindet sich in [BMR+98].

PAC unterteilt eine interakive Anwendung in drei Hauptkomponenten: Die *Presentation Techniques*, den *Dialogue Controller*, sowie den *Functional Core*. Das Paket-Diagramm in Abbildung 3.5 zeigt die Abhängigkeiten zwischen diesen Komponenten.

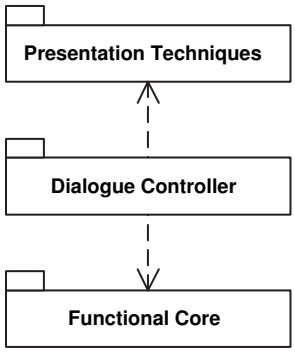

Abbildung 3.5: Die Hauptkomponenten einer PAC-Anwendung

Der Functional Core enthält das Domain Model der Anwendung. Die Presentation Techniques Komponente bildet zusammen mit dem Dialogue Controller einen PAC-Dialog. Ein PAC-Dialog entspricht technisch im Wesentlichen einem WAM-Werkzeug. Die Presentation Techniques Komponente ist für die Darstellung der Benutzungsoberfläche zuständig, während der Dialogue Controller für die Funktionalität und das Verhalten des PAC-Dialogs verantwortlich ist.

Die Aufteilung eines PAC-Dialogs in Presentation Techniques Komponente und Dialogue Controller entspricht der Aufteilung der vom JWAM Rahmenwerk (siehe Abschnitt 4.4) unterstützten Mono-Tools. Mono-Tools bestehen ebenfalls aus einer Präsentations- und einer Steuerungskomponente.

Der Dialogue Controller besteht aus sogenannten Agenten. Ein Agent ist für eine Aufgabe des Werkzeugs verantwortlich. Agenten sind in sich geschlossene Einheiten des Programms, die einen Zustand besitzen, sich ggf. in der Benutzungsschnittstelle repräsentieren, und mit dem Benutzer interagieren können. Ein Agent kann einen fachlichen oder technisch motivierten Dienst repräsentieren, ein semantisches Konzept verkörpern oder der Koordination anderer Agenten dienen.

Aufbau eines PAC-Agenten

Ein Agent besteht aus bis zu drei Komponenten: *Presentation*, *Abstraction* und *Control*. Das Kollaborationsdiagramm in Abbildung 3.6 zeigt die Komponenten eines PAC-Agenten.

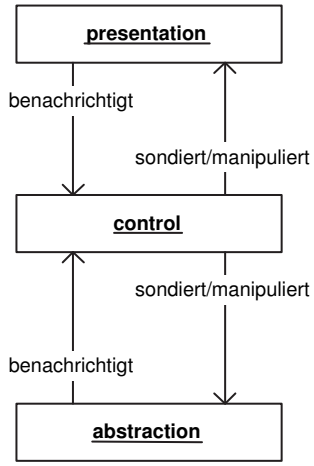

Abbildung 3.6: Aufbau eines PAC-Agenten

Die optionale Präsentationskomponente (Presentation) ist für das vom Benutzer wahrnehmbare Verhalten des Agenten verantwortlich. Sie kennt das verwendete GUI-Toolkit und hat Zugriff auf die Benutzungsschnittstelle des Programms. Die Präsentationskomponente benachrichtigt die Kontrollkomponente über Benutzeraktionen, die an die Abstraktionskomponente weitergereicht werden müssen.

Die ebenfalls optionale Abstraktionskomponente (Abstraction) implementiert die Fachlogik des Agenten und definiert dessen Zustand. Sie hat Zugriff auf den funktionalen Kern des Programms. Die Abstraktionskomponente benachrichtigt die Kontrollkomponente über Zustandsänderungen.

Die Kontrollkomponente (Control) ist in jedem Agenten vorhanden. Sie hat zwei Aufgaben: Zum einen verbindet sie die Abstraktion mit der Präsentation und zum anderen ist sie für die Kommunikation mit anderen Agenten bzw. deren Kontrollkomponenten verantwortlich. Die Kontrollkomponente ist ein Adapter zwischen der Präsentations- und der Abstraktionskomponente. Zustandsänderungen an einer dieser Komponenten werden an die jeweils andere Komponente weitergereicht. Dabei übernimmt die Kontrollkomponente ggf. auch die Anpassung von Datenformaten beim Weiterreichen von Informationen von einer zur anderen Komponente.

In einem Zeichenprogramm kann es z. B. einen Agenten geben, der die Werkzeugpalette repräsentiert. Dieser Agent besitzt eine Präsentationskomponente, die die Werkzeugpalette auf dem Bildschirm darstellt. Ein weiterer Agent repräsentiert die Zeichenfläche. Auch dieser Agent besitzt eine Präsentationskomponente. Das Zeichnen eines Kreises erfordert vom Benutzer mehrere Arbeitsschritte. Er muss das „Kreis-Werkzeug“ in der Werkzeugpalette auswählen und dann mit der Maus den Kreis in der gewünschten Größe an eine

gewünschte Position zeichnen. Diese zwei Aktionen werden gewöhnlich zu einem Befehl zusammengefasst, der dann später auch „am Stück“ rückgängig gemacht werden kann. Ein Agent, der die Aktionen „Kreis-Werkzeug auswählen“ und „Maus bei gedrückter Maustaste über Zeichenfläche bewegen“ zu dem Befehl „Kreis zeichnen“ zusammenfasst, benötigt keine eigene Präsentationskomponente.

Auch die Abstraktionskomponente muss nicht in jedem Agenten vorhanden sein. Wenn eine Abstraktionskomponente jeden Methodenaufruf an ein Objekt des funktionalen Kerns delegieren würde ohne eigene Funktionalität hinzuzufügen, kann die Kontrollkomponente stattdessen auch direkt auf dieses Objekt zugreifen. Es kann auch Agenten geben, die eine reine Präsentationsfunktionalität besitzen. Solche Agenten werden gebraucht, wenn das verwendete GUI-Toolkit die gewünschte Präsentationsform nicht anbietet. Auch solche Agenten benötigen keine Abstraktionskomponente.

Struktur aus mehreren Agenten

Die Agenten einer Anwendung werden hierarchisch strukturiert. Innerhalb dieser Baumstruktur kann es verschiedene Kategorien von Agenten geben. Die Wurzel des Baumes wird durch den *Top-Level-Agenten* gebildet. Darunter kommen *Intermediate-Level-Agenten*. *Bottom-Level-Agenten* bilden die Blätter des Baumes.

Jeder Agent ist von seinem Eltern-Knoten im Baum und damit transitiv auch von dessen Eltern-Knoten bis zum Top-Level-Agenten abhängig. Das Klassendiagramm in Abbildung 3.7 zeigt beispielhaft die Abhängigkeiten zwischen Agenten in einer PAC-Struktur.

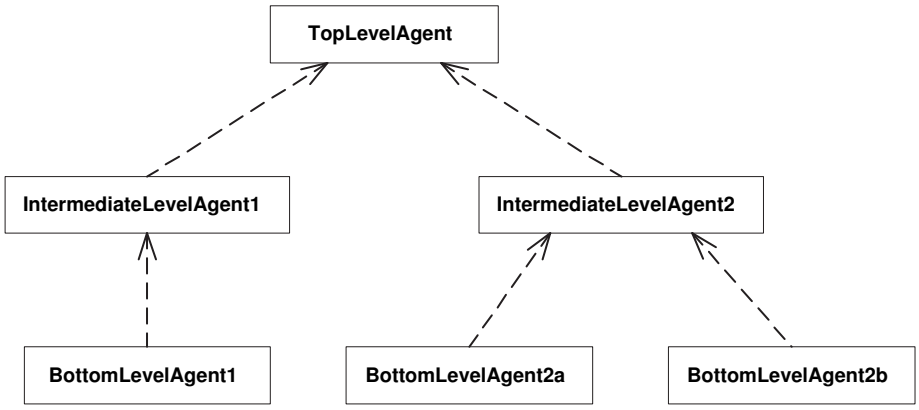

Abbildung 3.7: Abhängigkeiten zwischen PAC-Agenten

Der Top-Level-Agent verwaltet die globalen Informationen einer PAC-Anwendung. Die übrigen Agenten können diese Informationen mit Hilfe des Top-Level-Agenten sondieren und manipulieren. Der Top-Level-Agent besitzt meist keine Präsentation.

Bottom-Level-Agenten repräsentieren ein semantisches Konzept. Dies kann z. B. ein Arbeitsmaterial, ein Behälter, ein Ort, ein Diagramm, oder auch ein technischer Dienst sein. Benutzer interagieren nur mit Bottom-Level-Agenten.

Medium-Level-Agenten kombinieren feingranulare Aufgaben von Agenten auf niedrigerer Ebene zu einer abstrakteren „größeren“ Aufgabe oder regeln die Kommunikation von Agenten, die in der Baumstruktur unter ihnen stehen.

3.2.2 Ziele

Das Hauptziel des PAC-Architekturmusters ist die Trennung der in einem Softwaresystem vorkommenden semantischen Konzepte. Dieses Ziel wird dadurch erreicht, dass jedes semantische Konzept als eigener Agent mit eigener Funktion und ggf. eigener Benutzungsschnittstelle implementiert wird.

Durch die Aufteilung des Systems in Agenten soll eine Unterstützung von Änderungen und Erweiterungen erreicht werden. Die Agenten eines Softwaresystems sollen weitestgehend unabhängig voneinander entwickelt werden können, so dass Änderungen an einem Agenten keine oder nur wenige Änderungen an anderen Agenten nötig machen.

3.2.3 Kommunikation zwischen Agenten

In einer PAC-Struktur soll es nur Abhängigkeiten „von unten nach oben“ geben, d.h., dass Bottom-Level-Agenten die konkreten Typen der darüberliegenden Agenten kennen dürfen, aber nicht umgekehrt (siehe Abbildung 3.7).

Kommunikation zwischen Agenten muss allerdings in beide Richtungen, also auch von „oben nach unten“ möglich sein. Wie die Kommunikation von oben nach unten bei gleichzeitiger Unabhängigkeit eines Medium-Level-Agenten von den unter ihm liegenden Agenten erreicht werden soll, beschreibt das PAC-Muster nicht.

Im folgenden werden zwei mögliche Wege diskutiert, wie Agenten mit anderen, in der PAC-Hierarchie unter ihnen liegenden Agenten, kommunizieren können.

Generische Kommunikation

In [BMR+98] wird vorgeschlagen, dass jeder Agent ein Interface implementiert, das eine generische Methode zum Empfangen von Nachrichten enthält. Wenn jeder Agent seinen Eltern-Knoten und seine Kind-Knoten unter dieser abstrakten Schnittstelle kennt, können die Agenten untereinander kommunizieren.

Damit ein Agent weiss, was er tun soll, wenn er eine Nachricht erhält, muss die Methode zum Empfangen der Nachricht einen Parameter erhalten, der die Art der Nachricht genauer spezifiziert. Der Typ des Parameters könnte z. B. ein Nachrichtenobjekt sein, das einen Namen hat, über den die Nachricht genauer spezifiziert wird. Ausserdem könnte ein solches Nachrichtenobjekt eine Liste mit eventuellen Parametern enthalten. Abbildung 3.8 zeigt ein Interface für Nachrichtenobjekte.

Auf diese Weise könnten Agenten nicht nur mit ihren Kind-Knoten, sondern auch mit ihren Eltern-Knoten kommunizieren, so dass Agenten auch „nach oben“ keine direkten Abhän-

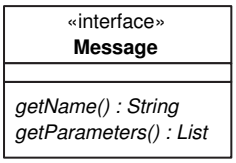

Abbildung 3.8: Interface für Nachrichtenobjekte

gigkeiten besitzen. Die Schnittstellen der Agenten könnten klein bleiben, weil sie die Methoden ihrer Eltern-Knoten nicht wiederholen müssten, um niedrigeren Knoten die Funktionalität höherer Knoten bereitzustellen. Abbildung 3.9 zeigt beispielhaft eine abstrakte Oberklasse „Agent“ für PAC-Agenten mit generischer Nachrichtenübermittlung.

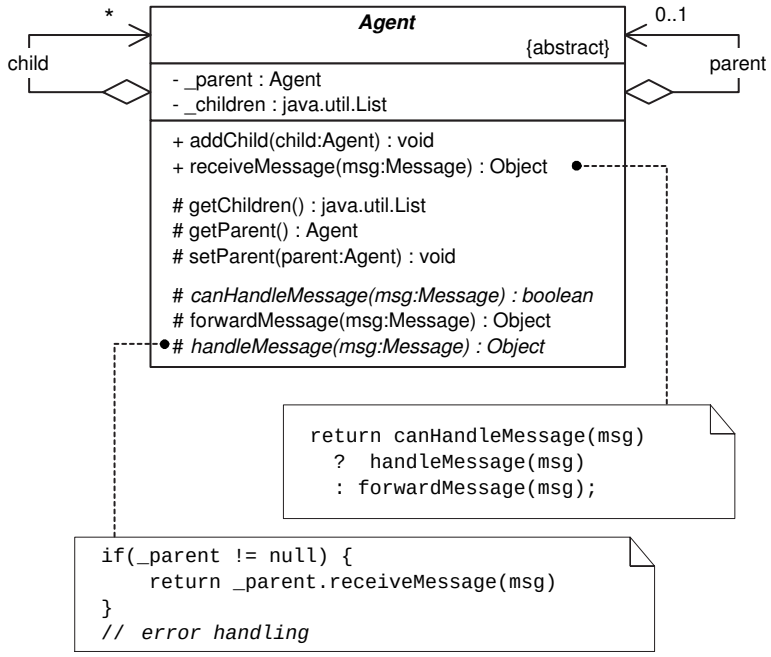

Abbildung 3.9: Oberklasse für Agenten mit generischer Nachrichtenübermittlung

Dieses Vorgehen ist allerdings mit Nachteilen verbunden. Fehler, wie falsch eingegebene Zeichenketten, können erst zur Laufzeit erkannt werden[3]. Die Interpretation der Zeichenketten kostet Zeit, geht also zu Lasten der Laufzeiteffizienz. Dies kommt besonders zum Tragen, wenn es sehr hohe Hierarchien von Agenten gibt, in denen sehr niedrige Agenten Nachrichten über viele Medium-Level-Agenten zu einem sehr hohen Agenten, wie dem Top-Level-Agenten senden. Auf ihrem Weg nach oben wird die Nachricht von jedem Medium-Level-Agenten interpretiert. Wenn ein Agent feststellt, dass er die Nachricht nicht

[3] Die Gefahr, einen solchen Syntaxfehler zu übersehen, kann allerdings mit Hilfe von Unit Tests (s. z. B. [Bec03] oder [Lin02]) minimiert werden.

verarbeiten kann, wird sie an den nächst-höheren Agenten weitergeleitet (vgl. Zuständigkeitskette (*chain of responsibility*) in [GHJV96]).

Ein weiterer Nachteil ist die Aufgabe der Typsicherheit. Um Informationen sondieren zu können, muss die generische Methode zum Empfangen von Nachrichten Werte an den Aufrufer zurückgeben können. Der Typ des Rückgabewertes muss generisch sein. In Java wäre dies z. B. `java.lang.Object`. In C++ könnte ein Zeiger auf `void`, oder besser der Typ `boost::any` (s. [BOO]) verwendet werden. Der Wert müsste dann vom Empfänger zu seinem konkreten Typ gewandelt werden (*type cast*).

Typsichere Kommunikation

Wenn jeder Agent seinen Eltern-Knoten in der Agenten-Hierarchie kennt, wird für die Kommunikation von unten nach oben kein generischer Nachrichtenmechanismus benötigt.

Für die Kommunikation in der Gegenrichtung kann das *Dependency-Inversion Principle* (DIP, siehe [Mar03]) angewendet werden: Zu jedem Agenten, der kein Bottom-Level Agent ist, wird eine Schnittstelle spezifiziert, die direkte Kind-Agenten implementieren müssen. Damit sind Kind-Agenten unter einer abstrakten Schnittstelle bekannt, unter der sie von ihren Eltern-Agenten angesprochen werden können.

Das Klassendiagramm in Abbildung 3.10 zeigt beispielhaft drei Agenten-Klassen, die jeweils Top-, Intermediate-, sowie Bottom-Level-Agenten repräsentieren. Agenten vom Typ `IntermediateAgent` und `BottomAgent` kennen ihren übergeordneten Agenten unter dessen konkreter Schnittstelle. Agenten vom Typ `TopAgent` bzw. `IntermediateAgent` kennen ihre untergeordneten Agenten über die abstrakten Schnittstellen `TopChild` bzw. `IntermediateChild`.

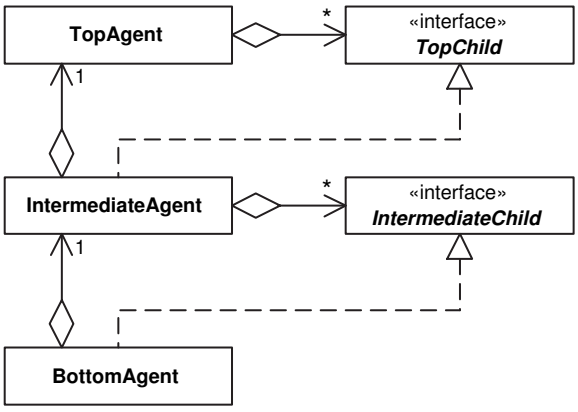

Abbildung 3.10: Anwendung des Dependency-Inversion Principle für PAC-Agenten

Der Nachteil dieser Architektur ist die u.U. große Menge an Schnittstellen, die für Top- und Intermediate-Level-Agenten erstellt werden müssen. Dieser Nachteil ist im Vergleich zur in [BMR+98] vorgeschlagenen generischen Kommunikation allerdings vernachlässigbar.

3.2.4 Bewertung

Im Folgenden wird das PAC-Muster aufgrund der in Kapitel 2 aufgestellten Anforderungen an eine Softwarearchitektur bewertet.

Trennung von Funktion und Interaktion

PAC-Agenten bestehen aus einer Kontroll-Komponente und aus einer jeweils optionalen Abstraktions- und Präsentations-Komponente. Durch die Trennung der Präsentations-Komponente von den übrigen Agenten ist eine Trennung von Funktion und Interaktion bei der Verwendung des PAC-Musters gegeben.

Modularisierung der Funktionalitäten

Jeder PAC-Agent repräsentiert ein semantisches Konzept des Softwaresystems. Die Abstraktionskomponente eines Agenten kann mehrere Funktionalitäten beinhalten. Die Unterteilung in semantische Konzepte ist also eine gröbere als die Unterteilung in einzelne Funktionalitäten. In einem PAC-System sind die einzelnen Funktionalitäten nicht in einem monolithischen Objekt zusammengefasst, sondern entsprechend den semantischen Konzepten des Systems gruppiert.

Dynamischer Aufbau

Das PAC-Muster wurde nicht speziell dafür entworfen, ein Werkzeug dynamisch zur Laufzeit mit Funktionalität anzureichern und stellt somit auch keine spezielle Unterstützung für diese Anforderung bereit.

PAC-Systeme sollen durch die Aufteilung in einzelne Agenten relativ einfach zu erweitern sein. Eine Erweiterung geschieht durch Hinzufügen einer Funktionalität zu einem bestehenden Agenten oder durch Hinzufügen eines weiteren Agenten zum System.

Ein dynamischer Aufbau der Funktionalitäten eines Agenten wird von der PAC-Architektur nicht explizit unterstützt. Das dynamische Hinzufügen weiterer Agenten zur Laufzeit ist aber technisch möglich, wenn die einzelnen Agenten entsprechend voneinander entkoppelt sind.

Universalität und Skalierbarkeit

Für sehr kleine Systeme bedeutet der Einsatz von PAC durch die nötige Infrastruktur für die Kommunikation der Agenten untereinander einen relativ großen Mehraufwand. Mit wachsender Größe des Systems amortisiert sich dieser Mehraufwand jedoch schnell durch die gewonnene Entkopplung der Agenten.

