Studienarbeit

Einbettung von Oberflächen bei der Werkzeugkomposition

Juni 2001

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Robert F. Beeger
Schottweg 14
22087 Hamburg

Matrikelnummer : 4824555

Betreuer : Heinz Züllighoven
Reviewer : Stefan Roock

"The whole is more than the sum of its parts"

Aristoteles, zitiert in [Weis 98]

"We often think that when we have completed our study of one we know all about two, because 'two' is 'one and one'. We forget that we have still to make a study of 'and'."

A. S. Eddington, zitiert in [Blish 00]

Inhalt

Abbildungen

Danksagung

Mein Dank geht zuerst an Stefan Roock für das intensive Reviewing. Heinz Züllighoven danke ich für wertvolle Hinweise, die diese Arbeit an einer wichtigen Stelle in die richtige Richtung bewegten. Holger Breitling danke ich für das rege Interesse und die Geduld beim gemeinsamen Integrationsmarathon.
Hans Hagen und den anderen Mitgliedern der Mailingliste ntg-context@ntg.nl danke ich für die wertvollen Tipps zum Umgang mit ConTEXt.
Und nicht zuletzt danke ich meinen Eltern für den in mich gesetzten Glauben und ihre Unterstützung.

1 Einleitung

1.1 Motivation

Im Dezember 1999 kam mir der Gedanke zu einem Studienarbeitsthema, dessen Bearbeitung interessant zu werden versprach. Außerdem könnte es das Entwickeln von Werkzeugen mit dem JWAM-Rahmenwerk um einiges vereinfachen. Damals war die Version 1.4 die aktuelle Ausgabe von JWAM.
Von komponentenorientierter Werkzeugkonstruktion, wie sie in [Fröse 99] beschrieben und vom Autor jener Diplomarbeit im „Themen-der-Softwaretechnik-Seminar“ vorgestellt wurde, war damals in JWAM nichts zu sehen.
Zu dieser Zeit schrieben zwei andere Studenten – Johanna Felski und Erdal Özkan – an ihrer Studienarbeit ([Felski 00]), deren Ergebnisse in JWAM 1.5 eingingen. JWAM 1.5 brachte die ersehnte komponentenorientierte Werkzeugkonstruktion, indem sie die in [Fröse 99] beschriebenen Konzepte weitgehend umsetzte und durch einige eigene Konzepte ergänzte.
Leider kümmerten sich die Autoren von [Felski 00] nicht um die Präsentationsseite. Es wurde sogar schwieriger Subwerkzeuge in ihre Kontextwerkzeuge einzubetten. Bot JWAM 1.4 hier noch einige – wenn auch unelegante – Möglichkeiten, so waren diese in JWAM 1.5 nicht mehr vorhanden, was dazu führte dass ich mein aktuelles Projekt – eine Beispielanwendung, die fast alle möglichen Features JWAMs zeigen und als Begleitbeispiel der in Arbeit befindlichen englischen Version von [Züllighoven 98] dienen soll – nicht auf der aktuellen JWAM-Version weiterentwickeln konnte.

1.2 Zielsetzung

Ich werde mich in dieser Studienarbeit mit der Frage beschäftigen, ob und wie eine Ausweitung der komponentenorientierten Werkzeugkonstruktion auf den Bereich machbar ist, in dem die Präsentation eines Werkzeuges modelliert wird. Ich werde zunächst zeigen, dass das Problem der Einbettung von Oberflächen bei der Werkzeugkonstruktion wirklich ein Problem ist. Daraufhin werde ich, nach einer abwägenden Betrachtung einiger bereits existierender Ansätze, versuchen, ein zu JWAM passendes Konzept zu entwickeln und dessen Umsetzbarkeit durch eine Realisierung belegen.

1.3 Begriffserklärung

Im folgenden werde ich einige im Verlauf dieser Arbeit benutzte Begriffe erklären beziehungsweise Definitionen für diese Begriffe angeben.

1.3.1 Komponenten und Werkzeugkomponenten

Ich werde in dieser Arbeit oft die Begriffe **Komponente** und **Werkzeugkomponente** benutzen. Da zumindest der Begriff der Komponente in der Fachliteratur häufig in verschiedenen Bedeutungen gebraucht wird, scheint eine Abgrenzung dieses Begriffes und seiner Bedeutung in dieser Arbeit angebracht.
Der Begriff „Komponente“ wird in dieser Arbeit der Definition in [Felski 00] folgend gebraucht. Felski und Özkan legen dort fest

> „Eine Komponente ist grobkörniger als eine Klasse. Sie kapselt mehrere kooperierende Klassen, so dass auf deren Schnittstelle nicht mehr direkt, sondern über eine einheitliche Schnittstelle zugegriffen wird. Über diese wohldefinierte Schnittstelle bietet eine Komponente ihre Dienste an und nimmt Dienste anderer Komponenten in Anspruch.“

Eine Komponente ist also hier, wie auch in [Felski 00], nicht unbedingt ein einzeln in binärer Form vertreibbares Stück Software wie dies etwa von Szyperski (siehe [Szyperski 98]) verlangt wird.

Wenn ich in dieser Arbeit von Komponenten spreche, meine ich oft Werkzeugkomponenten. Werkzeugkomponeneten passen sehr gut zur Definition von Felski und Özkan. Werkzeugkomponenten können in ihrer Größe variieren, kapseln aber, unabhängig von ihrer Größe, mehrere kooperierende Klassen unter einer einheitlichen Schnittstelle. Es spricht nichts dagegen, Werkzeugkomponenten zu konstruieren, die der Definition Szyperskis genügen. Es wird aber oft vorkommen, dass Werkzeugkomponenten mit anderen fachlich dazugehörenden Teilen in einer größeren Komponenten ausgeliefert werden, weshalb sie also nicht immer der in [Szyperski98] vorgelegten Definition genügen müssen.

1.3.2 WAM

WAM ist die Abkürzung für **Werkzeug**, **Automat** und **Material**, drei sehr wichtigen Begriffen des **Werkzeug-und-Material-Ansatzes**.
Die grundsätzliche Idee hinter WAM ist es, Interviews mit den späteren Anwendern zu führen, deren Fachsprache soweit wie für das Projekt nötig zu lernen und sich dann mit den späteren Anwendern über das zu schaffende Anwendungssystem in der Sprache der Anwender zu unterhalten. Dabei werden Materialien, also die Dinge, die bearbeitet werden, und Werkzeuge, mit denen die Materialien bearbeitet werden, identifiziert.
Beim Entwickeln der Software wird dann versucht, die in den Interviews entdeckten Dinge zu modellieren und in dieser Software neu zu erschaffen, damit die späteren Anwender in dieser die Gegenstände wiederfinden, mit denen sie auch vorher schon gearbeitet haben.
Züllighoven und andere beschreiben dies in der nötigen Tiefe und Breite in [Züllighoven 98]. Im Zuge dieser Arbeit werde ich einige dort beschriebene Konzepte aufgreifen und näher beleuchten. Dies wird sich jedoch auf das hier dominante Thema, die Werkzeugkonstruktion, beschränken.