Ein für die Beurteilung der Skalierbarkeit der PAC-Architektur wichtiger Aspekt wird im PAC-Muster nicht beschrieben. Die Agenten eines Softwaresystems müssen miteinander interagieren. Die dafür nötige Infrastruktur wird in PAC nicht definiert. Wie bei der MVC-Architektur hängt die Skalierbarkeit stark von der verwendeten Kommunikations-Infrastruktur ab. Es besteht die Gefahr, dass eine für ein kleines Softwaresystem entwickelte Infrastruktur nicht mehr zum System passt, wenn dieses im Code- und Funktionsumfang wächst.

Umsetzbarkeit

Auch die Umsetzbarkeit der PAC-Architektur hängt nicht zuletzt von der verwendeten Infrastruktur ab. Es wurden bereits einige Projekte auf Basis der PAC-Architektur realisiert. Dazu zählen u. a. ein von Stadtherr und Traving in [ST93] beschriebenes System zur Überwachung von Telekommunikationsnetzen und ein von Crowley in [Cro85] beschriebenes System zur Steuerung von mobilen Robotern.

Zusammenfassung

Tabelle 3.2 zeigt eine Zusammenfassung der Bewertung des Presentation/Abstraction/-Control-Musters.

Anforderung		Presentation/Abstraction/Control
Trennung von Funktion und Interaktion	+	Eigene Präsentationskomponente pro Agent
Modularisierung der Funktionalitäten	+	Modularisierung der semantischen Konzepte
	-	Semantisches Konzept oft gröber als eine Funktionalität
Dynamischer Aufbau	?	nicht explizit vorgesehen
	+	Modularisierung der semantischen Konzepte
Universalität & Skalierbarkeit	?	wenige praktische Umsetzungen
Umsetzbarkeit	?	wenige praktische Umsetzungen

Tabelle 3.2: Zusammenfassung der Bewertung des PAC-Musters

3.3 Funktion/Interaktion (FK/IAK)

3.3.1 Überblick

Das Funktion/Interaktion-Muster unterteilt ein Werkzeug in Funktions- und Interaktionskomponenten (FK und IAK). Das Kollaborationsdiagramm in Abbildung 3.11 zeigt das Zusammenspiel der Komponenten.

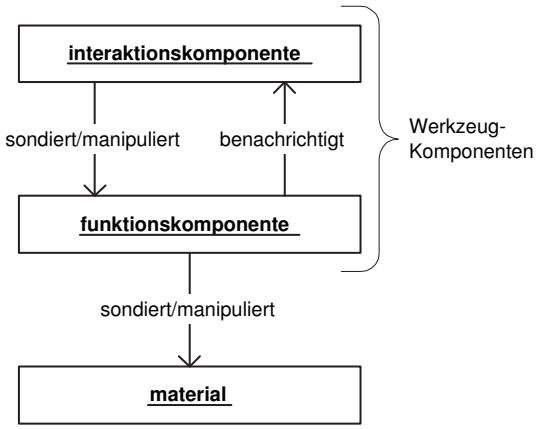

Abbildung 3.11: Das Funktion/Interaktion-Architekturmuster

Definition: Funktionskomponente (nach [Zül98], S. 240)

> Die Funktionskomponente realisiert die fachliche Funktionalität eines Softwarewerkzeugs. Sie bearbeitet das Material über die Schnittstelle der entsprechenden Aspekte. Eine Funktionskomponente bietet mittels sondierender Operationen Informationen über den eigenen Bearbeitungszustand und den Bearbeitungszustand des Materials an. Sie verwaltet den eigenen Bearbeitungszustand, das Werkzeuggedächtnis, das von Aktivitäten des Benutzers und dem Materialzustand abhängig ist.

Definition: Interaktionskomponente (nach [Zül98], S. 238)

> Die Interaktionskomponente realisiert Handhabung und Präsentation eines Softwarewerkzeugs. Sie abstrahiert mit Hilfe von Ein- und Ausgabekomponenten, den sogenannten *Interaktionstypen* (z. B. Knöpfe, Menüs), vom zugrundeliegenden Fenstersystem und ruft diese Interaktionstypen zur Darstellung und Benutzereingabe auf. Eine Interaktionskomponente nimmt einen Strom von Systemereignissen, die durch Aktionen des Benutzers ausgelöst werden, als Programmereignisse entgegen und interpretiert sie. Dabei setzt sie präsentationsbezogene Ereignisse (z. B. neue Farbeinstellungen) selbst um und leitet anwendungsbezogene Ereignisse als Aufrufe an ihre FK weiter.

Funktions- und Interaktionskomponenten sind lose miteinander gekoppelt. Die IAK kennt die Schnittstelle der FK, während die FK die IAK nicht kennt.

Die IAK sondiert den Zustand des Werkzeugs und der bearbeiteten Materialien an der FK und setzt diese Informationen in eine ensprechende Präsentation um. Anwendungsbezogene Benutzeraktionen setzt die IAK in modifizierende Methodenaufrufe an der FK um. Die FK führt die gewünschte Modifikation am Material aus und benachrichtigt dann über einen Reaktionsmechanismus die IAK über die ausgeführte Änderung. Die IAK reagiert darauf mit einer Aktualisierung der Präsentation.

3.3.2 Trennung von Handhabung und Präsentation

Die Interaktion eines Werkzeugs kann weiter aufgeteilt werden in Handhabung und Präsentation. Während die Präsentation vom verwendeten Toolkit abhängig ist, kann die Handhabung durch den Einsatz von Interaktionstypen Toolkit-unabhängig entwickelt werden.

Definition: Interaktionstyp (IAT) (nach [Zül98], S. 266)

> Ein Interaktionstyp ist eine abstrakte Form der Handhabung eines Werkzeugs. Er kapselt die konkrete Präsentation und den Interaktionsmechanismus, den die grafischen Bausteine eines User Interface Toolkit anbieten. Dabei wandelt ein IAT Systemereignisse in Programmereignisse um. Ein IAT präsentiert und liefert nur Fachwerte. Er hat also keine globalen Effekte. Ein IAT wird von einer Interaktionskomponente verwendet und macht sie damit unabhängig vom gewählten Toolkit.

Statt mit konkreten Widgets arbeitet die IAK mit abstrakten Interaktionstypen. So können z. B. alle Widgets, die eine Menge von Elementen zur Selektion anzeigen, wie z. B. Listen oder Kombinationsfelder, durch den IAT `Selection` repräsentiert werden. Aus Sicht der Handhabung ist es egal, welche konkrete Präsentation hinter einer `Selection` steckt. Es ist nur wichtig, dass der Benutzer ein von diesem Widget präsentiertes Element auswählen kann und dass auf dieses Ereignis reagiert werden kann.

3.3.3 Werkzeugkomposition

Komplexe Werkzeuge können aus Teilwerkzeugen (Sub-Werkzeug) zusammengesetzt werden. Dabei besitzt jedes Sub-Werkzeug seine eigene Funktions- und Interaktionskomponente und kann seinerseits aus weiteren Sub-Werkzeugen bestehen. Ein aus Sub-Werkzeugen zusammengesetztes Werkzeug wird als Kombi-Werkzeug bezeichnet.

Definitionen: Einfaches Werkzeug, Kombi-Werkzeug, Sub-Werkzeug, Kontext-Werkzeug (nach [Zül98], S.279)

> Ein *einfaches Werkzeug* besitzt anwendungsfachlich eine elementare Funktionalität. Softwaretechnisch besitzt es keine Sub-Werkzeuge. Ein *Kombi-Werkzeug*

kombiniert verschiedene fachliche Dienstleistungen zur Erledigung einer komplexen Aufgabe. Softwaretechnisch ist es aus existierenden Sub-Werkzeugen zusammengesetzt. Ist ein Werkzeug als technische Komponente innerhalb eines anderen Werkzeugs eingesetzt, so wird es als *Sub-Werkzeug* bezeichnet. Ein *Kontext-Werkzeug* bildet den technischen und konzeptionellen Abschluß für Sub-Werkzeuge. Es realisiert damit ein fachliches Kombi-Werkzeug. Kombi-Werkzeug und einfaches Werkzeug sind anwendungsfachliche Begriffe des Benutzungsmodells. Sub- und Kontext-Werkzeug sind Begriffe, die bei der technischen Konstruktion von Werkzeugen verwendet werden.

Jedes Sub-Werkzeug ist für eine Teilaufgabe des Werkzeugs zuständig. Das Kontext-Werkzeug steuert seine Sub-Werkzeuge, die keine Abhängigkeiten untereinander besitzen, und erfüllt so die Gesamtaufgabe des Werkzeugs.

3.3.4 Ziele

Das Hauptziel des Funktion/Interaktion-Musters ist wie beim MVC-Muster die Trennung von Interaktion und Funktion. Durch diese Trennung sollen die Funktion und die Interaktion unabhängig voneinander entwickelt werden können.

Ein anderes Ziel ist die Modularisierung durch Sub-Werkzeuge. Komplexe Werkzeuge sollen aus kleinen, wiederverwendbaren Sub-Werkzeugen zusammengebaut werden können. Damit einher geht das Ziel der Wiederverwendbarkeit von Sub-Werkzeugen.

3.3.5 Probleme

Interaktionstypen

Für jedes Toolkit, für das Werkzeuge mit Trennung der Handhabung und Präsentation erstellt werden sollen, muss es konkrete Präsentationen für alle verwendeten Interaktionstypen geben. Die Entwicklung entsprechender Präsentationen kann je nach Toolkit unterschiedlich aufwändig sein.

Im Swing-Toolkit ist es beispielsweise erlaubt, eigene Klassen von den im Toolkit enthaltenen Widget-Klassen abzuleiten. Dies ermöglicht es, die Präsentationsklassen, die die IAT-Interfaces implementieren, von den Widget-Klassen erben zu lassen. Das Klassendiagramm in Abbildung 3.12 ist ein Beispiel für die Implementation einer Präsentation für den `Selection`-Interaktionstyp.

Andere Toolkits, wie z. B. das Standard Widget Toolkit (SWT), erlauben kein Ableiten von konkreten Widget-Klassen. Hier müssen aufwändigere Methoden, wie z. B. der Einsatz des Brücken-Musters (*Bridge*, siehe [GHJV96]), angewandt werden. Abbildung 3.13 zeigt einen möglichen Entwurf des `Selection`-Interaktionstyps für das SWT-Toolkit.

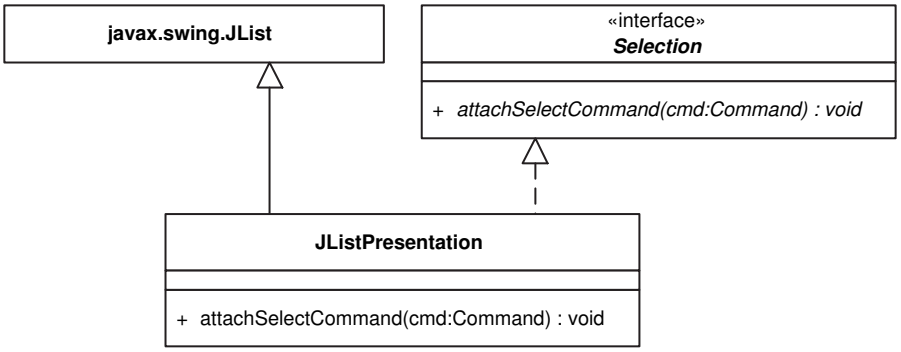

Abbildung 3.12: Ein Beispiel-IAT für Swing

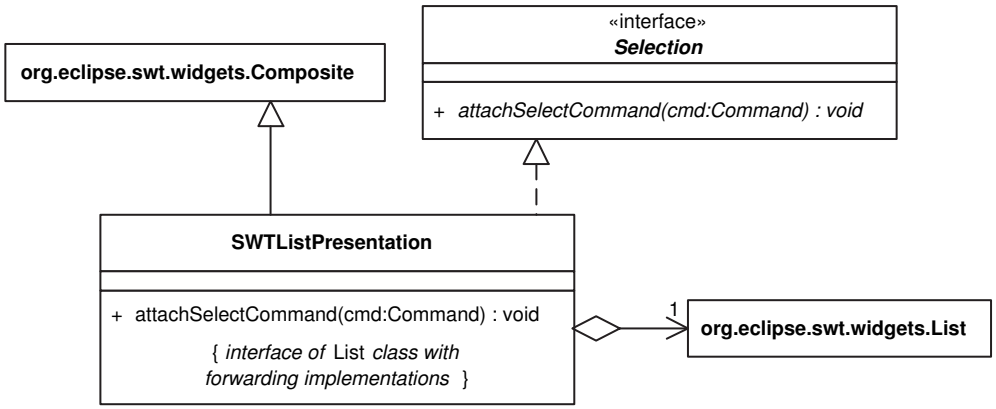

Abbildung 3.13: Ein Beispiel-IAT für SWT

Die Präsentation eines Werkzeugs wird oft mit Hilfe von sogenannten GUI-Buildern erstellt. GUI-Builder sind Werkzeuge, die dem Entwickler erlauben, eine Präsentation ohne Programmieraufwand grafisch zusammenzustellen. Nicht jeder GUI-Builder erlaubt die Verwendung eigener Widgets (z. B. `JListPresentation` statt `JList` aus Abbildung 3.12). In diesem Fall müssen die Klassen des verwendeten Toolkits direkt verwendet werden. Die Widgets müssen gegenüber der Interaktionskomponente dann mittels Proxy-Objekten (siehe Abbildung 3.14) repräsentiert werden.

Das Erstellen eines kompletten Satzes von Präsentationen für ein Toolkit ist ein erheblicher Aufwand. Viele moderne GUI-Toolkits enthalten komplexe GUI-Elemente wie Tabellen und Baumstrukturen. Für solche Elemente allgemeine Interaktionstypen zu erstellen und für die jeweiligen Toolkits ensprechende Präsentationen zu implementieren, ist keine triviale Aufgabe. Der Einsatz von Interaktionstypen lohnt sich daher meist nur dann, wenn bereits eine Bibliothek mit Interaktionstypen für die jeweiligen Zielplattformen existiert.

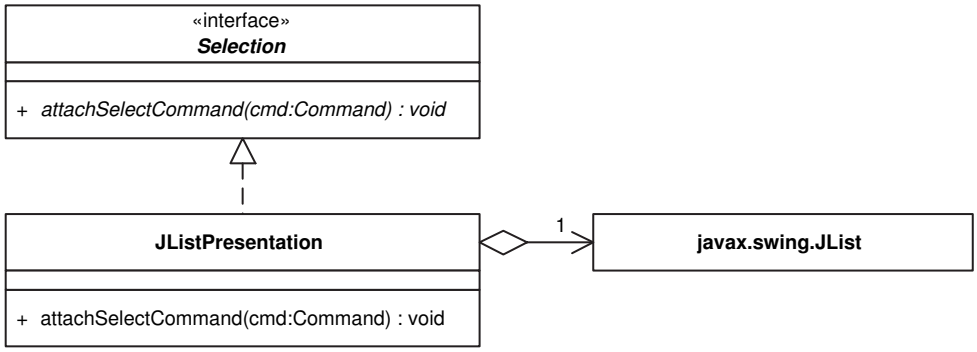

Abbildung 3.14: Proxy-Lösung eines IATs

Werkzeugkomposition

Der in Abschnitt 1.3.2 besprochene Werkzeugtyp zum grafischen Bearbeiten von Materialien läßt sich nicht gut in Sub-Werkzeuge unterteilen.

Dokumentbasierte Werkzeuge zeigen meist eine Arbeitsfläche an, auf der der Benutzer Materialien frei verschieben und manipulieren kann. Die einzelnen Funktionalitäten haben in diesem Werkzeugtyp keine abgegrenzte eigene Darstellung. Die Funktionalitäten werden vielmehr durch den Benutzer aktiviert, indem er explizit eine Funktionalität aus einem Menü oder einer Werkzeugleiste auswählt, oder die grafischen Elemente anklickt, verschiebt, miteinander verbindet, etc.

Sub-Werkzeuge teilen sich keine gemeinsame Benutzungsoberfläche. Jedes Sub-Werkzeug hat seine eigene Präsentation. Die einzelnen Präsentationen werden vom Kontextwerkzeug zu einer Präsentation zusammengesetzt. Sub-Werkzeuge sind also für dialogbasierte Werkzeuge (siehe Abschnitt 1.3.1) konzipiert.

Werkzeuge neigen dazu, im Laufe der Zeit aufgrund von neuen Anforderungen zu wachsen. Damit besteht manchmal die Notwendigkeit, ein Werkzeug ab einer bestimmten Menge von Funktionalitäten zu refaktorieren: Aus dem einfachen Werkzeug wird ein Kombi-Werkzeug. In den meisten Fällen wird es einfacher sein, einem bestehenden Werkzeug einfach eine weitere Funktionalität hinzuzufügen, als es vorher zu refaktorieren. Daher bedarf es auf Seiten des Entwicklers einer gewissen Disziplin, wenn komplexe Werkzeuge konsequent aus Sub-Werkzeugen zusammengesetzt werden sollen.

3.3.6 Bewertung

Im Folgenden wird das FK/IAK-Muster aufgrund der in Kapitel 2 aufgestellten Anforderungen an eine Softwarearchitektur bewertet.

Trennung von Funktion und Interaktion

Die Trennung von Funktion und Interaktion ist das Hauptziel dieses Musters. Dieses Ziel wird durch die Trennung des Werkzeugs in Funktions- und Interaktionskomponenten in ähnlicher Weise erreicht, wie beim MVC-Muster durch eine Trennung in Model und View/Controller.

Durch die Trennung der Interaktion und Handhabung kann die Präsentation gegenüber dem MVC-Muster noch besser isoliert werden. Dies verbessert die Anpassbarkeit der Darstellung eines Werkzeugs.

Modularisierung der Funktionalitäten

Das FK/IAK-Muster sieht die Aufteilung eines komplexen Werkzeugs in Sub-Werkzeuge vor. Dies erlaubt eine Modularisierung sowohl der Funktionalitäten als auch der Präsentationen. Dies gilt allerdings nur für dialogbasierte Werkzeuge. Dokumentbasierte Werkzeuge lassen sich meist nicht in Sub-Werkzeuge aufteilen.

Dynamischer Aufbau

Ein dynamischer Aufbau eines Werkzeuges aus Sub-Werkzeugen ist im FK/IAK-Muster nicht explizit vorgesehen und wird z.Z. auch von keinem Rahmenwerk unterstützt.

Universalität, Skalierbarkeit und Umsetzbarkeit

Der Aufbau einer FK/IAK-Anwendung ist – besonders beim Einsatz von Präsentationsformen und Sub-Werkzeugen – sehr komplex. Diese Komplexität kann zum großen Teil in einem Rahmenwerk „versteckt“ werden. Ein solches Rahmenwerk für die Entwicklung von WAM-Anwendungen mit FK/IAK-Unterstützung ist das Java-Rahmenwerk JWAM (siehe [JWA]).

Die JWAM-Entwickler haben erkannt, dass es sich trotz Unterstützung durch ein Rahmenwerk, insbesondere für sehr kleine Werkzeuge, nicht immer lohnt, das FK/IAK-Muster anzuwenden. Deshalb sind in JWAM auch Werkzeuge vorgesehen, die ohne diese Trennung von Funktion und Interaktion auskommen.

Hieraus ergibt sich ein Widerspruch: Komplexe Werkzeuge sollen möglichst aus kleineren, einfachen Sub-Werkzeugen zusammengesetzt werden. Für kleine Werkzeuge lohnt sich meist die Aufteilung in FK und IAK nicht. Daraus folgt, dass auch grosse Werkzeuge oft keine FK/IAK-Trennung besitzen.