1.3.3 JWAM

JWAM ist ein Java-Rahmenwerk, das speziell auf die Entwicklung von Anwendungen nach WAM ausgerichtet ist. JWAM wird von der APCON Workplace Solutions, einem dem Arbeitsbereich Softwaretechnik des Fachbereichs Informatik der Universität Hamburg nahen Unternehmen, entwickelt. Nähere Informationen zu JWAM sind unter www.jwam.de zu finden.

1.3.4 Präsentation

Unter **Präsentation** verstehe ich im folgenden die Oberfläche eines Werkzeuges. Die Oberfläche eines Werkzeuges, auch häufig GUI[1] genannt, ist das, was ein Benutzer von einem Werkzeug sieht, womit er also arbeitet. Ich nenne die Oberfläche Präsentation, weil es der Teil des Werkzeuges ist, der dieses dem Benutzer präsentiert.

1.4 UML-Diagramme

In dieser Arbeit sind einige UML-Diagramme enthalten. Diese Diagramme sind zu der Notation, die in "Das UML Handbuch" (siehe [3Amigos 99]) beschrieben wird, konform. Diese Diagramme benutzen die verkürzte Notation, bei der die Attribute und Operationen nicht im Diagramm enthalten sind.
Eine Besonderheit ist hier, dass Klassen, die im JWAM enthalten sind, einen grauen Hintergrund haben, während alle anderen einen weißen haben.

[1] Graphical User Interface

1.5 Übersicht

In Kapitel 2 „Werkzeugkonstruktion nach WAM und mit JWAM 1.5“ werde ich einen Überblick über die Werkzeugkonstruktion nach WAM und deren Umsetzung in JWAM 1.5 geben. Ich werde hier näher auf die Einbettung von Subwerkzeugen eingehen und das Problem beleuchten, mit dem ich mich in dieser Arbeit beschäftigen werde.
In Kapitel 3 „Existierende Konzepte und Realisierungen“ werde ich einige existierende Konzepte, Realisierungen und die Möglichkeit eines Einsatzes in einem WAM-Kontext und in JWAM im Speziellen betrachten.
Kapitel 4 „Konzeption“ stellt meine Konzeption vor, mit der ich das mir gestellte Problem zu lösen gedenke.
Kapitel 5 „Realisierung“ geht auf interessante Details der Realisierung ein und begründet die Unterschiede zur Konzeption
In Kapitel 6 „Zusammenfassung und Ausblick“ wage ich abschließend einen Ausblick und stelle einige Visionen über die Zukunft der Werkzeugkonstruktion mit JWAM vor.

2 Werkzeugkonstruktion nach WAM und mit JWAM 1.5

In diesem Kapitel werde ich einen Teil des Werkzeug-und-Material-Ansatzes und dessen Umsetzung in JWAM 1.5 näher betrachten: die Werkzeugkonstruktion. In diesem Kapitel wird die Problemstellung dieser Arbeit konkretisiert.

2.1 Die Anatomie eines Werkzeuges nach WAM

Ein Werkzeug ist – wie bereits erwähnt – ein Gegenstand, mit dem wir einen anderen Gegenstand, das Material dieses Werkzeuges, bearbeiten. [Züllighoven 98] stellt die Anatomie eines Werkzeuges nach WAM vor. Diese ist in der aus [Züllighoven 98] entnommenen Abbildung 2.1 zu sehen.

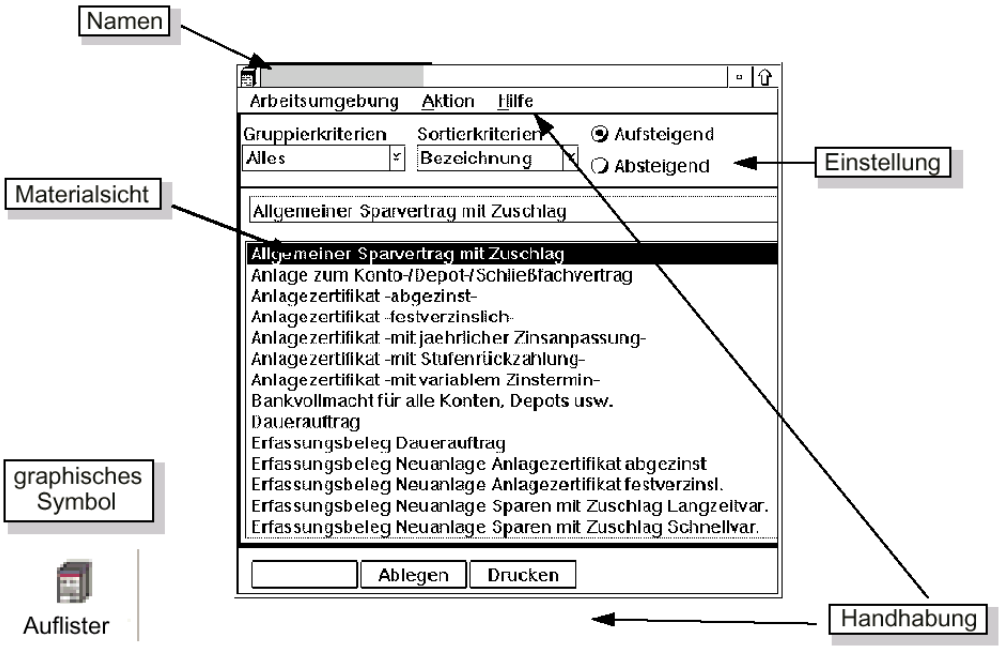

Abbildung 2.1 Anatomie eines Werkzeuges nach WAM (aus [Züllighoven 98])

Die Präsentation eines Werkzeuges hat also einen Bereich, in dem das Material selbst angezeigt wird. Auch hat sie einen Bereich, in dem Einstellungen vorgenommen werden können. In dem gezeigten Beispiel kann man dort zum Beispiel einstellen, wie die angezeigte Liste sortiert werden soll. Außerdem befinden sich oberhalb des Einstellungsbereichs und unterhalb des Materialanzeigebereichs zwei Bereiche, in denen komplexere Handhabungen beziehungsweise Aktionen durchgeführt werden können, die durch die ausschliessliche Verwendung des Materialbereiches nicht ausführbar sind. Dazu gehört hier zum Beispiel das Drucken des Materials. Der obere der beiden Handhabungsbereiche enthält Menüs und in neueren Anwendungssystemen finden hier auch Symbolleisten ihren Platz, die eine Auswahl der in den Menüs enthaltenen Aktionen enthalten. Der untere Bereich, der Knöpfe für wichtige und häufig ausgeführte Aktionen enthält, wird in neueren Anwendungen häufig durch die Symbolleisten ersetzt.