Die Trennung von Handhabung und Präsentation kann sehr schwierig umzusetzen sein. Der Einsatz von Interaktionstypen lohnt sich deshalb meist nur dann, wenn bereits ein Rahmenwerk mit Interaktionstypen und den dazugehörigen Präsentationen für die gängigsten

Toolkits existiert. Das JWAM Rahmenwerk enthält z. Z. nur Präsentationen für das Swing-Toolkit. Dies schränkt den Nutzen der Trennung von Handhabung und Präsentation sehr ein.

Auch die Erstellung von Kombi-Werkzeugen aus Sub-Werkzeugen ist ohne unterstützendes Rahmenwerk eine sehr komplexe Aufgabe. Dass diese Aufgabe gut von einem Rahmenwerk unterstützt werden kann, zeigt das JWAM-Rahmenwerk, das die Erstellung von Kombi-Werkzeugen ermöglicht.

Dokumentbasierte Werkzeuge profitieren kaum von Sub-Werkzeugen, da sie hauptsächlich aus einer Zeichenfläche bestehen, die sich nicht gut in Sub-Werkzeuge unterteilen lässt.

Zusammenfassung

Tabelle 3.3 zeigt eine Zusammenfassung der Bewertung des Funktion/Interaktion-Musters.

Anforderung		Funktion/Interaktion
Trennung von Funktion und Interaktion	+	Ziel des Musters
	+	Funktion in eigener Komponente (FK)
	+	Interaktion in eigener Komponente (IAK)
Modularisierung der Funktionalitäten	+	Aufteilung in Subwerkzeuge
	-	Subwerkzeuge nur für dialogbasierte Werkzeuge sinnvoll
	-	Subwerkzeuge haben oft mehr als eine Funktionalität
Dynamischer Aufbau	?	nicht explizit vorgesehen
Universalität & Skalierbarkeit	-	Subwerkzeuge nur für dialogbasierte Werkzeuge sinnvoll
Umsetzbarkeit	-	hohe Komplexität
	+	Komplexität kann in Rahmenwerk „versteckt“ werden

Tabelle 3.3: Zusammenfassung der Bewertung des FK/IAK-Musters

3.4 Vergleich der Architekturmuster

Keines der in diesem Kapitel untersuchten Architekturmuster erfüllt alle in Kapitel 2 aufgestellten Anforderungen.

Das MVC-Muster zielt fast ausschliesslich auf die Trennung von Funktion und Interaktion. Dynamisches Hinzufügen von Funktionalität wird durch die Schichtenarchitektur erschwert.

Das PAC-Muster unterteilt ein Werkzeug in Agenten, die semantische Konzepte repräsentieren. Die Trennung von Funktion und Interaktion geschieht hier innerhalb jedes Agenten. Ein semantisches Konzept kann zwar mehrere Funktionalitäten umfassen, so dass ein Agent eine gröbere Einheit darstellt als die geforderte Modularisierung der Funktionalitäten erfordert. Aber gegenüber dem MVC-Muster lassen sich die in dieser Arbeit aufgestellten Anforderungen an eine Werkzeugarchitektur mit einem PAC-ähnlichen Muster leichter umsetzen.

Das FK/IAK-Muster mit Subwerkzeugen, Präsentations- und Interaktionsformen eignet sich vor allem für dialogbasierte Werkzeuge. Die Modularisierung eines Werkzeugs mit Subwerkzeugen ist gröber als die geforderte Modularisierung der einzelnen Funktionalitäten. Ausserdem scheint die gleichzeitige Verwendung von Funktions- und Interaktionskomponenten sowie Subwerkzeugen nicht praktikabel zu sein (siehe Abschnitt 3.3.6).

Tabelle 3.4 zeigt eine Zusammenfassung der Tabellen 3.1, 3.2 und 3.3.

Anforderung	MVC	PAC	FK/IAK
Trennung von Funktion und Interaktion	+ Ziel des Musters + Interaktion in eigener Schicht	+ Eigene Präsentations-komponente pro Agent	+ Ziel des Musters + Funktion in eigener Komponente (FK) + Interaktion in eigener Komponente (IAK)
Modularisierung der Funktionalitäten	? Aufbau des Models nicht spezifiziert (Aufteilung eines Werkzeugs in mehrere M/V/C-Tripel theoretisch möglich)	+ Modularisierung der semantischen Konzepte - Semantisches Konzept oft gröber als eine Funktionalität	+ Aufteilung in Subwerkzeuge - Subwerkzeuge nur für dialogbasierte Werkzeuge sinnvoll - Subwerkzeuge haben oft mehr als eine Funktionalität
Dynamischer Aufbau	- Aufteilung in technische Schichten	? nicht explizit vorgesehen + Modularisierung der semantischen Konzepte	? nicht explizit vorgesehen
Universalität und Skalierbarkeit	? nicht detailliert genug spezifiziert + viele praktische Umsetzungen	? wenige praktische Umsetzungen	- Subwerkzeuge nur für dialogbasierte Werkzeuge sinnvoll
Umsetzbarkeit	? nicht detailliert genug spezifiziert + viele praktische Umsetzungen	? wenige praktische Umsetzungen	- hohe Komplexität + Komplexität kann in Rahmenwerk „versteckt“ werden

Tabelle 3.4: Gegenüberstellung der Bewertungen der Architekturmuster

4 Java GUI-Toolkits und Application Frameworks

Die folgenden Abschnitte beschreiben die Java GUI-Toolkits AWT, Swing und SWT/JFace. Diese Toolkits werden für die in Kapitel 6 erstellten Beispielanwendungen verwendet und/oder enthalten Konzepte und Muster, die in das in den Kapiteln 5 und 6 erarbeitete Manipulator-Muster eingehen.

Darauf folgt ein Abschnitt über das Java Rahmenwerk JWAM. Dieses Rahmenwerk unterstützt die Werkzeugkonstruktion nach dem Werkzeug & Material-Ansatz (WAM).

4.1 Abstract Windowing Toolkit (AWT)

4.1.1 Überblick

Das Abstract Windowing Toolkit (AWT) ist das erste von zwei Toolkits, die mittlerweile zur Java Standardbibliothek gehören. Das AWT enthält Klassen zur Erzeugung von Fenstern, Dialogen und der üblichen Grafik-Elemente wie Schaltfächen, Eingabefeldern, Auswahlmenüs, etc.

Diese Widget-Klassen benutzen sogenannte Peer-Interfaces, die die eigentliche Funktionalität der Widgets bereitstellen. Für jede Plattform, für die es eine Java-Implementation gibt, gibt es eigene Implementationen dieser Peer-Interfaces. Peer-Objekte repräsentieren jeweils das native Widget der jeweiligen Plattform. Dadurch hängt das genaue Aussehen der AWT-Widgets von der Plattform ab, auf der sie dargestellt werden.

Widget-Klassen erzeugen ihre Peers mittels des Entwurfsmusters „Abstrakte Fabrik“ (siehe [GHJV96]).

Abbildung 4.1 zeigt die AWT-Widget-Klasse `Button` mit seinem Peer-Interface `ButtonPeer`. AWT-Buttons benutzen die Klasse `Toolkit` als abstrakte Fabrik für Implementationen des Peer-Interfaces. In der Linux-Implementation der AWT-Bibliothek werden die Klassen `MToolkit` und `MButtonPeer` als konkrete Implementationen von `Toolkit` bzw. `ButtonPeer` verwendet.

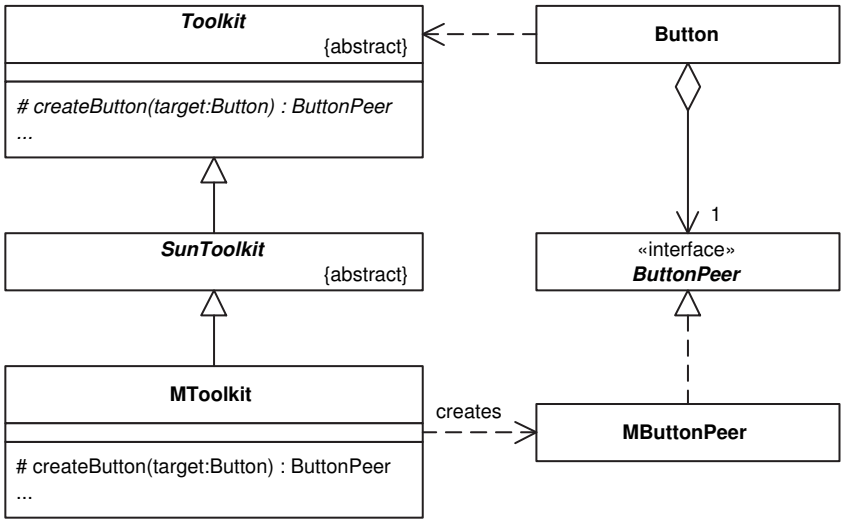

Abbildung 4.1: AWT-Widget `Button` mit Peer-Interface und abstrakter Fabrik

4.1.2 Das Listener-Muster

Anwendungen mit grafischer Benutzungsoberfläche sind meist ereignisgesteuert. Für diesen Zweck bieten praktisch alle GUI-Toolkits eine Implementation eines Reaktionsmusters an, mit deren Hilfe Widgets beobachtet werden können.

Seit Java Version 1.1 verwendet das AWT das Java Listener-Muster als Reaktionsmuster. Das Listener-Muster entspricht im Wesentlichen dem in [GHJV96] beschriebenen Beobachter-Muster. Die Beobachter heissen hier *Listener* und können für jedes Ereignis, das an einem Widget auftreten kann, eine entsprechende auf das jeweilige Ereignis reagierende Methode besitzen.

Das Listener-Muster verwendet das sogenannte Push-Modell. D.h., dass die auf ein Ereignis reagierenden Methoden einen Parameter besitzen, der das Ereignis genauer beschreibt. Diese Technik steht im Gegensatz zum Pull-Modell, bei dem der Beobachter das Subjekt sondiert, um herauszufinden, wie sich der Zustand des Subjekts geändert hat.

Zu jedem Listener-Interface, das mehr als eine Methode besitzt, gibt es im AWT eine dieses Interface implementierende Adapter-Klasse. Die Implementationen der Adapter-Klasse sind leer. Beobachter, die sich nicht für jedes Ereignis, für das das Listener-Interface eine Methode bereitstellt, interessieren, können die entsprechende Adapter-Klasse erweitern und nur die relevanten Methoden redefinieren.

Das Listener-Muster wird auch in Java Swing (siehe Abschnitt 4.2) und dem Standard Widget Toolkit (SWT, siehe Abschnitt 4.3) verwendet.

4.1.3 Bewertung

Das AWT wurde inzwischen fast vollständig vom Java Swing Toolkit abgelöst. Es gab eine Übergangsphase, in der insbesondere für Web-Applikationen weiterhin das AWT statt Swing eingesetzt wurde, weil Swing nicht für alle Web-Browser zur Verfügung stand. Dieser Umstand hat sich inzwischen geändert, so dass nach den lokal installierten Desktop-Anwendungen auch Web-Applikationen vermehrt mit Swing entwickelt werden.

Das in AWT eingesetzte Listener-Muster hat sich inzwischen als das Standard-Reaktionsmuster für Java etabliert und wird inzwischen unter anderem auch in anderen GUI-Toolkits sowie in XML-Parsern eingesetzt.

4.2 Java Swing

4.2.1 Überblick

Java Swing ist das zweite der beiden Java Standard-GUI-Toolkits. Wie das Abstract Windowing Toolkit (AWT) ist auch Swing in der Laufzeitumgebung (*Java Runtime Environment*, JRE) und dem freien Entwicklungspaket (*Java Development Kit*, JDK) der Firma Sun in den Versionen ab 1.2 enthalten.

Swing ist vollständig in Java implementiert und damit plattformunabhängig. Statt die jeweiligen nativen Widgets der zugrundeliegenden Plattform zu benutzen, zeichnet Swing alle GUI-Elemente vollständig selbst. Dadurch ist das Aussehen einer Swing-Anwendung unabhängig von der Plattform, auf der das Programm läuft. Um einer Swing-Anwendung trotzdem ein plattformspezifisches Aussehen geben zu können, enthält Swing die Möglichkeit, das sogenannte *Look and Feel* zu beeinflussen. Abbildung 4.2 zeigt beispielhaft einen Dialog aus Borlands Java-IDE JBuilder mit den Look and Feels der Plattformen Microsoft Windows und Apple Macintosh.

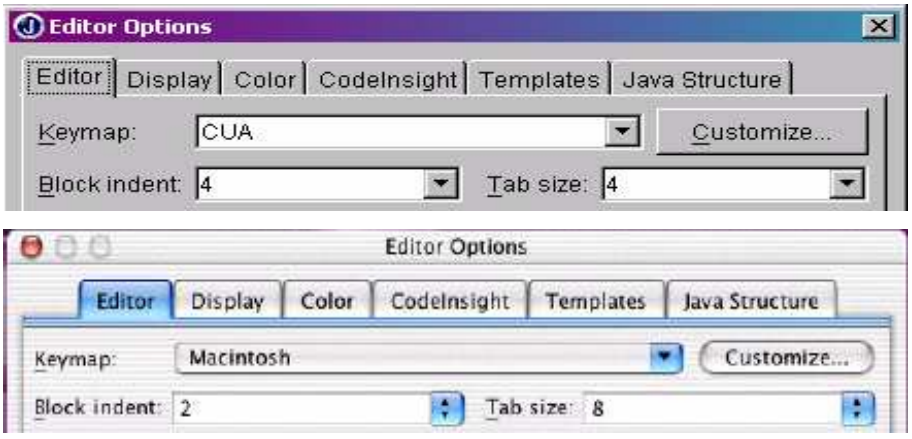

Abbildung 4.2: Beispiel für unterschiedliche Look-And-Feels

4.2.2 Model/Delegate-Architektur

Swing benutzt eine vereinfachte MVC-Architektur namens *Model/Delegate*. Jedes Swing-Widget (Typ `JComponent`) besteht aus einem Model und einem *UI Delegate*. Das Model entspricht dem Model im MVC-Muster. Der UI Delegate vereint die View und den Controller in einer Komponente (s. Abb 4.3) und implementiert das Look-And-Feel des Widgets.

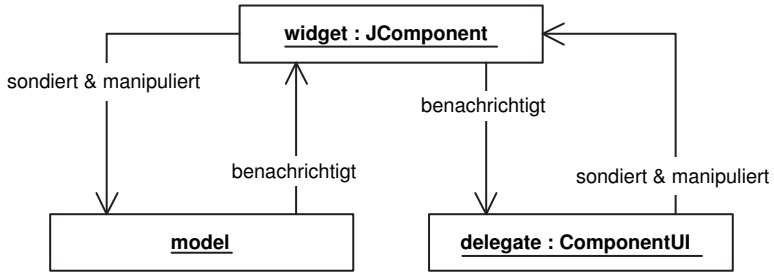

Abbildung 4.3: Model/Delegate-Architektur in Java Swing

Swing verwendet das schon in AWT eingeführte Listener-Konzept als Reaktionsmechanismus zwischen Model und View.

4.2.3 Swing Actions

Um eine durch den Benutzer ausführbare Aktion wie z. B. das Aktivieren einer Schaltfläche oder eines Menüpunktes vom konkreten Widget, das diese Aktion repräsentiert, zu entkoppeln, führt Swing sogenannte *Action*-Objekte ein.

Eine Action enthält Informationen darüber, wie Widgets, die diese Action repräsentieren, dargestellt werden sollen. Zu diesen Informationen gehört z. B. ein Text, der auf Schaltflächen und Menüeinträgen erscheint und die Information, ob die Action gerade ausgeführt werden kann. Nicht-ausführbare Actions werden an der Benutzungsoberfläche meist durch „ausgegraute“ Widgets repräsentiert. Des Weiteren kann eine Action ein Symbol und einen Tooltip-Text enthalten.

Eine Action kann durch mehrere Widgets gleichzeitig repräsentiert werden. So kann eine Action beispielsweise gleichzeitig als Schaltfläche auf einer Werkzeugleiste und als Menüeintrag an der Benutzungsoberfläche repräsentiert werden.

Actions sind eine Umsetzung des Befehlsmusters (*command pattern*, s. [GHJV96]). Die `actionPerformed`-Methode einer Action wird aufgerufen, wenn das entsprechende Ereignis durch den Benutzer ausgelöst wurde.

Konkrete Action-Klassen implementieren das *`Action`*-Interface. Das *`Action`*-Interface wiederum erweitert das *`ActionListener`*-Interface (s. Abb. 4.4).

Actions haben einen Zustand. So ist z. B. von außen über die `setEnabled`-Methode festlegbar, ob eine Action vom Benutzer aktiviert werden kann, oder nicht. Andere Eigenschaften

von Actions können mit Hilfe der `putValue`-Methode gesetzt und entsprechend mit `getValue` wieder ausgelesen werden. Zum modifizieren und sondieren der Eigenschaften wird diesen beiden Methoden jeweils ein String-Schlüssel mitgegeben. Einige Schlüssel werden bereits im *Action*-Interface deklariert.

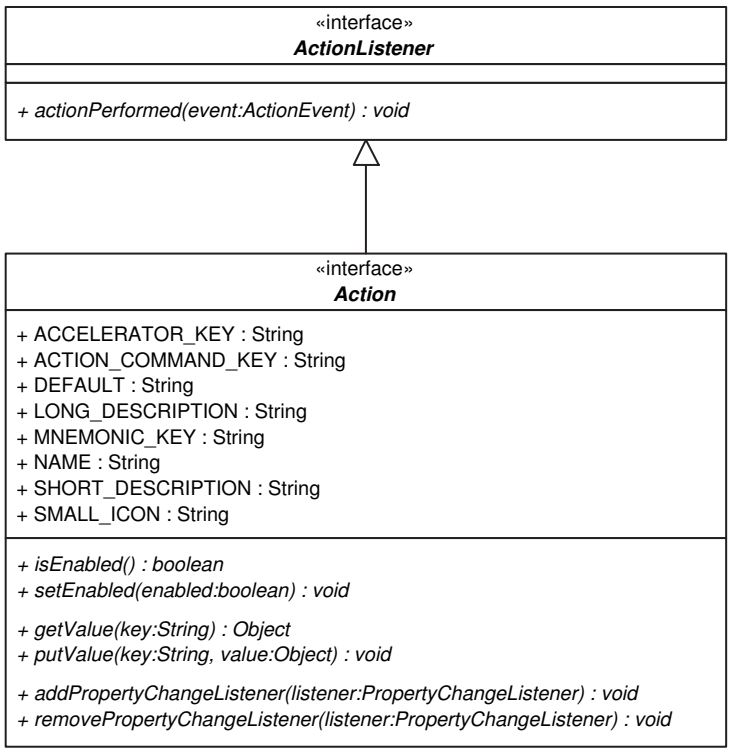

Abbildung 4.4: Das *Action*-Interface

Widgets, die Actions repräsentieren, melden sich an der jeweiligen Action mittels der Methode `addPropertyChangeListener` an und aktualisieren ihr Aussehen, sobald sich eine die GUI betreffende Eigenschaft der Action ändert.

Eine Aktion entspricht einer Funktionalität. Daher bieten Actions die Möglichkeit, Funktionalitäten voneinander zu trennen. Leider gilt dies nur für Funktionalitäten, die an der Benutzungsoberfläche darstellbar und damit vom Benutzer auswählbar sind. Actions können nicht auf andere Ereignisse als einem `ActionEvent` reagieren.

4.2.4 Bewertung

Java Swing ist ein Toolkit, mit dem große Anwendungen realisiert werden können. Beispiele sind die integrierte Entwicklungsumgebung IntelliJ IDEA (siehe [IDE]), Valuatum (s. [Val]) und JWord von Techdigm (s. [JWo]).