2.2 Werkzeugkonstruktion nach WAM

Ein Werkzeug hat also eine Präsentation über die ein Benutzer mit dem Werkzeug interagieren kann. Interaktion alleine führt aber nicht zur Erledigung einer Aufgabe. Dafür muss hinter der Oberfläche – der Präsentation – auch Funktionalität stecken. Sicherlich kann man diese beiden

Aspekte vermischen und in einer Klasse implementieren. Das führt aber bei größeren Anwendungen unweigerlich zu Unübersichtlichkeiten, daraus folgenden Fehlern und vielleicht zu einem interaktionsgetriebenen Design. Ein Design ist dann interaktionsgetrieben, wenn ein Entwickler zuerst die Präsentation entwickelt und sich dann überlegt, was für eine Funktionalität er hinter die einzelnen Elemente dieser Präsentation stellen soll. Das ist jedoch nicht zielführend, da bei einem Werkzeug die Funktionalität das wichtigste ist. Ein Werkzeug muss primär eine bestimmte Funktion erbringen – häufig die Bearbeitung eines bestimmten Typs von Material. Erst wenn diese geklärt ist, wird die Interaktion mit diesem Werkzeug wichtig, die dem Benutzer die Nutzung dieser Funktionalität ermöglicht.
Um ein interaktionsgetriebenes Design zu vermeiden und die einzelnen Aspekte, unter denen ein Werkzeug betrachtet werden kann, klar zu trennen, wird in [Züllighoven 98] das Entwurfsmuster **Trennung von Funktion und Interaktion** vorgestellt, dessen Anwendung darauf hinausläuft, dass zuerst die Funktion und erst dann die Interaktion in verschieden Klassen implementiert werden.
Die Autoren des "Objektorientiertes Konstruktionshandbuch" ([Züllighoven 98]) stellen nach dem eben genannten Muster ein weiteres vor, das die Interaktion noch weiter unterteilt. Dieses, **Trennung von Handhabung und Präsentation** genannte Muster, macht eine Unterscheidung zwischen der Präsentation, also dem was der Benutzer auf dem Bildschirm sieht, und der Handhabung, also der Art und Weise, wie er mit der Präsentation umgeht. So ist es zum Beispiel für das Werkzeug belanglos, ob der Benutzer auf einen Knopf klickt oder einen Menüeintrag auswählt. Bei beiden Vorgehensweisen wird etwas aktiviert. Es gibt viele andere Beispiele dieser Art. Auch beim Auswählen eines Elementes aus einer Menge ist es für das Funktionieren des Werkzeuges unerheblich, ob diese Auswahl in einer Liste oder einer Combobox passiert. Für das Werkzeug selbst ist die Handhabung wichtig, nicht die Präsentation. Das genannte Entwurfsmuster legt die Verwendung von Interaktionsformen (Wie gehe ich mit der Oberfläche um?) und den tatsächlichen Präsentationsformen (Was für Oberflächenelemente sehe ich? Knöpfe, Menüs, Eingabefelder) nahe. Wird in der Interaktionskomponente eines Werkzeuges, also in dem Teil des Werkzeuges, der die Interaktionen des Benutzers mit der Präsentation interpretiert, auf die Präsentation nur über deren Interaktionsformen zugegriffen, so wird diese, und damit das ganze Werkzeug, unabhängig von der Präsentation. Es kann sogar sein, dass die Präsentation von einem Softwareergonomen erstellt wird, während der Rest von einem Softwareentwickler implementiert wird. Das einzige, worüber sich diese beiden dann einigen müssen, ist, welche Interaktionsformen es geben soll.
Ein weiterer Vorteil dieser Herangehensweise wird deutlich, wenn man sich der Evolution von GUI-Toolkits bewusst wird. War im Java-Umfeld vor einigen Jahren das Advanced Windowing Toolkit (AWT) das einzige GUI-Toolkit, so wird heutzutage nur noch das Swing-Toolkit benutzt. Wird bei der Werkzeugkonstruktion das genannte Entwurfsmuster benutzt, dann ist es ein Leichtes, die Präsentation, die ein altes GUI-Toolkit verwendet, durch eine neue zu ersetzen, die ein neues GUI-Toolkit verwendet. Die Interaktionsformen ändern sich nicht. Nur die Präsentationsformen ändern sich. Die werden aber nur in der Klasse verwendet, die die Präsentation des Werkzeuges definiert.
Neben kleinen, überschaubaren Werkzeugen wird es auch immer komplexe, große und weniger überschaubare Werkzeuge geben. Verspricht ein Werkzeug groß und unüberschaubar zu werden, macht es Sinn, über eine Unterteilung dieses Werkzeuges in Subwerkzeuge nachzudenken. Dabei wird das Werkzeug im Ganzen betrachtet, und es wird versucht, einzelne Teilbereiche dieses Werkzeuges in kleinere Werkzeuge auszulagern, die dann als Subwerkzeuge des dann „Kontextwerkzeug“ genannten Werkzeuges fungieren und an die die Erledigung von Teilaufgaben delegiert wird. Diese Subwerkzeuge können in die Präsentation des Kontextwerkzeuges eingebettet werden oder in eigenen Fenstern laufen. Im ersten Fall merkt der Benutzer gar nicht, dass er hier ein Werkzeug benutzt, dass aus mehreren Werkzeugen zusammengebaut ist. [Züllighoven 98] stellt hierzu das Muster **Werkzeugkomposition** vor.

Ich werde in Abschnitt 2.4 näher auf dieses Thema eingehen und die Realisierung mit JWAM zeigen.