Swing bringt eine eigene Architektur namens Model/Delegate mit. Diese Architektur hätte

zu einer besseren Trennung zwischen Ansicht/Interaktion (Delegate) und Daten (Model) führen können, wenn die Model-Interfaces nicht Bestandteil des Swing-Toolkit wären. Mit einer Trennung der Models vom Toolkit könnten andere Toolkits diese Architektur aufgreifen und die selben standardisierten Model-Interfaces verwenden.

Auch die Implementation von Swings Action-Muster ist nur mit Swing einsetzbar. Das *`Action`*-Interface enthält zwar keine Abhängigkeiten zum Swing-Toolkit, befindet sich aber im `javax.swing`-Package. Deshalb sind auch Anwendungen, die zwar keine Swing-GUI besitzen, aber das Action-Interface verwenden, vom Swing-Toolkit abhängig.

4.3 Standard Widget Toolkit (SWT) und JFace

4.3.1 Überblick

Das Standard Widget Toolkit (SWT) ist ein GUI-Toolkit, das aus dem Eclipse Projekt (s. [ECL]) hervorgegangen ist. Anders als Java Swing (s. Abschnitt 4.2) gibt SWT keine Architektur vor. Das Toolkit bietet lediglich Klassen für verschiedene Widgets, aber keine dazugehörenden Model-Interfaces o.ä. an.

Das Rahmenwerk JFace stammt ebenfalls aus dem Eclipse Projekt und setzt auf dem SWT auf. JFace enthält die höheren Konzepte, die im SWT fehlen. Das Rahmenwerk unterstützt die Verbindung von SWT Widgets mit Model Objekten und enthält ein Action-Konzept (s. Abschnitt 4.2).

Anwendungen, die mit SWT erstellt wurden, sind auf viele Plattformen[1] portierbar. Für jede Zielplattform gibt es eine Version des Toolkits. Um die Portierbarkeit zu gewährleisten, sind die Interfaces der SWT-Klassen für alle Plattformen gleich. Um eine möglichst hohe Laufzeitgeschwindigkeit zu erzielen, werden in den Implementierungen des Toolkits – wenn möglich – native Widgets benutzt.

Wie AWT und Swing verwendet auch SWT das Listener-Muster (s. Abschnitt 4.1), um Widgets beobachtbar zu machen.

4.3.2 JFace Viewers

JFace enthält sogenannte Viewer Klassen, die Widgets mit Models verbinden. Die Hauptaufgabe eines Viewers ist, die Informationen eines Models in einem Widget sichtbar zu machen und die Darstellung des Widgets bei Änderung des Modelzustands zu aktualisieren (siehe auch [BG04]).

Um die Darstellung zu beeinflussen, müssen Viewer Klassen meist nicht durch Vererbung erweitert werden. Stattdessen kann die Darstellung oft durch Strategie-Objekte (Strategie-Muster, s. [GHJV96]) bestimmt werden. So kann z. B. an einem Objekt vom `ListViewer`, das eine Liste von Elementen darstellt, eine Strategie vom Typ `ViewerSorter` gesetzt werden, die die Sortierung der anzuzeigenden Elemente bestimmt.

[1] z.Z. Windows 98/ME/2000/XP, Linux (Motif + GTK), Solaris 8, QNX, AIX, HP-UX und Mac OS X

4.3.3 JFace Actions

JFace Actions sind eine Implementierung des in Abschnitt 4.2.3 vorgestellten Action Konzepts für SWT-Widgets. JFace Actions sind vom Typ `IAction`. JFace Actions entsprechen konzeptionell den Swing Actions. Eine Action enthält Informationen darüber, wie sie an der Benutzungsoberfläche dargestellt werden soll und kann durch mehrere Widgets gleichzeitig repräsentiert werden.

4.3.4 Probleme

Ressourcenverwaltung

Benutzer des SWT-Frameworks müssen sich um Ressourcenverwaltung kümmern. Ein mittels `new` erzeugtes Widget muss später mit der `dispose`-Methode freigegeben werden. Dies liegt daran, dass sich eine Virtuelle Java-Maschine zwar darum kümmert, dass die Ressource Arbeitsspeicher automatisch freigegeben wird, wenn sie nicht mehr vom Programm benötigt wird, alle anderen limitierten Ressourcen, wie z. B. Datenbankverbindungen oder sogenannte Window-Handles, aber nicht automatisch verwaltet werden. Hinzu kommt das Fehlen eines deterministischen Destruktors in Java. Zwar können Klassen die in `java.lang.Object` definierte Methode `finalize` redefinieren, um ihre Ressourcen „aufzuräumen“, aber es gibt keine Garantie, wann oder ob diese Methode von der Virtuellen Maschine aufgerufen wird (vergleiche auch [Blo01], Item 6).

Erweiterbarkeit

Die meisten Widget-Klassen des SWT dürfen ausserhalb des SWT-Frameworks nicht erweitert werden. Die Einhaltung dieser Bedingung wird von den Widget-Klassen zur Laufzeit überprüft. Um trotzdem eigene Widget-Klassen implementieren zu können, gibt es von dieser Regel zwei Ausnahmen. Die Klassen `Canvas` und `Composite` können als Basisklassen verwendet werden. Widgets, die von `Canvas` erben, sind einfache Widgets, die sich selbst zeichnen, während `Composite`-Widgets aus mindestens einem anderen Widget aufgebaut sind. Mittels `Composite`-Widgets können Widgets, die nicht durch Vererbung erweiterbar sind, mittels des Kompositum-Musters erweitert werden. Erweiterung mittels Kompositum-Muster ist aufwändiger als Erweiterung durch Vererbung. Funktionalität des im Kompositum enthaltenen Objekts muss durch delegierende Methoden zugänglich gemacht werden. Des Weiteren kann aufgrund unterschiedlicher Typen das Kompositum nicht überall dort eingesetzt werden, wo das enthaltene Objekt eingesetzt werden kann.

4.3.5 Bewertung

Das SWT-Rahmenwerk besitzt einen niedrigeren Abstraktionsgrad als beispielweise Java Swing. Auch ist es durch die vom Entwickler zu beachtende Ressourcenverwaltung etwas schwieriger zu verwenden.

Eine MVC-ähnliche Architektur (wie bei Java Swing ebenfalls ohne eigene Controller-Komponenten), sowie andere abstraktere Konzepte wie z. B. Actions erhält eine Anwendung durch zusätzliche Verwendung des JFace Rahmenwerks. JFace besitzt im Gegensatz zum SWT allerdings Abhängigkeiten zu anderen Eclipse-Bibliotheken und ist damit nicht in der gleichen Weise wie SWT unabhängig von Eclipse einsetzbar[2].

Durch die Trennung der technischen Aspekte (SWT) von den abstrakteren Konzepten (JFace) haben Werkzeugentwickler die Wahl, ob sie für Benutzungsoberflächen das SWT „direkt“, mit dem JFace Rahmenwerk, oder mit einer anderen, evtl. selbst entwickelten Abstraktionsschicht, verwenden möchten.

4.4 JWAM

4.4.1 Überblick

JWAM ist ein Java-basiertes Rahmenwerk, das die Entwicklung von Werkzeugen nach dem Werkzeug- und Materialansatz (WAM) unterstützt (siehe auch [JWA]). Das Rahmenwerk wurde am Arbeitsbereich Softwaretechnik der Universität Hamburg von Studenten und wissenschaftlichen Mitarbeitern entwickelt. JWAM wird nicht nur zu wissenschaftlichen Zwecken in Lehrveranstaltungen und als Grundlage für Studien- und Diplomarbeiten, sondern auch in industriellen Kooperationsprojekten eingesetzt.

Das Rahmenwerk ist aus verschiedenen Schichten aufgebaut. Abbildung 4.5 zeigt eine Übersicht über die JWAM-Schichten (vgl. [GLL+90]).

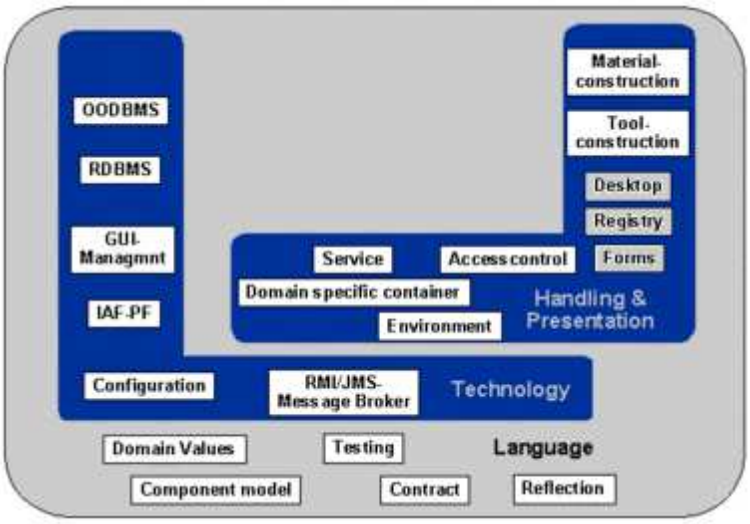

Abbildung 4.5: Die Schichten des JWAM-Rahmenwerks

[2] Van Emmenis zeigt in [Van03], wie JFace ausserhalb von Eclipse verwendet werden kann und welche Bibliotheken dafür aus dem Eclipse Projekt extrahiert werden müssen.

Die Systembasisschicht (*Language*) kapselt technische Komponenten, auf die nicht direkt von Code aus fachlichen Schichten zugegriffen werden soll. Durch diese Abstraktion werden die technischen Komponenten (z. B. GUI-Toolkits, Serialisierungsmechanismen, etc.) austauschbar. Weiterhin enthält die Systembasisschicht allgemeine, in der Sprache Java oder in dessen Standardbibliothek nicht enthaltene Funktionalitäten, wie z. B. eine Unterstützung für die Programmierung nach dem Vertragsmodell (*design by contract*, siehe [Mey97], Seiten 331ff), Behälter-Klassen[3], sowie eine Umsetzung eines Reaktionsmusters.

Die Technologieschicht (*Technology*) stellt eine Brücke zwischen der Systembasisschicht und den höheren Schichten wie der Handhabungs- und Präsentationsschicht und der – nicht im Rahmenwerk enthaltenen – fachlichen Schicht. Die Technologieschicht enthält technologische Komponenten, wie z. B. den JWAM-Message-Broker, der eine Implementation eines Konzeptes zur Interprozess-Kommunikation darstellt. Des Weiteren enthält die Technologieschicht z. B. eine Implementation der in Abschnitt 3.3 beschriebnen Präsentations- und Interaktionsformen.

Die Handhabungs- und Präsentationsschicht (*Handling & Presentation*) enthält hauptsächlich Klassen zur Unterstützung der Werkzeugkonstruktion nach dem Funktion/Interaktion-Muster (siehe Abschnitt 3.3) inklusive Werkzeugkomposition (siehe auch [Bee01]).

4.4.2 Manipulatoren

JWAM enthält z. Z. noch im Alpha-Stadium befindliche Klassen zur Entwicklung von grafischen Werkzeugen. Abbildung 1.3 auf Seite 14 zeigt das Beispiel-Werkzeug *BasicModellingTool*, das mit Hilfe dieser Klassen entwickelt wurde.

Die JWAM-Klassen unterstützen die Entwicklung von Werkzeugen, mit denen Zeichnungen (Materialklasse `Figure`), wie z. B. Klassendiagramme, bearbeitet werden können. Eine Zeichnung besteht aus Zeichnungselementen (Materialklassen `FigureElement` und `FigureConnector`). Die Zeichnung wird auf der Zeichenfläche (`ModellingArea`) angezeigt. Auf welche Weise die Zeichnung vom Benutzer bearbeitet werden kann, wird durch die im Werkzeug gesetzten Manipulatoren (`Manipulator`, s.u.) bestimmt. Abbildung 4.6 zeigt einen Überblick über die wichtigsten Klassen des BasicModellingTools.

Das Beispiel-Werkzeug aus Abbildung 1.3 zeigt – ausgehend von einer gegebenen Klasse – alle Klassen mit ihren Benutzt-Beziehungen an. Das Werkzeug untersucht dazu mit Hilfe der *Java-Reflection-API* die Attribute einer Klasse und baut anhand deren Typen rekursiv einen Klassen-Baum auf. Das Beispiel-Werkzeug zeigt sich selbst, also alle Klassen ausgehend von `toolBasicModellingTool`, an.

Ein Manipulator ist ein befehlsartiges Objekt (s. Befehlsmuster in [GHJV96]), das genau eine Art der Manipulation an einem Material repräsentiert. Im BasicModellingTool gibt es z. B. Manipulator-Klassen zum Hinzufügen, Verschieben und Entfernen von Zeichnungselementen. Abbildung 4.7 zeigt die Oberklasse für aller Manipulatoren des BasicModellingTools.

Die Idee, die Funktionalitäten eines grafischen Editors auf einzelne Befehlsobjekte aufzuteilen, ist nicht neu. Vor dem BasicModellingTool hat z. B. schon das auf dem InterViews

[3] Die JWAM-Behälter-Klassen wurden vor dem neuen, in Java 2 enthaltenen, Collection Framework entwickelt.

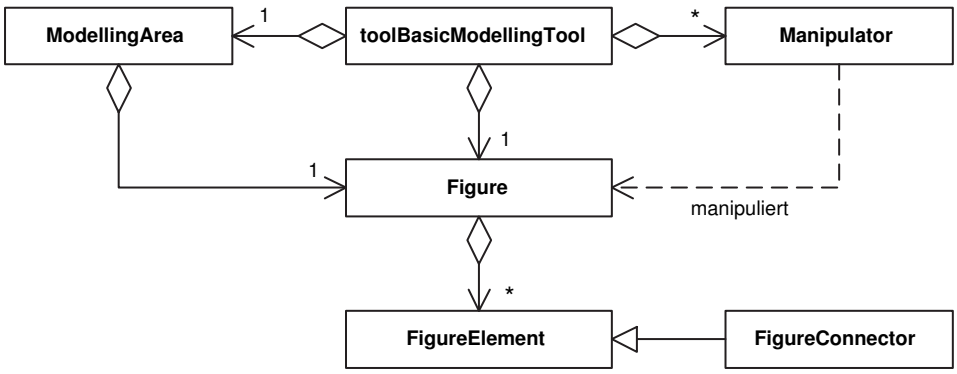

Abbildung 4.6: Aufbau des BasicModellingTools

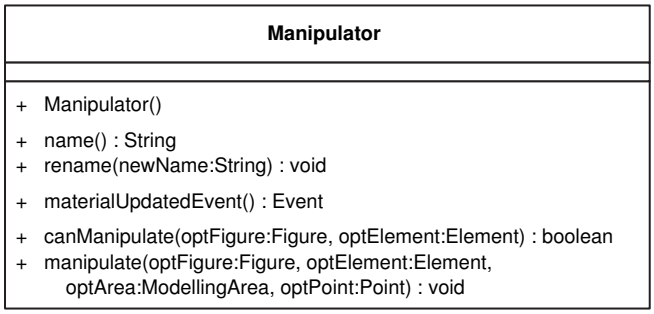

Abbildung 4.7: Oberklasse für Manipulatoren in JWAM

Rahmenwerk (siehe [Int]) basierende UniDraw (siehe [Uni]) eine ähnliche Architekur gehabt.

Manipulatoren haben einen Namen, der zur Anzeige in der GUI (z. B. auf Knöpfen und in Menüs) verwendet werden kann. Manipulatoren können Auskunft darüber geben, ob sie ein bestimmtes Material bearbeiten können (`canManipulate`). Wenn ein Manipulator ein Material verändert hat, wird dies über ein Ereignis (`materialUpdatedEvent`) an die Umgebung gemeldet.

Dem Beispiel-Werkzeug werden zur Laufzeit von außen Manipulatoren zum Hinzufügen von Klassen (*figure element*) und Beziehungen (*figure connector*) sowie zum Entfernen von Zeichnungselementen hinzugefügt. Klassen werden als Rechtecke, in denen der Klassenname steht, angezeigt. Beziehungen werden als Pfeile repräsentiert.

Wenn der Benutzer mit der rechten Maustaste auf die Zeichenfläche klickt, prüft das Werkzeug zunächst, welches Zeichenelement sich an der gegebenen Position befindet. Dann zeigt es dem Benutzer ein Kontext-Menü an, in dem die Namen aller Manipulatoren stehen, die dieses Zeichenelement bearbeiten können. Welche dies sind, prüft das Werkzeug mit der Methode `Manipulator.canManipulate`. Wenn der Benutzer einen Menüeintrag aus-

wählt, wird am entsprechenden Manipulator die Methode `manipulate` aufgerufen. Abbildung 4.8 zeigt diesen Vorgang anhand eines Interaktionsdiagrammes. Das Diagramm zeigt das BasicModellingTool mit nur einem Manipulator.

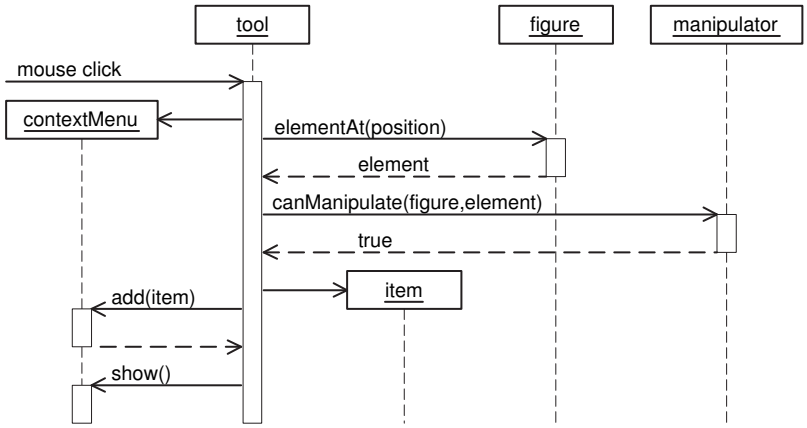

Abbildung 4.8: Dynamische Erzeugung eines Kontext-Menüs

Probleme

Die hier verwendeten Manipulatoren kennen nicht nur die zu bearbeitenden Materialien, sondern haben auch direkten Zugriff auf die Benutzungsoberfläche. Dies ist notwendig, da einige Manipulatoren auf Benutzeraktionen, wie z. B. Mausklicks, das Ziehen der Maus bei gedrückter Maustaste, u.s.w. reagieren müssen. Mittels JWAM Manipulatoren kann also die Anforderung, die Funktionlitäten eines Werkzeugs zu modularisieren, auf Kosten der Trennung von Funktion und Interaktion erfüllt werden.

Die Materialien, die mit dem BasicModellingTool bearbeitet werden können, zeichnen sich selbst auf die Benutzungsoberfläche. Dies ist sehr untypisch für Materialien im WAM-Kontext. Die Lösung, dass sich die darzustellenden Elemente einer Grafik selbst zeichnen, ist in grafischen Anwendungen allerdings sehr verbreitet. Dies liegt daran, dass andere Lösungen entweder softwaretechnisch zweifelhaft sind, oder eine erheblich niedrigere Laufzeitgeschwindigkeit aufweisen.

Eine aus softwaretechnischer Sicht zweifelfhafte Lösung ist die, über alle Elemente einer Grafik zu iterieren, in jeder Iteration zu prüfen, um welchen konkreten Element-Typ es sich jeweils handelt und daraufhin eine entsprechende grafische Ausgabe zu erzeugen. Fowler beschreibt in [Fow99] auf Seite 255, wie eine solche Implementation refaktoriert werden sollte, um eine softwaretechnisch saubere Lösung zu erhalten. Er schlägt vor, die Überprüfung des Element-Typs durch Polymorphie zu ersetzen. Diese Refaktorierung führt wieder zu der Lösung, in der sich die grafischen Elemente selbst zeichnen.

Eine andere Lösung ist die, dass jedes Element eine Beschreibung seiner grafischen Repräsentation enthält, die von aussen abfragbar ist. Nun kann diese grafische Repräsentation

an der Benutzungsoberfläche angezeigt werden. Dies würde das Problem aber nur vom eigentlichen Material zum Repräsentations-Objekt verschieben, wenn die Repräsentation ein Kompositum aus Grafik-Primitiven wie Linie, Kreis, etc. wäre. Die Repräsentation müsste sich nicht selbst zeichnen, wenn sie ausschliesslich aus gleichartigen Grafik-Primitiven, wie z.B. Punkten bestünde. Aber eine solche Lösung würde viel Laufzeiteffizienz einbüßen, da die Funktionen moderner Grafikkarten zur Erzeugung komplexerer Formen ungenutzt blieben.

Die Materialien des BasicModellingTools kennen direkt die grafische Komponente (`Graphics2D`) des Java Swing Toolkits, auf der sie sich zeichnen. Das BasicModellingTool befindet sich in der aktuellen JWAM-Version noch im Alpha-Stadium. Um die Materialien nicht vom GUI-Toolkit abhängig zu machen, sollten die Materialien in späteren Iterationen des Werkzeugs auf einer abstrakten Schnittstelle arbeiten.

4.5 Bewertung und Zusammenfassung

Actions, wie sie sowohl von Java Swing, als auch von JFace implementiert werden, sind ein sinnvolles Konzept zur Trennung einer Funktionalität von seiner Darstellung. Eine Werkzeugarchitektur, die die Funktion eines Werkzeugs in einzelne Funktionalitäten modularisiert, sollte es unterstützen, dass Funktionalitäten möglichst einfach durch Actions repräsentiert werden können.

Die im JWAM Rahmenwerk verwendeten Manipulatoren stellen einen interessanten Ansatz zur Modularisierung der Funktionalitäten eines Werkzeugs dar. So, wie Manipulatoren in JWAM implementiert sind, passen sie allerdings noch nicht zum Action-Konzept, da Manipulatoren ihre Umgebung nicht benachrichtigen, wenn sich eine ihrer die Präsentation betreffenden Eigenschaften ändert. Im Beispielwerkzeug BasicModellingTool müssen sie dies auch nicht, da ihre Repräsentation nicht ständig im Blick ist, sondern nur auf Anfrage des Benutzers (rechter Mausklick auf ein Zeichnungselement) in einem Menü angezeigt werden. Eine Werkzeugarchitektur sollte es aber auch unterstützen, dass die Funktionalitäten eines Werkzeugs beispielsweise auch in sogenannten Werkzeugleisten (*tool bars*) repräsentiert werden.

JWAM-Manipulatoren trennen nicht zwischen Funktion und Interaktion. Dies ist aber eine der in Kapitel 2 aufgestellten Anforderungen an eine Werkzeugarchitektur.

Grafische Werkzeuge stellen eine Herausforderung an die WAM-Metaphern dar. Objekte, die Materialien repräsentieren, greifen im WAM-Kontext nicht selbst auf die Benutzungsschnittstelle zu. Züllighoven schreibt dazu in [Zül98], Seite 168:

> Da in unserem Ansatz Materialen immer durch Werkzeuge oder Automaten bearbeitet werden, ist bei ihnen die Frage der Präsentation und Handhabung kein primäres Problem, denn die Präsentation und Handhabung wird ja stets durch ein Werkzeug vermittelt. Also können wir uns beim Entwurf von Materialien auf die fachliche Funktionalität konzentrieren.

Wie in Abschnitt 4.4.2 beschrieben, ist es aber oft notwendig, dass grafisch darstellbare Materialien – zumindest indirekt – auf die Benutzungsschnittstelle zugreifen. Dies ist immer dann notwendig, wenn ein Werkzeug Materialien bearbeitbar macht, die unterschiedliche grafische Repräsentationen haben und dem Werkzeug nur unter einer gemeinsamen abstrakten Schnittstelle bekannt sind.

Ob dies ein Bruch zum WAM-Ansatz darstellt, hängt auch davon ab, wie der Material-Begriff definiert ist. Es stellt sich die Frage, ob ein Objekt dann ein Material ist, wenn sein dynamischer Typ auschliesslich materialartige Eigenschaften besitzt, oder ob es ausreicht, wenn einer seiner Typen bzw. sein statischer Typ Material-Character hat. In letzterem Fall könnte ein Objekt einerseits die Rolle eines Materials spielen, weil es eine materialartige Schnittstelle implementiert und sich andererseits selbst zeichnen, weil es eine weitere, nichtmaterialartige Schnittstelle (z.B. „`Drawable`“), implementiert.

Im Folgenden wird ein Objekt dann als Material angesehen, wenn es eine Schnittstelle implementiert, die ausschliesslich Materialeigenschaften besitzt. Wenn ein solches Objekt die Rolle eines Materials spielen soll, sollte es auch nur unter dieser Materialschnittstelle bekannt sein.

5 Konzeption eines Manipulatormusters

Die in Kapitel 2 aufgestellten Anforderungen an eine Werkzeugarchitektur werden von keinem der in Kapitel 3 untersuchten Architekturmuster vollständig erfüllt. Das MVC-Muster und die von diesem Muster abgeleiteten Architekturen, die in Java Swing und SWT/JFace verwendet werden, eignen sich vor allem für die Entwicklung von dialogbasierten Werkzeugen, während das PAC-Muster und die Manipulatoren des JWAM Rahmenwerks sich auch bzw. ausschliesslich für dokumentbasierte grafische Werkzeuge eignen.

Dieses Kapitel beschreibt die Konzeption einer Werkzeugarchitektur unter Verwendung der in den vorherigen Kapiteln erarbeiteten Erkenntnisse. Das Ziel ist eine vereinheitlichte Architektur zur Entwicklung von Werkzeugen nach dem WAM-Ansatz.

5.1 Dynamischer Aufbau und Modularisierung der Funktionalitäten

Ein Werkzeug, das von außen erweiterbar sein soll, muss eine Plugin-Architektur (siehe [Fow03]) aufweisen. Eine Voraussetzung für eine Plugin-Architektur ist die Befolgung des *Open-Closed*-Prinzips (s. [Mey97], S. 57ff). Das Werkzeug muss offen für Erweiterungen durch neue Plugins sein. Gleichzeitig soll es gegen Änderungen des eigenen Quelltextes geschlossen sein. Eine weitere Voraussetzung ist, dass weder das Werkzeug, noch die Funktionalität, die dem Werkzeug hinzugefügt werden soll, im Quelltext vorliegen muss.

Es gibt verschiedene Techniken und Entwurfsmuster, die die Umsetzung des Open-Closed-Prinzips unterstützen. Die am häufigsten angewendete Technik zur Erweiterung der Funktionalität einer Klasse ist Vererbung. Die Erweiterung einer Klasse durch Vererbung erfordert keine Änderungen an der beerbten Klasse. Zumindest für die Programmiersprache Java ist auch die zweite Bedingung erfüllt: Weder beerbte Klasse, noch die erbende Klasse muss im Quelltext vorliegen.

Allerdings bedeutet Vererbung immer Erweiterung einer vorhandenen Klasse. Bei einer Plugin-Architektur geht es meist nicht darum, bereits vorhandene Funktionalitäten zu erweitern, sondern darum, neue Funktionalitäten hinzuzufügen. Deshalb eignen sich Verhaltensmuster wie das Befehlsmuster, das Strategiemuster, oder das Besuchermuster (siehe [GHJV96]) besser zur Umsetzung einer Plugin-Architektur.

Das in Abschnitt 4.4.2 beschriebene Manipulator-Muster des JWAM-Frameworks baut auf dem Befehlsmuster auf, so dass sich die JWAM-Manipulatoren gut als Grundlage für eine

Plugin-Architektur eignen. Die JWAM-Manipulatoren werden daher als Ausgangsbasis für das im Folgenden entwickelte allgemeinere Manipulator-Muster dienen.

Manipulatoren erfüllen die Anforderung der Modularisierung der Funktionalitäten, indem jede fachliche Funktionalität durch eine eigene Manipulator-Klasse repräsentiert wird.

5.2 Trennung von Funktion und Interaktion

Für die Trennung von Funktion und Interaktion muss zwischen dialog- und dokumentbasierten Werkzeugen unterschieden werden (vgl. Abschnitt 1.3).

Dialogbasierte Werkzeuge bestehen aus einer Werkzeug- und einer Repräsentationsklasse (GUI). Dies entspricht der Architektur von Monotools im JWAM Rahmenwerk. Anders als bei Monotools werden Funktionalitäten dem Werkzeug in Form von Manipulator-Objekten hinzugefügt. Der Werkzeugzustand kann von Manipulatoren beobachtet werden, so dass die Manipulatoren auf Benutzeraktionen reagieren können.

Für dokumentbasierte Werkzeuge gibt es weniger Gründe, zwischen Funktion und Interaktion zu trennen als für dialogbasierte Werkzeuge, da die Art der Darstellung durch die Materialstruktur vorgegeben ist. Das Material wird sozusagen „direkt“ angezeigt. Eine Trennung von Funktion und Interaktion ist trotzdem wünschenswert, da dies den Wechsel zwischen GUI-Toolkits ermöglicht.

Anders als bei dialogbasierten Werkzeugen ist es für dokumentbasierte Werkzeuge daher nicht problematisch, wenn die Funktion die Interaktion (Zeichenfläche) – zumindest unter einer abstrakten Schnittstelle – kennt. Wenn Manipulatoren auf einer abstrakten Zeichenfläche arbeiten, können verschiedene konkrete Implementationen für verschiedene GUI-Toolkits erstellt werden.

5.2.1 Abstraktion des GUI-Toolkits

Die Implementation einer Toolkit-spezifischen Zeichenfläche ist mit Aufwand verbunden. Dieser Aufwand wird sich mit steigender Anzahl der implementierten grafischen Werkzeuge armortisieren, da die Zeichenflächen-Klassen in den meisten dieser Werkzeuge wiederverwendet werden können. Oft wird die Implementation einer konkreten Zeichenfläche hauptsächlich aus einfachen, weiterleitenden Methoden (*forwarding methods*) bestehen, da moderne Toolkits meist bereits Zeichenflächen-Klassen enthalten (z. B. `Graphics2D` in Java-Swing oder `GC` im Standard Widget Toolkit).

Auch für die in den meisten Toolkits verfügbaren Standard-Dialoge wie z. B. Hinweisdialoge, Dateiauswahldialoge, Druckdialoge etc. kann eine wiederverwendbare Abstraktion mit Toolkit-spezifischen konkreten Implementationen erstellt werden. Dadurch bekommen Manipulatoren die Möglichkeit, mit dem Benutzer über eine abstrakte Schnittstelle zu kommunizieren, ohne die konkrete GUI oder das verwendete Toolkit kennen zu müssen.

5.3 Klassifizierung von Manipulatoren

In den folgenden Abschnitten werden Manipulatoren aufgrund ihrer Funktionalität, der Art und Weise, wie sie ihre Funktionalität ausführen, sowie ihrer Repräsentation und der Art ihrer Aktivierung, unterschieden.

Durch diese Klassifizierung von Manipulatoren wird ein einheitliches Vokabular für Manipulatoreigenschaften geschaffen, das – ähnlich wie die Benennung von Entwurfsmustern – eine einfachere Kommunikation ermöglicht.

5.3.1 Materialmodifizierende und ansichtmodifizierende Manipulatoren

In einem Werkzeug gibt es unterschiedliche Arten von Funktionalitäten. Es gibt Funktionalitäten, die – aufbauend auf der Funktionalität des Domain Models – Materialien modifizieren. Andere Funktionalitäten modifizieren den Werkzeugzustand, was zu einer Änderung der Materialansicht führt.

Beispielsweise gibt es in Textverarbeitungsprogrammen meist eine Funktionalität, um einem Textabschnitt eine andere Schriftart zuzuweisen. Diese Funktionalität modifiziert das Material. Eine andere Funktionalität, die die meisten Textverarbeitungsprogramme besitzen, ist die, das Dokument auf dem Bildschirm zu skalieren. Diese Funktionalität modifiziert nur die Materialansicht, aber nicht das Material selbst.

Entsprechend der Funktionalität, die durch einen Manipulator repräsentiert wird, kann ein Manipulator als materialmodifizierend oder ansichtmodifizierend klassifiziert werden.

5.3.2 Selbstständige und delegierende Manipulatoren

Weiterhin kann zwischen Manipulatoren, die ihre Funktionalität selbstständig ausführen, sowie Manipulatoren, die Aufgaben an andere Komponenten delegieren, um ihre Funktionalität auszuführen, unterschieden werden. Bei diesen Komponenten kann es sich z. B. um andere Manipulatoren, Automaten, Dienste oder auch Subwerkzeuge handeln.

Beispielsweise kann in einem grafischen Editor ein Manipulator ein dialogbasiertes Subwerkzeug öffnen, wenn der Benutzer an einem Material Änderungen vornehmen möchte, die grafisch nicht so leicht durchführbar wären. Abbildung 5.1 zeigt das dokumentbasierte Bildbearbeitungswerkzeug Photoshop, das für die Funktionalität „Ebene duplizieren“ ein dialogbasiertes Subwerkzeug öffnet.

5.3.3 Aktive und passive Manipulatoren

Manipulatoren führen ihre Funktionalität aufgrund unterschiedlicher Ereignisse aus.

Ein Manipulator, der in einem dokumentbasierten, grafischen Werkeug für das Verschieben von Elementen auf der Zeichenfläche verantwortlich ist, reagiert darauf, dass der Benutzer

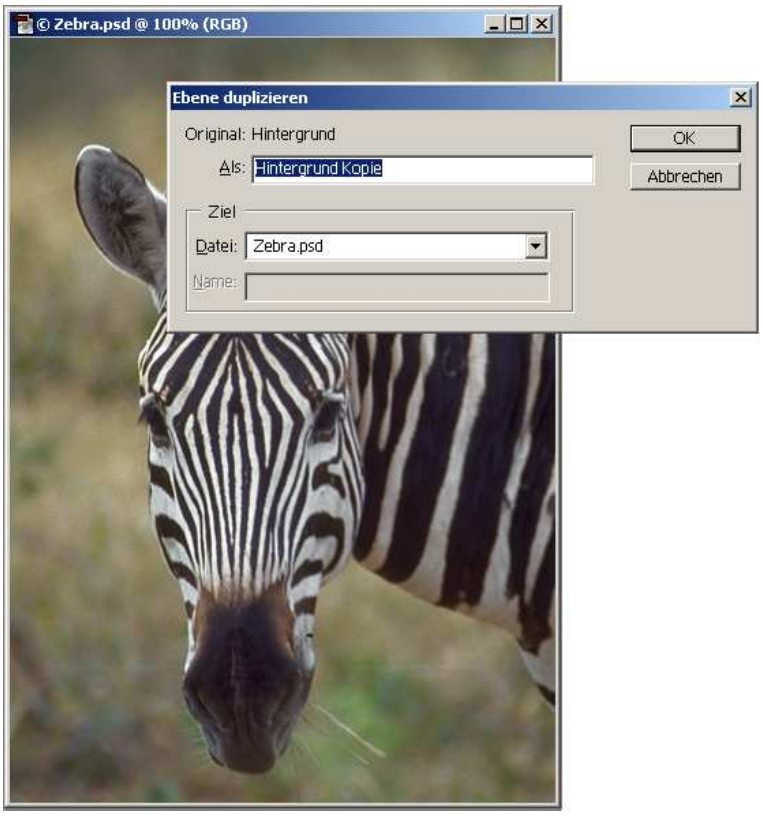

Abbildung 5.1: Subwerkzeug des Bildbearbeitungswerkzeugs Photoshop

ein Grafikelement auswählt und dann die Maus bei gedrückter Maustaste über die Zeichenfläche bewegt. Ein solcher Manipulator hat keine grafische Darstellung. Er ist für den Benutzer unsichtbar. Der Manipulator manifestiert sich nur durch seinen durch die Benutzeraktion hervorgerufenen Effekt. Da solche Manipulatoren selbstständig aktiv werden, wenn der Benutzer eine bestimmte Aktion ausführt, werden sie im Folgenden als *aktive Manipulatoren* bezeichnet.

Andere Manipulatoren, wie z. B. ein Manipulator zum Entfernen eines grafischen Elements von der Zeichenfläche, haben eine grafische Repräsentation. Meist werden dem Benutzer solche Manipulatoren als Knöpfe oder Menüelemente angezeigt. Diese Manipulatoren werden nicht selbstständig aktiv. Sie warten darauf, dass der Benutzer sie explizit durch Auswahl der entsprechenden Schaltfläche oder des Menüeintrags aktiviert. Daher werden solche Manipulatoren im Folgenden als *passive Manipulatoren* bezeichnet.

5.4 Grafische Darstellung passiver Manipulatoren

Die Repräsentation eines passiven Manipulators soll unabhängig vom verwendeten GUI-Toolkit sein. Daher stellen Manipulatoren keine konkreten Widgets für ihre Darstellung zur

Verfügung. Stattdessen besitzen sie Informationen darüber, wie ein konkretes Widget den Manipulator darstellen soll.

Um einen Manipulator an der Benutzungsoberfläche darstellen zu können, benötigt er mindestens einen Text, der auf dem Widget angezeigt werden soll. Zusätzlich kann ein Manipulator ein Symbol besitzen, das ihn grafisch repräsentiert. Symbole werden in unterschiedlichen Toolkits durch unterschiedliche Klassen repräsentiert, weshalb ein Manipulator nicht das Symbol selbst, sondern eine URL enthält, die angibt, wo sich die Grafik befindet. Ausserdem muss ein passiver Manipulator die Information, ob er gerade ausgeführt werden kann, zur Verfügung stellen. Manipulatoren, die aufgrund des aktuellen Werkzeugzustands gerade nicht ausführbar sind, können dann vom Toolkit entsprechend gekennzeichnet werden. In der Regel geschieht dies durch „ausgrauen“ der entsprechenden Schaltfäche oder des Menüeintrags.

Im JWAM BasicModellingTool werden Manipulatoren nur in Kontext-Menüs angezeigt. Wenn der Benutzer ein Grafikelement mit der rechten Maustaste anklickt, erzeugt das Werkzeug ein Kontextmenü, das nur die gerade aktivierbaren Manipulatoren anzeigt. Wenn ein Manipulator aber ständig im Blick sein soll (z. B. als Schaltfäche auf einer Werkzeugleiste), muss er seine Umgebung über Änderungen seines Zustandes informieren, damit das den Manipulator repräsentierende Widget entsprechend aktualisiert werden kann. Passive Manipulatoren müssen also mittels eines Reaktionsmechanismus' beobachtbar sein.

Umgekehrt müssen Manipulatoren mit grafischer Repräsentation den Werkzeugzustand beobachten, um nach jeder Änderung entscheiden zu können, ob sie gerade ausgeführt werden können und dann ggf. ihre Repräsentation aktualisieren zu können.

5.5 Beobachtbarkeit von Manipulatoren

Die Funktion eines Werkzeugs soll beobachtbar sein, damit die Umgebung – meist die Interaktion – auf Änderungen am Material und am Werkzeugzustand reagieren kann.

Bei Werkzeugarchitekturen, bei denen eine Interaktionskomponente auf Änderungen genau einer Funktionskomponente reagieren soll, kann die Funktionskomponente beobachtbar gemacht werden. Die IAK meldet sich dann bei der FK an und wird danach über Änderungen am Werkzeug- und Materialzustand benachrichtigt.

Bei der Verwendung von Manipulatoren stimmt die Granularität der Interaktion nicht mit der der Funktion überein. Am Beispiel des BasicModellingTools des JWAM-Rahmenwerks ist zu sehen, dass es zu einer Interaktionskomponente beliebig viele Funktionskomponenten (Manipulatoren) geben kann.