2.3 Entwicklung einfacher Werkzeuge mit JWAM

JWAM ist ein Java-Rahmenwerk zur Entwicklung von Anwendungssystemen, bei deren Entwicklung WAM angewandt wird. Als ein solches setzt es einen Großteil der in [Züllighoven 98] beschriebenen Konzepte um.
Die Struktur zweier mit JWAM konstruierter, einfacher Werkzeuge ist in Abbildung 2.2 als UML-Klassendiagramm zu sehen. Dieses Diagramm zeigt nicht die Beziehung der Werkzeuge zu ihren Materialien, da es in dieser Arbeit primär um die Präsentationsseite geht. Das Material wird aber natürlich immer von dem Teil des Werkzeuges benutzt werden, der für die Funktionalität des Werkzeuges zuständig ist.

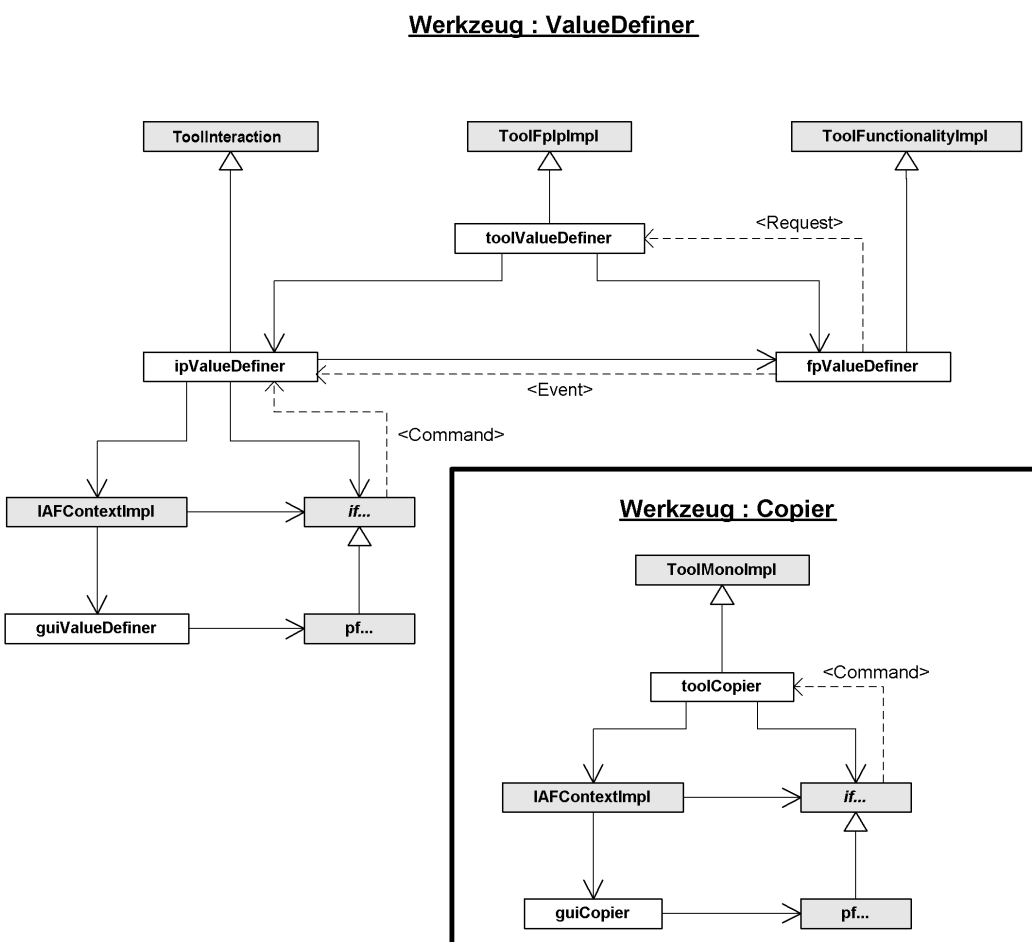

Abbildung 2.2 Zwei einfache mit JWAM konstruierte Werkzeuge

Abbildung 2.2 zeigt zwei Varianten der Werkzeugkonstruktion mit JWAM in der Version 1.5. Das UML-Diagramm zum Werkzeug Copier stellt die einfachere der beiden vor. Hier wird das Muster **Trennung von Handhabung und Präsentation** verwendet aber nicht das Muster **Trennung von Funktion und Interaktion**.

Das Werkzeug Copier kopiert ein Material des Typs `CopyAble`. Die Präsentation dieses Werkzeuges bietet neben der Möglichkeit den Kopiervorgang zu starten auch die, auszuwählen, wie oft das Material kopiert werden soll
Dieses Werkzeug hat eine von der JWAM-Klasse `ToolMonoImpl` erbende Klasse `toolCopier`. Das Tool-Objekt – ein Exemplar der entsprechenden Tool-Klasse – eines Werkzeuges repräsentiert dieses nach außen und bietet Zugriff auf alle wichtigen Schnittstellen dieses Werkzeuges. Im Fall des Werkzeuges Copier ist das Tool-Objekt nicht nur Repräsentant. Es kümmert sich auch um die Interaktion mit dem Benutzer und deren funktioneller Interpretation. Um das Muster **Trennung von Handhabung und Präsentation** umzusetzen, wird ein Objekt der Klasse `IAFContextImpl` benutzt. Diesem wird beim Starten ein Objekt der speziell für dieses Werkzeug entwickelten Klasse `guiCopier` übergeben, in dem es die angeforderten Interaktionsformen heraussuchen kann. Interaktionsformen sind benannte Schnittstellen – Interfaces –, die von Präsentationsform-Klassen implementiert werden, die gleichzeitig von für das jeweilige GUI-Toolkit spezifischen Widget-Klassen[2] erben. Das `IAFContextImpl`-Exemplar sucht alle zu einer Interaktionsform-Schnittstelle passenden Widgets heraus und findet über den Interaktionsformnamen das passende. Mehr Informationen zum Thema Interaktionsformen sind auch in [Bleek 99], [Lippert 97] und [Sturm 98] zu finden.
Das Tool-Objekt meldet Objekte der Klasse `cmdObject` an den Interaktionsformen an, deren Ausführung bestimmte Interaktionen des Benutzers signalisiert (zum Beispiel das Aktivieren eines Aktivators). Das Command-Konzept in JWAM ist eine Umsetzung des Entwurfsmusters **Command** (siehe dazu [Gamma 98]).
Diese Variante, die Repräsentation, Funktion und Interaktion vermengt, ist für einfache Werkzeuge ohne oder aber nur mit wenigen überschaubaren eigenen Werkzeugzuständen gut verwendbar.
Wird ein Werkzeug komplexer oder soll es auch einen eigenen Werkzeugzustand haben, ist die Umsetzung des Musters **Trennung von Funktion und Interaktion** sinnvoll. Ein Beispiel hierfür ist das in Abbildun 2.2 als UML-Diagramm gezeigte Werkzeug ValueDefiner, dass dazu dient die Wertemengen zweier im EMS enthaltener Fachwertklassen durch den Benutzer festlegen zu lassen. Hier kommt im Vergleich zum Werkzeug Copier eine Subklasse der JWAM-Klasse `ToolInteraction` und eine Subklasse der JWAM-Klasse `ToolFunctionalityImpl` dazu. Die Klasse `fpValueDefiner`[3] implementiert die Funktionalität des Werkzeuges, während die Klasse `ipValueDefiner`[4] die Interaktion des Werkzeuges implementiert. `ipValueDefiner` kennt und benutzt `fpValueDefiner`, so dass ein Objekt dieser Klasse die interpretierten Eingaben des Benutzers in Form von Befehlen an die Funktionalität weiterleiten kann. Ein Objekt der Klasse `ipValueDefiner` meldet sich an dem entsprechenden Objekt von `fpValueDefiner` für Ereignisse an, die für es interessant sind.
Die Funktionalität eines Werkzeuges gibt über Ereignisse bekannt, wenn sich der Zustand des Werkzeuges oder des aktuell bearbeiteten Materials ändert. Die Funktionalität kennt in dieser Variante der Werkzeugkonstruktion die Interaktion des Werkzeuges nicht. Diese ist auch für die Funktionalität irrelevant. Das einzig wichtige ist, dass die Funktionalität Anweisungen bekommt, die sie ausführt, und Ereignisse bekannt gibt, wenn diese Anweisungen eine Änderung des Zustandes hervorrufen.
Kann die Funktionalität eine Anweisung nicht ausführen, leitet sie diese an die ihr übergeordnete Instanz weiter. Eine solche Anweisung, die nicht von der Funktionalität des Werkzeuges ausgeführt werden kann, ist zum Beispiel die Anweisung gerade dieses Werkzeug zu schließen. Ein Werkzeug kann sich nicht alleine schließen. Das Schließen eines Werkzeuges muss von einer höheren Instanz