Die Interaktion muss sich nun nicht mehr nur an einer Komponente, sondern an vielen Komponenten anmelden, um über Änderungen am Werkzeug- und Materialzustand benachrichtigt zu werden. Diesen Weg geht auch das BasicModellingTool.

Aber nicht nur die Interaktion muss über diese Änderungen benachrichtigt werden. Oft müssen auch Manipulatoren selbst wissen, ob andere Manipulatoren das Material bearbei-

tet haben. So müssen beispielsweise passive Manipulatoren ihre Repräsentationen ständig entsprechend des Werkzeug- und Materialzustandes aktualisieren.

Wenn ein Beobachter sich an allen Manipulatoren anmelden muss, muss also nicht nur die Interaktion sich an jedem Manipulator anmelden, sondern auch jeder Manipulator an jedem anderen Manipulator. Abbildung 5.2 zeigt die Abhängigkeiten zwischen einer Interaktion und nur drei Manipulatoren. Jedes Objekt benachrichtigt und/oder sondiert jedes andere Objekt. Die Anzahl der Abhängigkeiten steigt damit exponential mit der Anzahl der Manipulatoren.

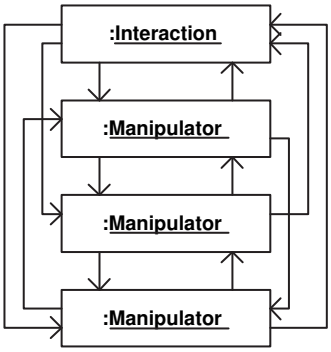

Abbildung 5.2: Sich gegenseitig beobachtende Manipulatoren

Ein Verschieben der Zuständigkeit für Benachrichtigungen vom Manipulator zum Material ist keine Lösung dieses Problems. Wenn es sich bei dem Material um ein Materialgeflecht handelt, gibt es auch hier wieder zu viele Komponenten, die beobachtet werden müssten. Ausserdem könnten Materialien ihre Umgebung nur über Änderungen ihres eigenen Zustands, nicht aber über Änderungen am Werkzeugzustand benachrichtigen.

Eine Lösung des Problems wäre es aber, eine Hülle für alle Manipulatoren eines Werkzeugs zu schaffen, die die Manipulatoren beobachtet und als Reaktion auf Ereignisse ihrerseits ihre Umgebung über diese Ereignisse benachrichtigt. Beobachter müssten sich nun nur noch an diesem Hüllen-Objekt statt an allen Manipulatoren anmelden. Dieses Hüllen-Objekt wäre damit eine Implementation des Fassaden-Musters (siehe [GHJV96]). Abbildung 5.3 zeigt wieder die Abhängigkeiten zwischen einer Interaktion und drei Manipulatoren. Die Anzahl der Abhängigkeiten wächst in dieser Lösung nur noch proportional mit der Anzahl der Manipulatoren.

Abweichend von der Beschreibung des Fassaden-Musters in [GHJV96], wird hier nicht nur von außen über das Hüllen-Objekt auf die Manipulatoren zugegriffen. Auch die Manipulatoren untereinander werden mit Hilfe des Hüllen-Objekts benachrichtigt. Um eine lose Kopplung zwischen Manipulator-Klasse und Hüllen-Klasse zu erreichen, darf die Hüllen-Klasse die Manipulatoren nur unter einer abstrakten Schnittstelle kennen. Abbildung 5.4 zeigt die Abhängigkeiten zwischen den Klassen.

Eine andere Lösung ist, eine zentrale „Benachrichtigungsstelle“ einzurichten. Genau wie am oben beschriebenen Hüllen-Objekt melden sich alle Beobachter an diesem zentralen Er-

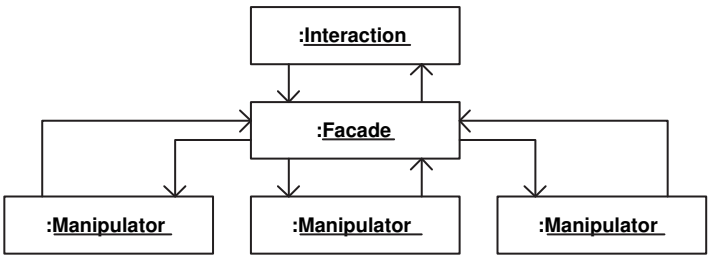

Abbildung 5.3: Manipulator-Fassade: Objekt-Beziehungen

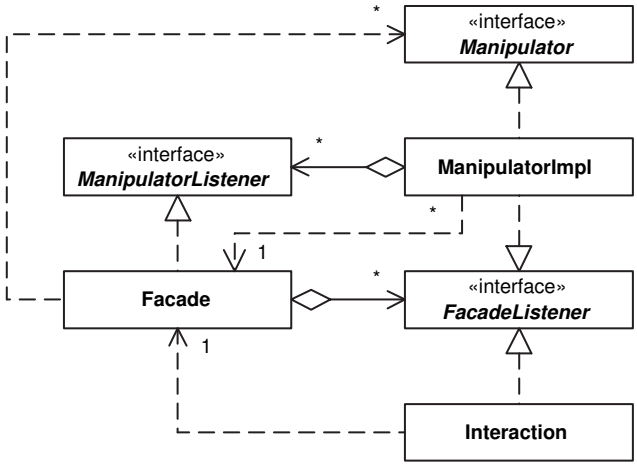

Abbildung 5.4: Manipulator-Fassade: Typ-Beziehungen

eignisverteiler an. Anders als das Hüllen-Objekt beobachtet der Ereignisverteiler die Manipulatoren nicht. Stattdessen geben Manipulatoren dem Ereignisverteiler, nachdem sie den Werkzeug- oder Materialzustand geändert haben, den Auftrag, die Umgebung über diese Änderung zu informieren.

Abbildung 5.5 zeigt auch für diese Lösung ein Interaktionsdiagramm mit drei Manipulatoren. Wie in der vorherigen Lösung wächst die Anzahl der Abhängigkeiten zwischen den Objekten proportional mit der Anzahl der Manipulatoren.

Anders als in der vorherigen Lösung muss der Ereignisverteiler (`EventDispatcher`) die Manipulatoren nur noch als abstrakten Beobachter kennen. Da sich der Ereignisverteiler im Gegensatz zum Hüllen-Objekt nicht mehr an den Manipulatoren anmeldet, müssen Manipulatoren selbst nicht mehr beobachtbar sein. Der Ereignisverteiler ist eine Umsetzung des Vermittler-Musters (siehe [GHJV96]). Wie das Klassendiagramm in Abbildung 5.6 zeigt, gibt es in dieser Lösung deutlich weniger statische Abhängigkeiten als in der Fassaden-Lösung.

Implementationen des Manipulator-Musters sollten den Ereignisverteiler verwenden, um Manipulatoren beobachtbar zu machen. Der Ereignisverteiler minimiert die Anzahl der Be-

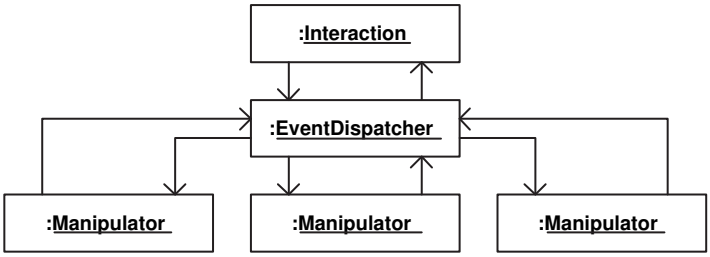

Abbildung 5.5: Ereignisverteiler: Objekt-Beziehungen

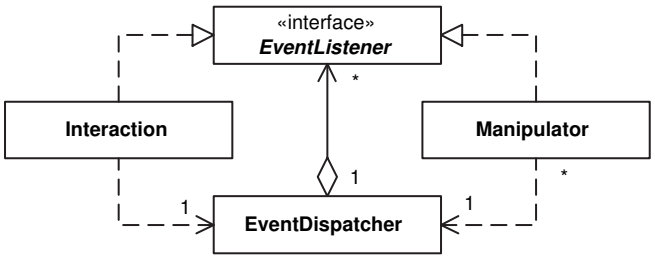

Abbildung 5.6: Ereignisverteiler: Typ-Beziehungen

ziehungen zwischen Objekten und erfordert nur an einer zentralen Stelle die Programmlogik eines Subjekts[1].

[1]Subjekt = beobachtetes Objekt

6 Umsetzung des Manipulatormusters

Dieses Kapitel beschreibt die Umsetzung des im vorangegangenen Kapitel erarbeiteten Manipulator-Musters anhand der Konstruktion eines konkreten Werkzeugs auf Basis von Manipulatoren.

Es wird ein Werkzeug zur Erstellung von UML-Klassendiagrammen entwickelt. Das Werkzeug wird hauptsächlich ein dokumentbasiertes, grafisches Werkzeug sein. Es wird aber auch Elemente eines dialogbasierten Werkzeugs besitzen. Daher werden Manipulatoren für beide Werkzeugarten benötigt.

Um die Möglichkeit einer Toolkit-unabhängigen Implementation zu zeigen, wird das Werkzeug in zwei Versionen entwickelt. Eine Version wird das Swing-Toolkit von Sun verwenden. Die andere Version verwendet das Standard Widget Toolkit des Eclipse Projekts. Beide Versionen sollen sich den größten Teil des Quellcodes teilen. Vor allem für die fachlichen Funktionalitäten der Werkzeuge soll der gleiche Code verwendet werden.

Im Folgenden wird zunächst in Abschnitt 6.1 ein Rahmenwerk entwickelt, das wiederverwendbare Klassen zur Erstellung von Manipulator-basierten Werkzeugen enthält. Dann wird in Abschnitt 6.2 das UML-Werkzeug mit Hilfe dieses Rahmenwerks konstruiert.

6.1 Das Rahmenwerk

6.1.1 Entwicklung einer Toolkit-unabhängigen Zeichenfläche

Um Unabhängigkeit vom verwendeten Toolkit zu erreichen, muss für grafische Werkzeuge eine Toolkit-unabhängige Zeichenfläche erstellt werden.

Es wird eine abstrakte Klasse `DrawingArea` erstellt. Für jedes Toolkit wird eine konkrete Unterklasse von `DrawingArea` erzeugt: `SwingDrawingArea` und `SWTDrawingArea`. Manipulatoren sollen keine der konkreten Implementationen kennen, sondern auf der Schnittstelle von `DrawingArea` arbeiten.

Abbildung 6.1 zeigt die Klassenhierarchie von `DrawingArea`. Die Methode `drawLine` steht hier stellvertretend für die Menge von Operationen zum Zeichnen auf der Zeichenfläche.

Theoretisch würde es genügen, wenn es in `DrawingArea` nur eine abstrakte Methode zum Zeichnen eines Punktes auf der Zeichenfläche gäbe. Alle anderen Zeichenoperationen könnten dann in `DrawingArea` mit Hilfe dieser Methode implementiert werden. Auf diese Mög-

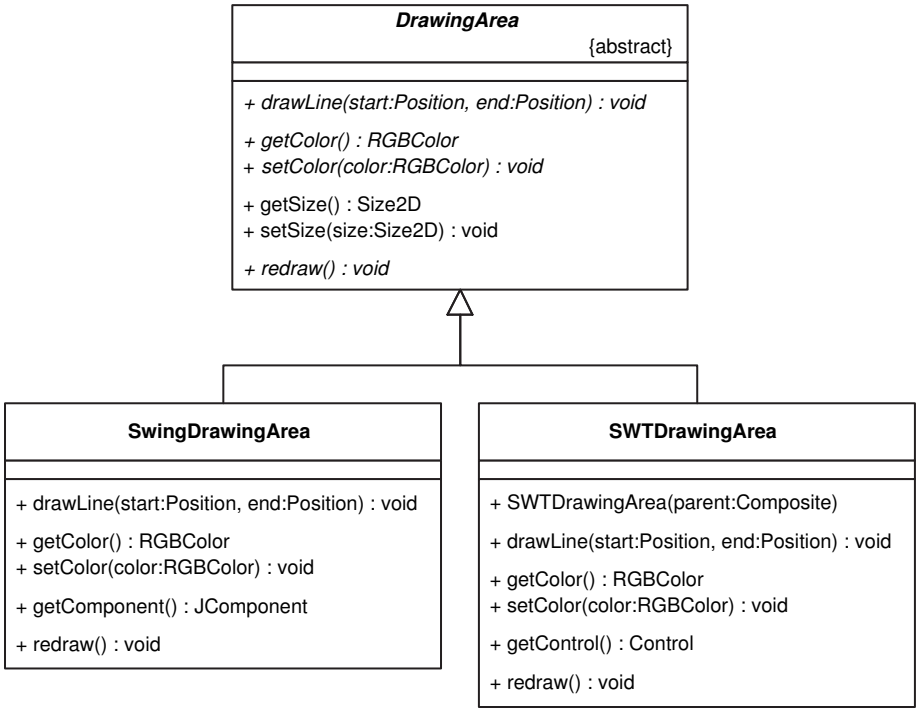

Abbildung 6.1: Klassenhierarchie: `DrawingArea`

lichkeit wird hier zugunsten der Möglichkeit verzichtet, komplexere Formen sehr laufzeiteffizient direkt durch die Hardware zeichnen zu lassen. Diese Möglichkeit entsteht dadurch, dass die konkreten Implementationen `SwingDrawingArea` und `SWTDrawingArea` jeweils Widgets verwenden, die bereits Methoden zum Zeichnen von Formen wie Rechtecken und Ellipsen besitzen. Dadurch besteht die Möglichkeit, dass diese Aufrufe bis zum Grafiktreiber weitergeleitet werden.

Die Methode `setColor` bestimmt die Farbe, mit der die nächste Zeichenoperation ausgeführt wird. Mit Hilfe der `redraw`-Methode wird die jeweilige konkrete Zeichenfläche dazu veranlasst, sich selbst neu zu zeichnen.

Die konkrete Klasse `SwingDrawingArea` besitzt eine Methode, über die das zur Darstellung verwendete Toolkit-Objekt sondierbar ist (`getComponent`). Diese Methode wird benötigt, damit das Toolkit-Objekt in die Benutzungsoberfläche integriert werden kann. Für die SWT-Benutzungsoberfläche ist eine solche Methode nicht nötig. Hier wird das Toolkit-Objekt durch Setzen des *Parent*-Objektes im Konstruktor in die Benutzungsoberfläche integriert. Daher besitzt die Klasse `SWTDrawingArea` einen Konstruktor, der ein solches Parent-Objekt übergeben bekommt.

Die Zeichenfläche muss vom Werkzeug und von Manipulatoren beobachtbar sein. So ist es für ein grafisches Werkzeug z. B. wichtig zu wissen, welche Aktionen der Benutzer mit der Maus auf der Zeichenfläche ausführt. Daher wird für die Zeichenfläche eine Beobachter-Schnittstelle `DrawingAreaListener` erstellt. Die Zeichenflä-

che wird um Methoden zum Hinzufügen (`addDrawingAreaListener`) und Entfernen (`removeDrawingAreaListener`) von Beobachtern erweitert. Abbildung 6.2 zeigt die erweiterte `DrawingArea` Klasse sowie die Beobachter-Schnittstelle.

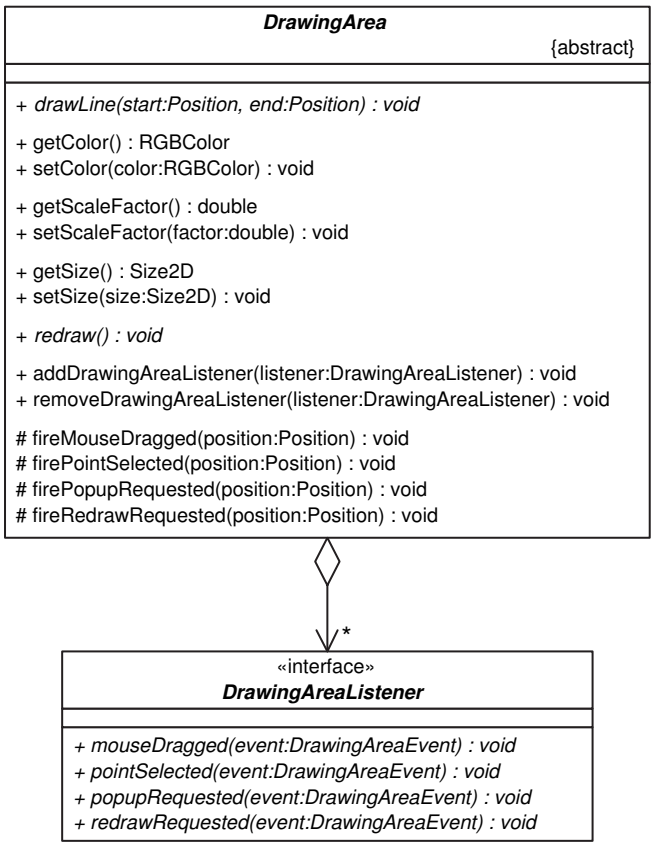

Abbildung 6.2: Klasse `DrawingArea` und Beobachter-Schnittstelle

Mit Hilfe der Methoden `fireMouseDragged`, `firePointSelected`, `firePopupRequested` und `fireRedrawRequested` können die konkreten Unterklassen von `DrawingArea` Beobachtern mitteilen, dass die Maus bei gedrückter linker Maustaste über die Zeichenfläche bewegt wurde, dass der Benutzer einen Punkt auf der Zeichenfläche angeklickt hat, dass die rechte Maustaste über der Zeichenfläche gedrückt wurde, oder dass die Grafik neu aufgebaut werden muss.

Die in Abbildung 6.2 gezeigte `DrawingArea`-Klasse besitzt Methoden zum Setzen und Sondieren eines Skalierungsfaktors (*scale factor*). Der Skalierungsfaktor ist initial mit dem Wert 1.0 belegt und bestimmt, in welcher Vergrößerung eine Zeichnung auf der Zeichenfläche angezeigt wird.

6.1.2 Basisklassen für grafisch darstellbare Materialien

Als Oberklasse für alle auf einer Zeichenfläche darstellbaren Materialien dient die Klasse `DrawingElement` (siehe Abbildung 6.3). Zeichnungselemente, die gemeinsam dargestellt und bearbeitet werden sollen, werden zu einer Zeichnung (`Drawing`) zusammengefasst.

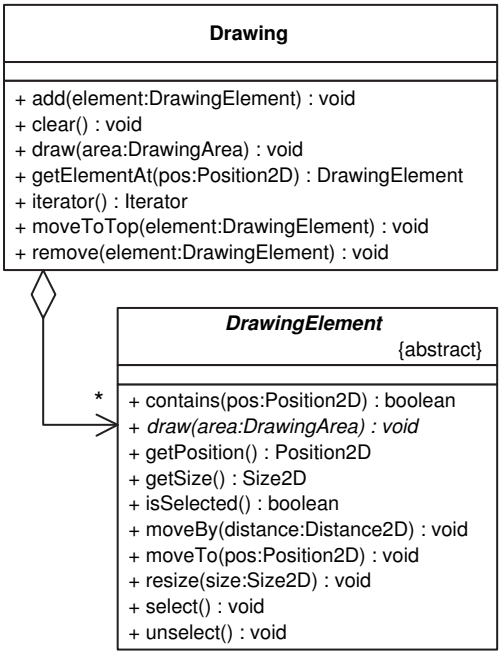

Abbildung 6.3: Materialien: Zeichnung und Zeichnungselemente

Anders als beim JWAM BasicModellingTool ist die Zeichenfläche nicht vom darauf abgebildeten Material abhängig. Die Rahmenwerksklassen `Drawing` und `DrawingElement` sind also optional zu verwenden. Dies erhöht die Wiederverwendbarkeit der Zeichenfläche.