[2] „Widget“ ist ein Kunstwort und ist eine Neuschöpfung aus den beiden Wörtern „Window“ und „Gadget“. Ein „Widget“ ist also ein „Fenster-Dings“, etwas, das in einem Fenster zu sehen ist. Die Dinge, die man in Fenstern – den Fenstern auf einem Bildschirm – sieht, sind Knöpfe, Eingabefelder, Optionsfelder und ähnliches. All diese Dinge sind Widgets.
[3] fp steht für „functional part“, was sich ins Deutsche mit „Funktionskomponente“ übersetzen lässt
[4] ip steht für „interaction part“, was sich ins Deutsche mit „Interaktionskomponente“ übersetzen lässt

durchgeführt werden. Im Normalfall ist es die Umgebung, die solche Anweisungen für Werkzeuge ausführt. Wenn ein Werkzeug als Subwerkzeug von einem anderen Werkzeug verwendet wird, ist dass Werkzeug, dass das andere als Subwerkzeug verwendet, die höhere Instanz für dieses. Mit dem Thema „Subwerkzeuge“ befasst sich der nächste Abschnitt näher.
Das Ereigniskonzept in JWAM ist eine Umsetzung des Entwurfsmusters **Observer** und das Weiterleiten von Anweisung an eine höhere Instanz ist eine Umsetzung des Entwurfsmusters **Chain of Responsibility** (für beide siehe [Gamma 98])

2.4 Entwicklung komplexer Werkzeuge mit JWAM

In Abschnitt 2.3 habe ich zwei Varianten der einfachen Werkzeugkonstruktion vorgestellt.
Komplizierter wird es, wenn Werkzeuge selbst wieder Subwerkzeuge enthalten. Die Arbeit von Felski und Özkan ([Felski 00]) hat JWAM zwar um einiges in Richtung „Komponentenorientierte Werkzeugkonstruktion“, wie sie in [Fröse 99] von Frank Fröse beschrieben wird, weiter gebracht, aber es fehlt noch einiges, dass man wirklich von einer komponentenorientierten Werkzeugkonstruktion sprechen könnte.
Abbildung 2.3 zeigt ein UML-Diagramm eines solchen Werkzeuges, dass Subwerkzeuge enthält. Dieses Diagramm zeigt nicht alle Beziehungen auf, sondern nur die für die Komposition von Werkzeugen interessanten. Natürlich bestehen für jedes Werkzeug für sich betrachtet auch weiterhin die in Abbildung 2.2 aufgezeigten Beziehungen.