Eine Zeichnung vom Typ `Drawing` kennt die abstrakte Zeichenflächenklasse `DrawingArea` und kann sich selbst über die `draw`-Methode auf die Zeichenfläche zeichnen. Hierfür ruft die Zeichnung an jedem Zeichnungselement dessen abstrakte `draw`-Methode auf, die von konkreten Materialklassen implementiert werden muss.

Die Klassen `Drawing` und `DrawingElement` enthalten des Weiteren Methoden, die in den meisten grafischen Werkzeugen benötigt werden. So kann beispielsweise an einer Zeichnung abgefragt werden, welches Element an einer gegebenen Position auf der Zeichenfläche liegt (`getElementAt`). Zeichnungselemente können verschoben (`moveBy`) und selektiert (`select`) werden.

6.1.3 Manipulatoren

Das Rahmenwerk enthält bereits einige Manipulatoren zum Bearbeiten von Zeichnungen vom Typ `Drawing`. Als gemeinsame Oberklasse dient die Klasse `DrawingManipulator` (siehe Abbildung 6.4). Zeichnungs-Manipulatoren kennen sowohl die Zeichnung, als auch die Zeichenfläche, auf der sie arbeiten.

Aktive Manipulatoren

Abbildung 6.4 zeigt die Klassenhierarchie der aktiven Manipulatoren. Es fällt auf, dass `SelectionManipulator` und `MoveManipulator` keine Methoden besitzen, mit denen Einfluss auf die Manipulatoren genommen werden kann. Dies liegt daran, dass diese Manipulatoren aktive Manipulatoren sind und ihre Aufgabe daher selbstständig wahrnehmen. Sie müssen nicht von außen gesteuert werden.

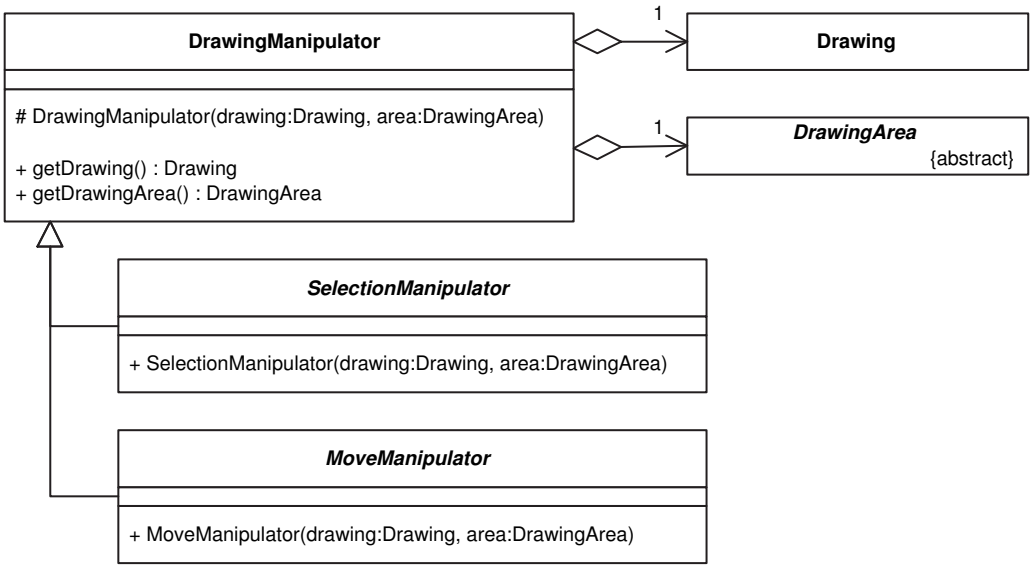

Abbildung 6.4: Aktive Manipulatoren

Ein `SelectionManipulator` reagiert auf Mausklicks auf der Zeichenfläche. Wenn sich an der Position, an der die linke Maustaste gedrückt wurde, ein Zeichnungselement befindet, markiert der Manipulator dieses Element als selektiert.

Ein `MoveManipulator` reagiert auf Mausbewegungen bei gedrückter Maustaste. Wenn die linke Maustaste über einem Zeichnungselement gedrückt wurde und dann die Maus, ohne die Taste loszulassen, bewegt wird, verschiebt der Manipulator das Zeichnungselement entsprechend der Mausbewegung.

Passive Manipulatoren

Passive Manipulatoren benötigen eine Repräsentation an der Benutzungsoberfläche, damit der Benutzer sie aktivieren kann. Abbildung 6.5 zeigt die Klasse `ManipulatorRepresentation`, die Informationen darüber enthält, durch welchen Text und welches grafische Symbol ein Manipulator repräsentiert werden soll. Ausserdem ist an Objekten dieser Klasse sondierbar, ob der dazugehörige Manipulator gerade aktiviert werden kann, oder nicht.

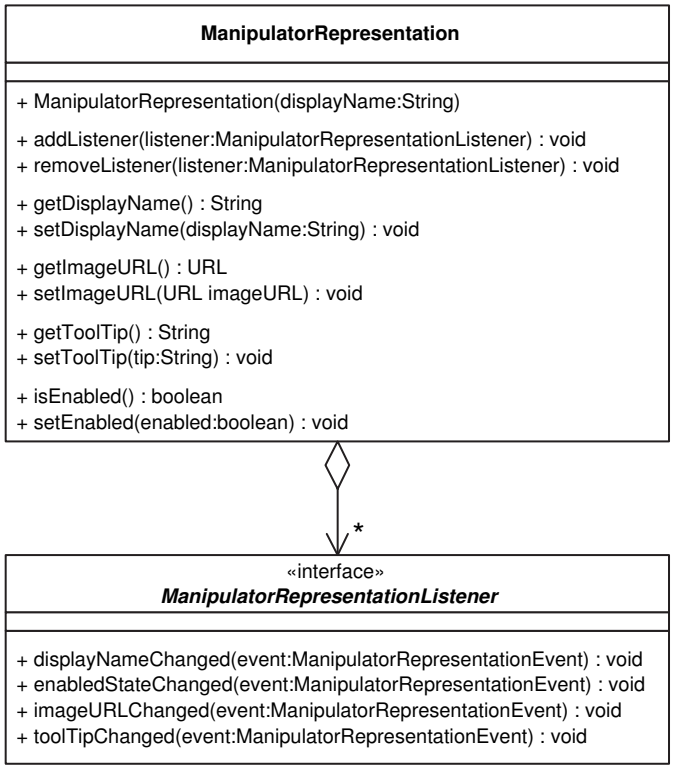

Abbildung 6.5: Manipulator-Repräsentation mit Beobachter-Schnittstelle

Ein `ManipulatorRepresentation`-Objekt ist mit Hilfe der Beobachter-Schnittstelle `ManipulatorRepresentationListener` beobachtbar. Dadurch kann die Benutzungsoberfläche auf Änderungen des Zustandes einer `ManipulatorRepresentation` reagieren, indem sie das dem Manipulator zugeordnete Widget entsprechend aktualisiert.

Abbildung 6.6 zeigt die Schnittstelle `PassiveManipulator`, die passive Manipulatoren implementieren können. Über diese Schnittstelle kann die Repräsentation des Manipulators sondiert werden. Über die Methode `manipulate` wird der Manipulator aktiviert.

Für das Rahmenwerk werden zwei konkrete, ansichtmodifizierende Manipulatoren zur Vergrößerung sowie zur Verkleinerung der Anzeige (*zoom in / zoom out*) implementiert. Als gemeinsame Oberklasse für alle passiven Manipulatoren, die auf Materialien vom Typ `Drawing` arbeiten, dient die Klasse `PassiveDrawingManipulator`. Abbildung 6.7 zeigt die

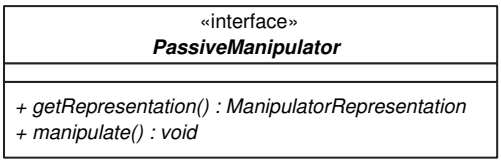

Abbildung 6.6: Schnittstelle für passive Manipulatoren

Klassenhierarchie der passiven Manipulatoren im Rahmenwerk.

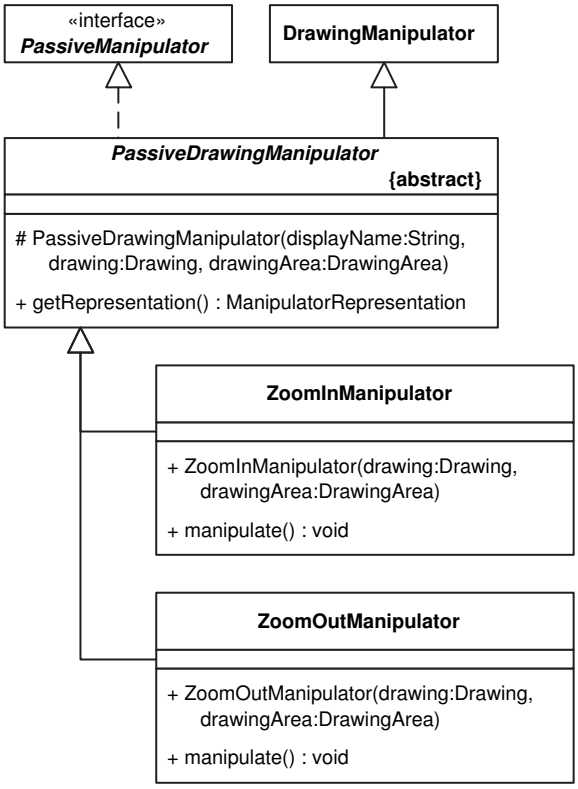

Abbildung 6.7: Konkrete passive Manipulatoren

6.2 Das UML-Werkzeug

Im Folgenden wird ein Beispielwerkzeug auf Basis des Manipulator-Musters entwickelt. Das Beispielwerkzeug ist ein dokumentbasiertes Werkzeug zur Erstellung und Bearbeitung von UML-Klassendiagrammen.

Um zu zeigen, dass das Konzept der Toolkit-unabhängigen Manipulatoren praktisch umgesetzt werden kann, wird das UML-Werkzeug parallel in zwei Versionen mit zwei un-

terschiedlichen GUI-Toolkits entwickelt. Abbildung 6.8 zeigt das UML-Werkzeug in der Swing- sowie in der SWT-Version[1].

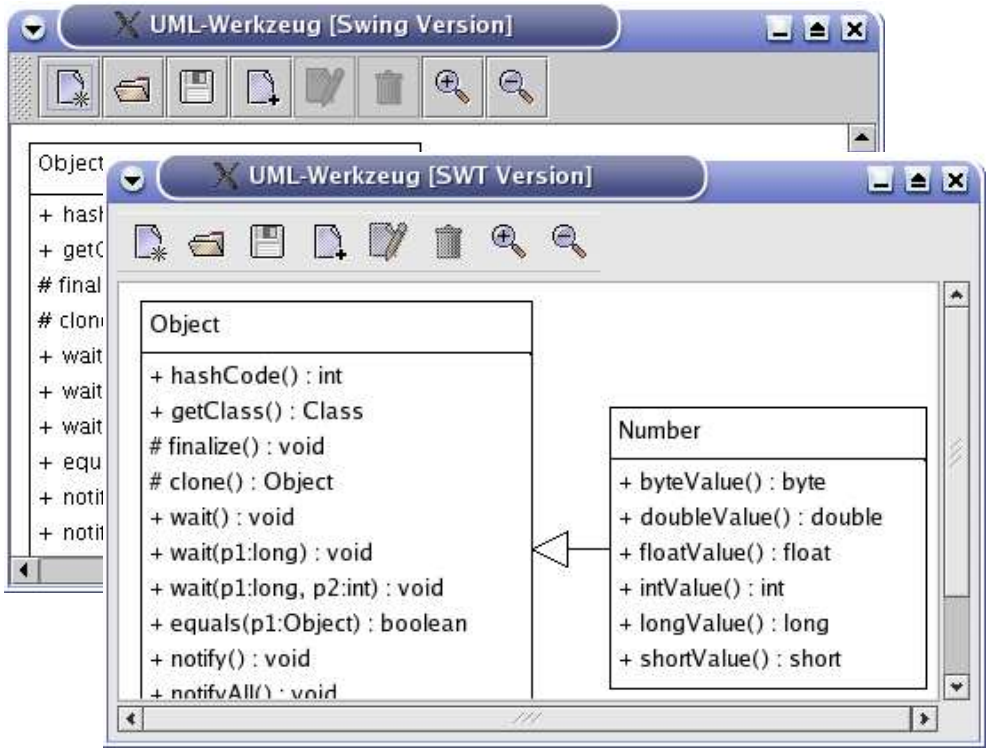

Abbildung 6.8: Die zwei Versionen des UML-Werkzeugs

Zusätzlich zum dokumentbasierten, grafischen Hauptwerkzeug wird ein dialogbasiertes Subwerkzeug entwickelt (siehe Abbildung 6.9), mit dem die Eigenschaften (u.a. Methoden und deren Parameter) einer Klasse bearbeitet werden können.

6.2.1 Die Materialien

Das UML-Werkzeug macht ein Materialgeflecht bearbeitbar. Dieses Materialgeflecht besteht aus UML-Klassen (Typ `UMLClass`), die untereinander in Beziehung stehen können. Bei diesen Beziehungen handelt es sich um Vererbung, Aggregation und Komposition (vgl. z. B. [FS99]). Abbildung 6.10 zeigt eine Übersicht über die Materialklassen des UML-Werkzeugs.

Eine UML-Klasse ist ein grafisch darstellbares Material und erbt deshalb von `Drawing-Element`. UML-Klassen besitzen einen Namen und eine Liste von UML-Methoden (Typ `UMLMethod`). Ausserdem können sie Abhängigkeiten zu anderen Klassen besitzen. Abbildung 6.11 zeigt die Operationen der Materialklasse `UMLClass`.

[1]Die verwendete SWT-Bibliothek ist in diesem Beispiel die GTK+ Version für Linux.

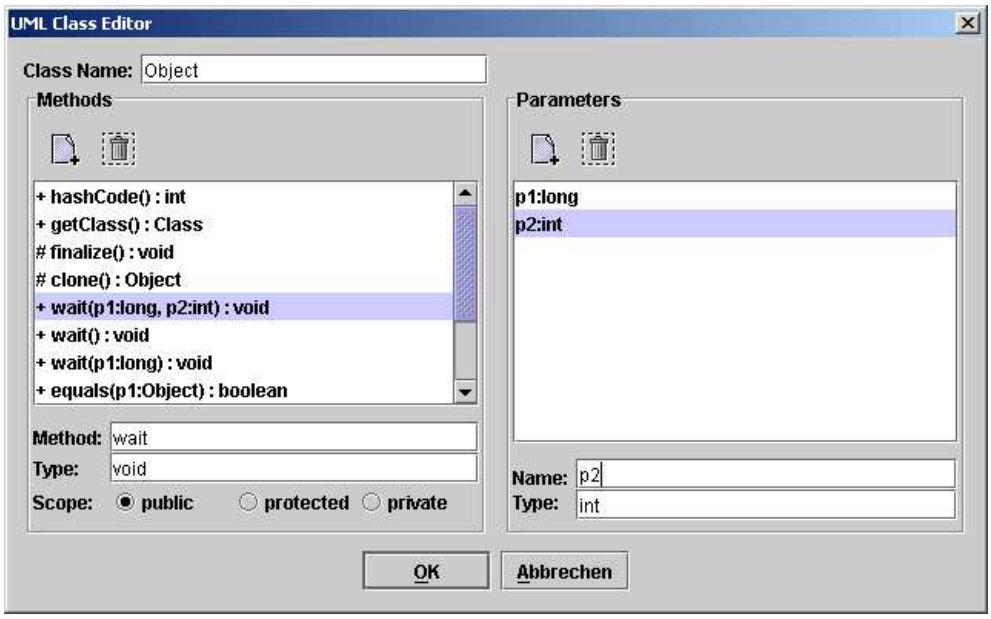

Abbildung 6.9: Das Subwerkzeug „UML Class Editor“

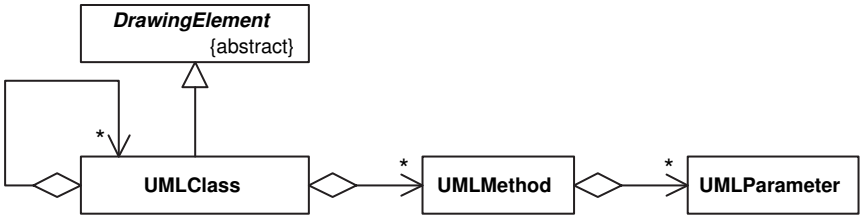

Abbildung 6.10: Die Material-Klassen des UML-Werkzeugs

UML-Methoden besitzen einen Namen, einen Rückgabetyp, sowie eine Liste von UML-Parametern (Typ `UMLParameter`). Abbildung 6.12 zeigt die Operationen der Materialklasse `UMLMethod`.

UML-Parameter bestehen aus einem Namen und einem Typ. Abbildung 6.13 zeigt die Operationen der Materialklasse `UMLParameter`.

6.2.2 Manipulatoren

Plugin-Fähigkeit

Die Manipulatoren, die das UML-Werkzeug verwenden soll, werden in einer externen Text-Datei beschrieben. Diese XML-Datei enthält die Namen der konkreten Manipulator-Klassen. Diese Klassen werden vom Werkzeug geladen. Für das UML-Werkzeug gilt die Vereinba-

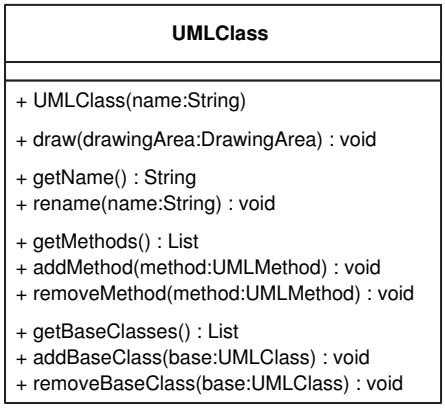

Abbildung 6.11: Die Operationen der Materialklasse `UMLClass`

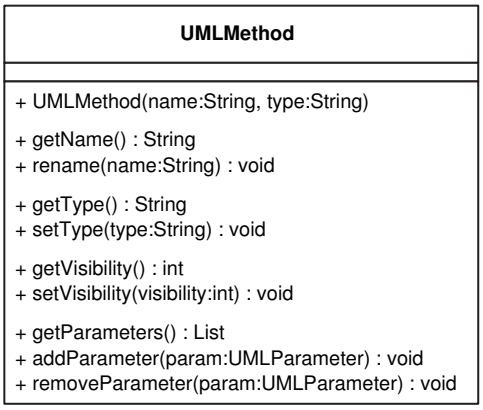

Abbildung 6.12: Die Operationen der Materialklasse `UMLMethod`

rung, dass Manipulatoren einen Konstruktor besitzen, der genau zwei Argumente besitzt. Das erste Argument ist die zu bearbeitende Zeichnung. Der zweite Parameter ist die Zeichenfläche, auf der die Zeichnung dargestellt wird. Für jede in der XML-Datei angegebene Manipulator-Klasse wird dieser Konstruktor mittels Javas Reflection API zur Erzeugung jeweils eines Objektes aufgerufen.

Manipulatoren

Das UML-Werkzeug erhält passive Manipulatoren zum Laden und Speichern von Diagrammen, sowie zum Anlegen, Bearbeiten und Löschen von Klassen, Methoden und Parametern. Diese Manipulatoren werden als Schaltflächen in der Werkzeugleiste des dokumentbasierten, grafischen Hauptwerkzeugs oder in der Werkzeugleiste eines dialogbasierten Subwerkzeugs angezeigt.

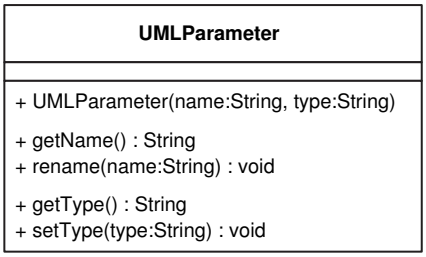

Abbildung 6.13: Die Operationen der Materialklasse `UMLParameter`

Ausser den bereits im Rahmenwerk vorhandenen Manipulatoren zum Selektieren und Verschieben von grafischen Elementen werden keine weiteren aktiven Manipulatoren benötigt.

7 Bewertung und Ausblick

7.1 Bewertung

Das Ziel der vorliegenden Arbeit ist, eine einheitliche Werkzeugarchitektur sowohl für dialogbasierte, als auch für dokumentbasierte Werkzeuge zu finden. Diese Werkzeugarchitektur soll es erlauben, ein Werkzeug um Funktionalität zu erweitern, ohne viele Änderungen am bestehenden Programm-Code der Anwendung vornehmen zu müssen (vgl. Abschnitt 1.4).

In Kapitel 2 wurden die Anforderungen aufgestellt, die die neue Werkzeugarchitektur erfüllen soll.

In den Kapiteln 3 und 4 wurden bestehende Architekurmuster, Application Frameworks, sowie GUI Toolkits auf Ideen hin untersucht, wie diese Anforderungen erfüllt werden können.

Schließlich wurde in Kapitel 5 ein Konzept des Manipulator-Musters erstellt, welches dann in Kapitel 6 umgesetzt wurde, indem ein Werkzeug mit Hilfe dieses Musters konstruiert wurde.

Im Folgenden wird das Manipulator-Muster nach den gleichen Kriterien bewertet werden, wie die im Kapitel 3 diskutierten Architekturmuster.

7.1.1 Trennung von Funktion und Interaktion

In einem nach dem Manipulator-Muster entwickelten Werkzeug bilden die von den Manipulatoren implementierten Funktionalitäten die Funktion des Werkzeugs.

Manipulatoren benachrichtigen ihre Umgebung über Änderungen am Materialzustand (vgl. Abschnitt 5.5). Die Interaktion eines Werkzeugs reagiert auf solche Ereignisse, indem sie die Präsentation dem neuen Materialzustand anpasst. Damit sind Manipulatoren zunächst unabhängig von der Interaktion.

Allerdings benötigen aktive Manipulatoren meist Zugriff auf die Interaktion des Werkzeugs. Hier gilt zwar auch, dass Manipulatoren Materialien modifizieren und die Interaktion über einen Reaktionsmechanismus auf diese Änderungen reagiert, aber damit aktive Manipulatoren wissen, wann sie ein Material manipulieren sollen, müssen sie die Interaktion beobachten.

Funktionalitäten wie das Verschieben und Ablegen von Materialien „wissen“ etwas über das Benutzungsmodell des Werkzeugs. Sie wissen z. B., dass ein Material vom Benutzer

verschoben und an einem anderen Ort wieder abgelegt werden kann. Solches Wissen ist typisch für Funktionalitäten innerhalb eines Werkzeugs. Funktionalitäten ohne Wissen über das Benutzungsmodell befinden sich meist im Domain Model ausserhalb eines Werkzeugs. Durch das Wissen der Werkzeug-Funktionalität über das Benutzungsmodell ist die Interaktion von der Werkzeug-Funktionalität nicht so stark getrennt wie von der im Domain Model enthaltenen Funktionalität. Deshalb erhöht sich die Kopplung zwischen Interaktion und Werkzeug-Funktionalität nicht, wenn ein Manipulator die Interaktion unter einer abstrakten Schnittstelle kennt, die den Teil des Benutzungsmodells repräsentiert, die der Manipulator sowieso kennt.

Wie das in Kapitel 6 erstellte Werkzeug zeigt, müssen Manipulatoren die Interaktion tatsächlich nicht unter ihrem konkreten Typ kennen, so dass Manipulatoren unabhängig vom GUI-Toolkit implementiert werden können und eine Trennung von Funktion und Interaktion gegeben ist.

Das Beispiel-Werkzeug lässt auch vermuten, dass in der Regel deutlich weniger aktive als passive Manipulatoren benötigt werden. Das UML-Werkzeug kommt mit nur zwei aktiven Manipulatoren (Selektieren und Verschieben von Materialien) aus, während es trotz des geringen Funktionsumfangs bereits zehn passive Manipulatoren besitzt.

7.1.2 Modularisierung der Funktionalitäten

Ein Manipulator repräsentiert genau eine Funktionalität. Wenn alle Funktionalitäten eines Werkzeugs durch Manipulatoren implementiert werden, ist das Ziel der Modularisierung der Funktionalitäten erreicht.

7.1.3 Dynamischer Aufbau

Das in der vorliegenden Arbeit konzipierte Architekturmuster beschreibt keinen expliziten Weg, wie ein Werkzeug dynamisch aus Manipulatoren aufgebaut wird. Das Muster wurde allerdings mit dem Ziel, einem Werkzeug Manipulatoren dynamisch zur Laufzeit hinzufügen zu können, entworfen. Dass ein dynamischer Aufbau möglich ist, zeigt die in Kapitel 6 beschriebene Umsetzung des Manipulator-Musters anhand eines UML-Werkzeugs.

7.1.4 Universalität und Skalierbarkeit

Universalität und Skalierbarkeit von Werkzeugarchitekturen sind oft nur aufgrund von Erfahrungen mit vielen Implementationen unterschiedlicher Werkzeuge zu beurteilen. Das in Kapitel 6 vorgestellte UML-Werkzeug ist das bisher einzige Werkzeug, das nach dem Manipulator-Muster entwickelt wurde. Daher ist die folgende Bewertung dieses Kriteriums lediglich eine begründete Vermutung.

Das UML-Werkzeug ist im Wesentlichen ein grafisches dokumentbasiertes Werkzeug. Es enthält allerdings auch ein dialogbasiertes Subwerkzeug. Das Manipulator-Muster ist also für beide Werkzeugarten anwendbar. Den größeren Nutzen scheint allerdings der do-

kumentbasierte Teil des Werkzeugs zu haben. Für dialogbasierte Werkzeuge gibt es andere, ähnlich gut geeignete Architekturen. Der Vorteil der Verwendung von Manipulatoren liegt hier hauptsächlich in der Vereinheitlichung der Architekturen sowie in der Plugin-Fähigkeit.

Der Aufwand, eine Funktionalität in Form eines Manipulators zum UML-Werkzeug hinzuzufügen ist unabhängig davon, wieviele Funktionalitäten das Werkzeug bereits besitzt. Das Manipulator-Muster scheint also gut für große Werkzeuge zu skalieren.

7.1.5 Umsetzbarkeit

Auch für diese Anforderung gilt, dass eine Bewertung aufgrund von Erfahrungen mit vielen Implementationen unterschiedlicher Werkzeuge erfolgen sollte. Die folgende Bewertung kann sich allerdings wieder nur auf die Erfahrungen mit dem UML-Werkzeug stützen.

Das UML-Werkzeug ist recht einfach aufgebaut. Es verwendet das ebenfalls in Kapitel 6 erstellte Manipulator-Rahmenwerk. Dieses Rahmenwerk versteckt keine dem Manipulator-Muster inhärente Komplexität, sondern bietet einheitliche Schnittstellen und Klassen zur Wiederverwendung an, die ansonsten in jedem Werkzeug neu geschrieben werden müssten.

Der größte Aufwand bei der Implementation des Werkzeugs lag in der Implementation der Benutzungsoberfläche sowie der Funktionalitäten. Dieser Aufwand ist unabhängig vom verwendeten Architekturmuster. Der zusätzliche Aufwand, den das Manipulator-Muster mit sich bringt, scheint also relativ klein zu sein.

Die Anforderung der Umsetzbarkeit scheint das Manipulator-Muster zu erfüllen, da es wenig komplex ist und gleichzeitig relativ zum Funktionsumfang eines Werkzeugs wenig zusätzlichen Implementierungsaufwand erfordert.

7.1.6 Zusammenfassung

Tabelle 7.1 zeigt eine Zusammenfassung der Bewertung des Manipulator-Musters.

7.2 Ausblick

Die in den vorangegangenen Kapiteln erarbeitete Werkzeug-Architektur ist nicht vollständig. Bei der Konzeption und Umsetzung der Architektur wurde versucht, den Detaillierungsgrad für die Beschreibung der Architektur auf einem Niveau zu halten, dass die wesentlichen Merkmale, die diese Architektur ausmachen, nicht aus dem Auge verloren werden.

Die folgenden Abschnitte beschreiben, an welchen Stellen die Manipulator-Architektur nach Ansicht des Autors verfeinert werden kann.

Anforderung		Manipulator-Muster
Trennung von Funktion und Interaktion	+	Funktion befindet sich in Manipulatoren
Modularisierung der Funktionalitäten	+	Ziel des Musters
	+	Jede Funktionalität wird durch einen Manipulator repräsentiert
Dynamischer Aufbau	+	Modularisierung der Funktionalitäten
	+	praktische Umsetzung im UML-Werkzeug
Universalität & Skalierbarkeit	?	wenig Erfahrung vorhanden
	+	Manipulatoren können sowohl für dokumentbasierte als auch für dialogbasierte Werkzeuge verwendet werden
	+	Aufwand scheint proportional zur Anzahl der Funktionalitäten zu sein
Umsetzbarkeit	?	wenig Erfahrung vorhanden
	+	Zusatzaufwand scheint gering zu sein
	+	Komplexität scheint gering zu sein

Tabelle 7.1: Zusammenfassung der Bewertung des Manipulator-Musters

7.2.1 Standardisiertes Plugin-Konzept

In Abschnitt 7.1.3 wurde bereits gesagt, dass das in der vorliegenden Arbeit vorgestellte Architekturmuster zwar einen dynamischen Aufbau eines Werkzeugs aus Manipulatoren erlaubt, dies aber nicht explizit unterstützt. Wünschenswert wäre eine Unterstützung von Plugin-Manipulatoren in einem wiederverwendbaren Rahmenwerk.

7.2.2 Manipulator-Repräsentation

Passive Manipulatoren besitzen ein Repräsentations-Objekt (Typ `ManipulatorRepresentation`), das Informationen darüber enthält, wie ein Manipulator an der Benutzungsschnittstelle dargestellt werden soll. Die vorliegende Arbeit hat bei der Beschreibung dieser Repräsentations-Objekte einige vereinfachende Annahmen getroffen.

Konfiguration von Texten und Symbolen

Zum einen wurde das Thema Internationalisierung nicht behandelt. Die Repräsentations-Objekte müssen für den Einsatz in unterschiedlichen Kulturen mit unterschiedlichen Sprachen unterschiedliche Texte enthalten. Wie Manipulator-Repräsentationen mit diesen verschiedenen Übersetzungen konfiguriert werden können, ist nicht das Thema dieser Arbeit, muss aber in vielen Projekten in der Praxis bedacht werden.

Zum anderen wurde davon ausgegangen, dass eine Funktionalität immer durch die gleichen grafischen Symbole dargestellt wird. Manipulatoren sollen unabhängig vom GUI-

Toolkit eingesetzt werden können. Unter anderem ermöglicht diese Unabhängigkeit die Portierung einer Anwendung auf unterschiedliche Toolkits. Unterschiedliche Toolkits stehen aber oft auch für unterschiedliche Bedienungsphilosophien. Sogenannte *Look-And-Feel*-Richtlinien beschreiben z. B. oft den Stil, mit dem die Symbole für eine Plattform gezeichnet werden sollten. So sehen beispielsweise die Schaltflächen, die das Öffnen und Speichern von Dokumenten symbolisieren, in Anwendungen für Microsoft Windows meist anders aus als in Anwendungen für den Apple Macintosh. Die in der vorliegenden Arbeit entwickelte Manipulator-Architektur verhindert nicht den Einsatz unterschiedlicher Symbole für unterschiedliche Plattformen, bietet allerdings auch keine explizite Unterstützung zur Lösung dieses Problems.

Layout der Benutzungsoberfläche

Manipulator-Repräsentationen enthalten zwar Informationen darüber, wie ein Manipulator dargestellt werden soll, aber nicht wo. Ein Werkzeug, dem ein passiver Manipulator hinzugefügt wird, kann anhand des Manipulators nicht entscheiden, an welcher Stelle der Benutzungsoberfläche die grafische Repräsentation des Manipulators angezeigt werden soll. Das Beispiel-Werkzeug aus Kapitel 6 fügt passive Manipulatoren in seine Werkzeugleiste ein. Dabei werden die Manipulator-Symbole in der Reihenfolge, in der die Manipulatoren dem Werkzeug hinzugefügt werden, angezeigt. Gruppierungen von Funktionalitäten durch Einfügen von Leerräumen in der Werkzeugleiste finden nicht statt.

Passive Manipulatoren sollten Informationen darüber enthalten, wo sie auf der Benutzungsoberfläche dargestellt werden sollen. Weiterhin sollten sie zu Funktionalitätsgruppen zusammengefasst werden können.

Literaturverzeichnis

[Ale01] ALEXANDRESCU, ANDREI: *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, Boston, MA, 2001.

[Bec99] BECK, KENT: *Extreme programming explained: embrace change*. Addison Wesley Longman, Inc., Reading, MA, 1999.

[Bec03] BECK, KENT: *Test-driven development: by example*. Addison-Wesley, Boston, MA, 2003.

[Bee01] BEEGER, ROBERT F.: *Einbettung von Oberflächen bei der Werkzeugkonstruktion*. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 2001.

[BG04] BECK, KENT und ERICH GAMMA: *Contributing to Eclipse: principles, patterns, and plug-ins*. Addison-Wesley, Boston, MA, 2004.

[Blo01] BLOCH, JOSHUA: *Effective Java. Programming Language Guide*. Addison-Wesley, Boston, MA, 2001.

[BMR+98] BUSCHMANN, FRANK, REGINE MEUNIER, HANS ROHNERT, PETER SOMMERLAD und MICHAEL STAL: *Pattern-orientierte Softwarearchitektur: Ein Pattern System*. Addison-Wesley (Deutschland) GmbH, Bonn, 1998.

[BOO] *Boost C++ Libraries*. [Online] `http://www.boost.org/`.

[CE00] CZARNECKI, KRZYSZTOF und ULRICH W. EISENECKER: *Generative programming: methods, tools, and applications*. Addison-Wesley, Boston, MA, 2000.

[Coc97] COCKBURN, ALISTAIR: *Surviving object-oriented projects: a manager's guide*. Addison-Wesley, Boston, MA, 1997.

[Cou87] COUTAZ, JOËLLE: *PAC, an Implementation Model for Dialog Design*. In: BULLINGER, H.-J und B. SHACKLE (Herausgeber): *Human-Computer-Interaction – INTERACT '87 proceedings*, Seiten 431–436. Elsevier Science Publishers B.V. (North Holland), September 1987.

[Cro85] CROWLEY, J.: *Navigation for an Intelligent Mobile Robot*. In: *IEEE Journal of Robotics and Automation, Bd. RA-1, Nr. 1*, Seiten 31–41, März 1985.

[Dea00] DEACON, JOHN: *Model-View-Controller (MVC) Architecture*, 2000. [Online] `http://www.jdl.co.uk/briefings/MVC.pdf`.

[ECL] *The Eclipse Project*. [Online] `http://www.eclipse.org/`.

[ELW98] ECKSTEIN, ROBERT, MARC LOY und DAVE WOOD: *Java Swing*. O'Reilly & Associates, Inc., Sebastopol, CA, 1998.

[Fow99] FOWLER, MARTIN: *Refactoring: improving the design of existing code*. Addison Wesley Longman, Inc., Reading, MA, 1999.

[Fow03] FOWLER, MARTIN: *Patterns of enterprise application architecture*. Addison-Wesley, Boston, MA, 2003.

[FS99] FOWLER, MARTIN und KENDALL SCOTT: *UML distilled: a brief guide to the standard object modelling language*. Addison-Wesley, Boston, MA, 2. Auflage, 1999.

[GHJV96] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley (Deutschland) GmbH, Bonn, 1996.

[GLL+90] GRYCZAN, GUIDO, CAROLA LILIENTHAL, MARTIN LIPPERT, STEFAN ROOCK, HENNING WOLF und HEINZ ZÜLLIGHOVEN: *Frameworkbasierte Anwendungsentwicklung (Teil 1)*. OBJEKTspektrum, Nr. 1:90–98, Januar/Februar 1990.

[GYJT97] GOSLING, JAMES, FRANK YELLIN und DAS JAVA-TEAM: *Java™ API. Band 2: Das Window Toolkit und Applets*. Addison-Wesley (Deutschland) GmbH, Bonn, 1997.

[IDE] *IntelliJ IDEA*. [Online] `http://www.intellij.com/idea/`.

[Int] *InterViews*.
[Online] `http://www.ivtools.org/ivtools/interviews.html`.

[ISO98] ISO/IEC 14822:1998(E): *Programming Languages — C++ (ISO and ANSI C++ standard)*, 1998.

[JWA] *JWAM: Java Framework for the Tools and Materials Approach*. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, [Online] `http://www.jwam.de/`.

[JWo] *Techdigm JWord*. [Online] `http://www.techdigm.com/`.

[KP88] KRASNER, GLENN und STEPHEN POPE: *A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80*. Journal of Object-Oriented Programming (JOOP), Seiten 26–49, August/September 1988.

[Lin02] LINK, JOHANNES: *Unit tests mit Java: Der Test-first-Ansatz*. dpunkt-Verlag, Heidelberg, 2002.

[Mar97] MARTIN, ROBERT C.: *Stability*, 1997.
[Online] `http://www.objectmentor.com/resources/articles/stability.pdf`.

[Mar03] MARTIN, ROBERT C.: *Agile software development: principles, patterns, and practices*. Prentice Hall, Inc., Upper Saddle River, NJ, 2003.

[Mey97] MEYER, BERTRAND: *Object-oriented software construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, 2. Auflage, 1997.

[Mod02] MODEL, MITCHELL L.: *Model View Controller History*, 2002. [Online] `http://c2.com/cgi/wiki?ModelViewControllerHistory`.

[Nor01] NORTHOVER, STEVE: *SWT: The Standard Widget Toolkit*, 2001. [Online] `http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html`.

[Ost03] OSTERMANN, BJÖRN: *Softwareaktionen Wiederholen und Rückgängigmachen. Undo/Redo-Konzepte für interaktive Anwendungen am Beispiel des objektorientierten Rahmenwerks JWAM*. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 2003.

[RJB99] RUMBAUGH, JAMES, IVAR JACOBSEN und GRADY BOOCH: *The unified modeling language manual*. Addison-Wesley, Boston, MA, 1999.

[RP97] RECHENBERG, PETER und GUSTAV POMBERGER: *Informatik Handbuch*. Carl Hanser Verlag, München, Wien, 1997.

[RW96] ROOCK, STEFAN und HENNING WOLF: *Konzeption und Implementierung eines „Reaktionsmusters" für objektorientierte Softwaresysteme*. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1996.

[ST93] STADTHERR, H. und C. TRAVING: *Building a Traffic Management System with C++*. Proceedings of the C++ User Group Technical Conference, München, 1993.

[Uni] *Unidraw*.
[Online] `http://www.ivtools.org/ivtools/unidrawinfo.html`.

[Val] *Valuatum*. [Online] `http://www.valuatum.com/`.

[Van03] VAN EMMENIS, ADRIAN: *Using the Eclipse GUI outside the Eclipse Workbench, Part 1: Using JFace and SWT in stand-alone mode*, 2003. [Online] `ftp://www6.software.ibm.com/software/developer/library/os-ecgui1.pdf`.

[Zül98] ZÜLLIGHOVEN, HEINZ: *Das objektorientierte Konstruktionshandbuch*. dpunkt-Verlag, Heidelberg, 1998.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbstständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Hamburg, 17.02.2004

Simon Ditrich