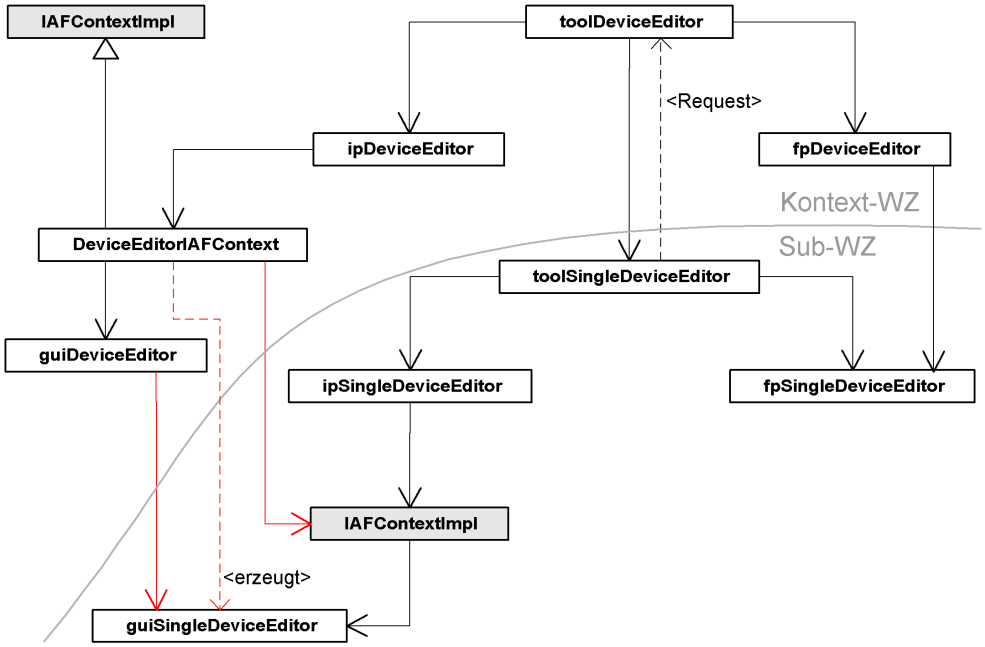

Abbildung 2.3 Komplexes mit JWAM konstruiertes Werkzeug

Dieses Werkzeug – Device-Editor – kann mehrere Materialien vom Typ `Device` bearbeiten, indem es für jedes Material ein Subwerkzeug Single-Device-Editor einbettet.
Werkzeuge dieser Art werden immer auf eine ähnliche Weise zusammengebaut. Das Kontext-Tool-Objekt kennt seine Sub-Tool-Objekte, und das Kontext-Funktionalität-Objekt kennt seine Sub-Funktionalität-Objekte. Diese Konstellation sollte sinnvoll erscheinen. Subwerkzeuge werden wegen ihrer Funktionalität, der Dienstleistung, die sie erbringen können, eingesetzt. Sie erweitern

die Funktionalität ihres Kontextwerkzeuges. Deshalb muss eine Kontext-Funktionalität ihre Sub-Funktionalitäten kennen und benutzen. Das Kontext-Tool-Objekt muss seine Sub-Tool-Objekte kennen, um deren Einbettung in die Wege leiten zu können und um Anweisungen, die von den Subwerkzeugen nicht erfüllt werden können, zu verarbeiten.
Nicht wirklich sinnvoll scheint aber zu sein, dass das Kontextwerkzeug die Klassen kennen muss, die die Präsentation der Subwerkzeuge definieren, und von diesen dann auch Exemplare erzeugt, um die Subwerkzeuge einzubetten. Dies ist aber genau der Fall, der in Abbildung 2.3 zu sehen ist. Der spezialisierte `IAFContext` `DeviceEditorIAFContext` weiß, dass er beim Einbetten eines neuen Subwerkzeuges ein Exemplar der Klasse `guiSingleDeviceEditor` erzeugen muss und dass er ein Exemplar der Klasse `IAFContextImpl` bereitstellen muss, über dass dann das Subwerkzeug seine Präsentation anspricht.
Und genau dieser Umstand verleitet mich dazu, zu behaupten, dass JWAM noch keine wirklich komponentenorientierte Werkzeugkonstruktion hat. Es hat keinen großen Nutzen, Werkzeugkomponenten zu benutzen, wenn ein nicht unerhebliches Wissen über den inneren Aufbau dieser Komponenten notwendig ist, um sie richtig einsetzen zu können.
Zielführender wäre hier ein automatisches, vom Rahmenwerk zu erledigendes, Einbetten der Subwerkzeuge, für dass kein detailliertes Wissen über den Aufbau der einzubettenden Werkzeugkomponente nötig ist.

2.5 Zusammenfassung

Ich habe in diesem Kapitel zunächst den Begriff des Werkzeuges geklärt und beschrieben, was ein Werkzeug nach WAM ausmacht. Ich bin dabei sowohl auf die „Anatomie“ eines Werkzeuges eingegangen (siehe dazu Abschnitt 2.1) als auch auf die Konstruktion von Werkzeugen nach WAM (siehe dazu Abschnitt 2.2). Bei dieser Beschreibung habe ich die zwei für die Werkzeugkonstruktion wichtigen Entwurfsmuster **Trennung von Funktion und Interaktion** und **Trennung von Handhabung und Präsentation** kurz beschrieben.
Nach dieser theoretischen Einführung habe ich zunächst in Abschnitt 2.3 anhand zweier Werkzeuge die Umsetzung der vorgestellten Entwurfsmuster vorgestellt.
Hiernach habe ich in Abschnitt 2.4 ein komplexeres Werkzeug vorgestellt, dessen Komplexität sich darin begründet, dass es ein anderes Werkzeug als sein Subwerkzeug benutzt. Hier habe ich schließlich das in der Einleitung (siehe Abschnitt 1.2) vorgestellte Problem anhand dieses Werkzeuges konkretisiert und bin zum Schluss gekommen, dass die Werkzeugkonstruktion in JWAM noch ein gutes Stück von der Komponentenorientierung entfernt ist, da beim Einsatz von Subwerkzeugen ein zu großes Wissen über diese Subwerkzeuge in ihren Kontextwerkzeugen vorhanden sein muss. Dieses Wissen zerstört die Komponentenkapsel der Subwerkzeuge, so dass sie nicht wirklich komponentenorientiert benutzt werden können.
Ich werde in dieser Arbeit versuchen, die Komponentenorientierung der Werkzeugkonstruktion ein weiteres Stück voranzutreiben.
Bevor ich jedoch dazu komme, werde ich im nächsten Kapitel bereits existierende Konzepte und Realisierungen aus diesem Bereich betrachten und abwägen, wie und ob sich diese im WAM-Kontext im Allgemeinen und für JWAM im Speziellen verwenden lassen.

3 Existierende Konzepte und Realisierungen

Im folgenden werde ich einige Konzepte und Realisierungen vorstellen, die das im vorigen Kapitel vorgestellte Problem der Einbettung von Subwerkzeugen oder zumindest einen Teil davon behandeln.
Dabei habe ich zwei Ansätze identifiziert, die das Problem von zwei verschiedenen Seiten angehen. Der eine Ansatz ist aufgabenorientiert und wird in Abschnitt 3.1 vorgestellt, während der andere dokumentenorientiert ist und in Abschnitt 3.2 vorgestellt wird.

3.1 Aufgabenorientierte Ansätze

Bei den aufgabenorientierten Ansätzen liegt das Hauptaugenmerk auf den Aufgaben, die erfüllt werden sollen.
Die aufgabenorientierten Ansätze führen weg von den großen, monolithischen Programmen. Hier werden Applikationen aus kleineren Applikationen zusammengebaut, die bestimmte Aufgaben erfüllen. Es ist eine Bewegung weg von omnipotenten Applikationen, die den Anspruch haben, jede Aufgabe erledigen zu können, hin zu kleinen spezialisierten Komponenten, die für eine bestimmte Aufgabe entwickelt werden und diese dann auch idealerweise zur Gänze erfüllen können. Entwickler können hier ihr Wissen in spezialisierte Komponenten einfließen lassen, die sie dann an Entwickler weiterverkaufen, die dieses spezielle Wissen nicht haben.
Wenn ein Entwickler nach der alten Art und Weise einen Terminplaner entwickeln will, muss er sich nicht nur mit dem Problem der Terminplanung befassen, sondern auch mit vielen anderen Dingen. Er wird zum Beispiel auch eine Textverarbeitungsfunktionalität anbieten müssen, mit der die Einträge im Terminkalendar editiert werden können.
Ein Entwickler, der aufgabenorientiert einen Terminplaner entwickelt, beschäftigt sich mit der Terminplanung selbst, denn in diesem Bereich kennt er sich aus. In der Entwicklung von Textverarbeitungen ist unser hypothetischer Entwickler kein Experte. Er wird sich deshalb eine Textverarbeitungskomponente von einem auf die Entwicklung von Textverarbeitungskomponenten spezialisierten Entwickler kaufen und diese in seinen Terminplaner integrieren.
Da er die Textverarbeitung nicht selbst entwickeln muss, braucht er auch weniger Zeit für die Entwicklung seines Terminplaners. Außerdem wählt er sich aus einer Auswahl aus Textverarbeitungskomponenten die beste aus und erreicht somit bei einer kürzeren Eintwicklungszeit eine größere Qualität des Produktes, als er selbst mit eigenen Entwicklungen erreichen kann.
Die aufgabenorientierten Ansätze führen bei einem genügend großen Pool an Komponenten tendenziell zu höherwertiger Software bei geringeren Entwicklungszeiten. Die größe des Komponentenpools ist wichtig bei den aufgabenorientierten Ansätzen. Ist der Pool klein, kann es leicht vorkommen, dass er für eine Aufgabe nur eine einzige Komponente anbietet. Wenn es für eine Aufgabe nur eine einzige Komponente gibt, die somit eine Monopolstellung inne hat, kann es passieren, dass diese Komponente sich der mangelnden Konkurenz wegen auf einem qualitativ niedrigen Niveau befindet. Qualitativ geringwertige Komponenten schmälern den Vorteil dieser Ansätze, da die Software, die diese Komponenten einsetzt, natürlich auch deren Fehler aufweist. Ein Nachteil ist dann auch, dass der Entwickler oft die Komponenten nicht verändern darf und sich so darauf verlassen muss, dass der Entwickler der Komponente in absehbarer Zeit eine neue fehlerbereinigte Version herausbringt.
Die aufgabenorientierten Ansätze leben also von einem lebhaften Komponentenpool, wie dies wohl bei allen Ansätzen der Fall ist, die in irgendeiner Form komponentenorientiert sind.

3.1.1 ActiveX-Steuerelemente

Zum Thema ActiveX zitiert [Schmitt 96] ein Aussage der ActiveX Group[5] : „ActiveX is a set of technologies from Microsoft that enables interactive content for the World Wide Web“.

[5] Die ActiveX Group ist eine von Microsoft aufgestellte Gruppe innerhalb der Open Group, die mit der Entwicklung und Pflege neuer Web-fähiger Technologien betraut ist.

Mit ActiveX sollen die Handhabungsmöglichkeiten, die mit den Programmen unter Microsoft Windows möglich sind, auch für das WWW erschlossen werden. Der Benutzer soll den Unterschied zwischen seinen lokal laufenden Windows-Programmen und den im WWW laufenden Programmen möglichst nicht mehr bemerken.
Eine der ActiveX-Technologien sind die ActiveX-Steuerelemente. ActiveX-Steuerelemente sind eine Weiterentwicklung der Visual-Basic-Steuerelemente, die sich zu ihrer Zeit einer großen Beliebtheit erfreuten. Zunächst waren diese Steuerelemente nur zur Ergänzung der Steuerelementpalette von Visual Basic gedacht. Das waren anfangs speziellere Oberflächenelemente wie zum Beispiel die Karteikarten-Steuerelemente, die jeder Windows-Benutzer mittlerweile aus etlichen Programmen kennt.
Diese Steuerelemente wurden dann komplexer, bis sie schließlich selbst zu kleinen spezialisierten Programmen wurden. Mit ActiveX-Steuerelementen werden diese spezialisierten Steuerelemente in mehr Kontexten einsetzbar – sogar auf Web-Pages ([Schmitt 97]).
Der Markt für ActiveX-Steuerelemente boomt. Es gibt mittlerweile alle möglichen Arten von ActiveX-Steuerelementen (zum Beispiel Kalender, Textverarbeitungen, Web-Browser und vieles mehr).
Da diese Komponenten als Steuerelemente konzipiert sind, können sie nicht selbstständig laufen. Sie haben keine Menüs und auch keine Symbolleisten. Ihnen fehlt also das Drumherum, dass einer Anwendung das typische Aussehen gibt.
ActiveX-Steuerelemente unterstützen neben dem normalen Laufzeitmodus auch einen Entwurfsmodus, in dem sie in einer Entwicklungsumgebung teilweise interaktiv, teilweise auch über sogenannte „Property-Pages“ angepasst werden können (siehe Abbildung 3.1).

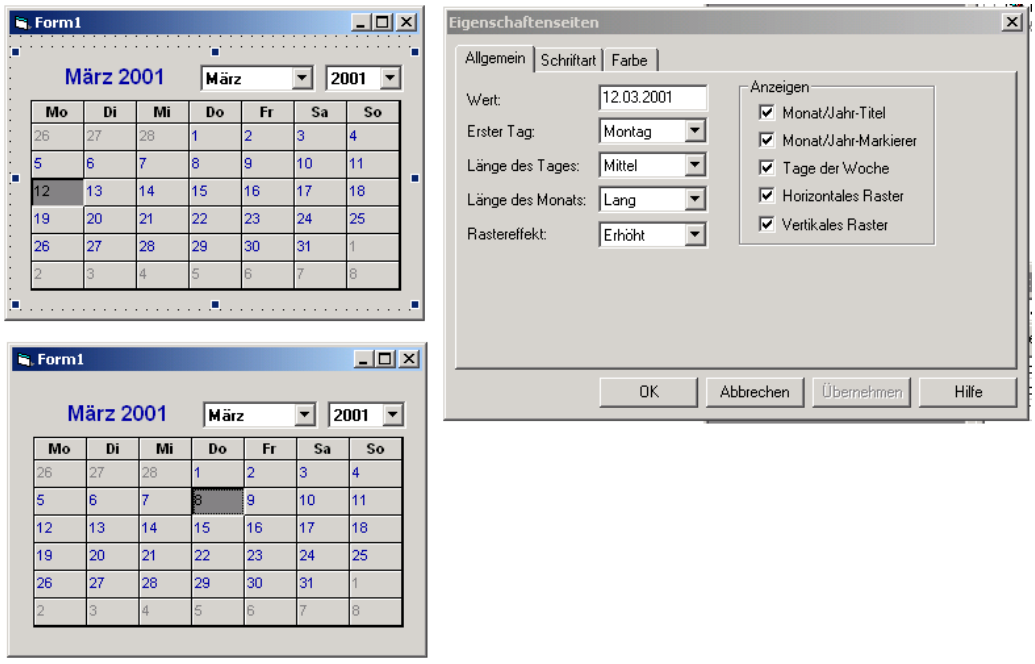

Abbildung 3.1 Ein ActiveX-Steuerelement im Entwurfsmodus (oben) und im Laufzeitmodus (unten)

ActiveX-Steuerelemente können in Anwendungen aber auch in Web-Pages eingebettet werden. Diese Web-Pages sind dann aber nur mit dem Microsoft Internet Explorer benutzbar.

3.1.2 Komponentenorientierte Werkzeugkonstruktion

Frank Fröse führt in seiner Diplomarbeit (siehe [Fröse 99]) den Begriff der **Werkzeugkomponenten** ein. Eine Werkzeugkomponente kapselt ein Werkzeug in einer Komponente und verschafft ihr so über den Begriff des Werkzeuges hinaus, wie er in [Züllighoven 98] definiert wurde, eine bessere Einsetzbarkeit als Subwerkzeug.
In [Fröse 99] werden einige standardisierte Schnittstellen vorgestellt, von denen einige im Bereich der Funktionalität und andere in dem der Interaktion und der Präsentation angesiedelt sind.
Ich werde mich hier auf die Diskussion der Schnittstellen der Interaktion und Präsentation beschränken und insbesondere die Einbettung von Subwerkzeugkomponenten in Kontextwerkzeugkomponenten betrachten.
Eine Werkzeugkomponente nach [Fröse 99] bietet zumindest eine `View`-Schnittstelle an. Diese Schnittstelle ist vom verwendeten GUI-Toolkit abhängig. So gibt es zum Beispiel eine `SwingView`-Schnittstelle, die für die Benutzung mit dem Swing-Toolkit ausgelegt ist. Diese Views können die tatsächliche Präsentation der Werkzeugkomponente zurückliefern, die dann von der die Werkzeugkomponente benutzenden Werkzeugkomponente eingebettet werden kann.
Fröse unterscheidet in seiner Diplomarbeit drei Arten von Werkzeugkomponenten:

- **ToolComponent** ist die einfachste Art von Werkzeugkomponente, die ihre Präsentation zur Einbettung in ein Kontextwerkzeug über ihre `View`-Schnittstelle zur Verfügung stellt.
- **CompositeToolComponent** erweitert ToolComponent um die Möglichkeit, andere Werkzeugkomponenten enthalten zu können
- **RootToolComponent** erweitert CompositeToolComponent um die Möglichkeit einen Abschluss für ein Werkzeug bilden zu können. Hier wird das Fenster erzeugt, in dem das Werkzeug laufen wird. Ein eigenständig lauffähiges Werkzeug wird immer ein RootToolComponent sein, das andere Werkzeugkomponenten enthält.

Außerdem bietet ein Werkzeug auch ausführbare Aktionen an, die von den dieses Werkzeug als Subwerkzeug einsetzenden Werkzeug genutzt werden können. Hervorzuheben ist hier, dass zur Realisierung dieser Aktionen eine die benannte Schnittstelle Action, die ein Bestandteil des Swing-Toolkits ist, implementierende Klasse verwendet wird. Obwohl das in [Fröse 99] vorgestellte Konzept ansonsten Toolkit-unabhängig ist, wird hier eine unnötige Abhängigkeit zum Swing-Toolkit hergestellt.

3.2 Dokumentenorientierte Ansätze

Bei den dokumentenorientierten Ansätzen ist eine Konzentration weg von den Applikationen auf die Dokumente selbst zu sehen. Die Applikationen treten in ihrer Wichtigkeit hinter den Dokumenten zurück. Sie sind ersetzbare Werkzeuge, mit denen die Dokumente erschaffen und bearbeitet werden.
Die dokumentenorientierten Ansätze führen auch weg von großen monolithischen Programmen, die jeden Aspekt und jedes Element eines Dokumentes beherrschen, hin zu kleinen spezialisierten Komponenten, die nur mit einem kleinen Teil eines Dokumentes umgehen können.
In diesem Zusammenhang sind die Begriffe „Verbunddokument“ und „Dokumentart“ sehr wichtig.
Verbunddokumente sind aus anderen Dokumenten zusammengesetzte Dokumente. Das Dokument, wie es sich als ganzes nach außen zeigt, ist ein „Container-Dokument“. Es enthält die Dokumente, aus denen es besteht.
Die Dokumente, aus denen das Container-Dokument besteht, können verschiedenster Art sein. Zur Zeit sind Texte, Vektorgraphiken, Bitmaps, Animationen, Viedeofilme, Tondokumente und anderes als Dokumentarten identifiziert.

Die Philosophie der dokumentenorientierten Ansätze geht dahin, dass es für jede dieser Dokumentarten einen oder mehrere spezialisierte Editoren und Anzeiger gibt.
Wie diese Editoren und Anzeiger in den Bearbeitungsprozess eines Dokumentes integriert werden, ist von Ansatz zu Ansatz verschieden.
Im folgenden sollen einige dieser Ansätze vorgestellt werden und unter dem Gesichtspunkt der Integration dieser spezialisierten Editoren und Anzeiger betrachtet werden.

3.2.1 OLE 2 – Visual Editing

OLE 2[6] ist Mircosofts Technologie zur Realisierung von Verbunddokumenten. OLE 2 wurde als Nachfolger von OLE 1 in der Mitte der neunziger Jahre vorgestellt und ist mittlerweile in COM+[7] und DCOM[8] eingeflossen.
OLE 2 brachte ein neues Schlagwort in die Landschaft der Verbunddokumente – das „Visual Editing[9]“ (siehe [Meyer 94]).
Beim Visual Editing sieht man alle Teile des Dokuments so, wie sie zusammengeführt wurden. Jeder Teil des Dokuments wird von seinem speziellen Anzeiger angezeigt. Wird nun ein Teil des Dokuments durch einen Doppelklick aktiviert – daher die besser passende Bezeichnung „In-Place-Activation“ –, so wird der passende Editor gestartet. Im OLE-Zusammenhang wird dies auch eine „Server-Applikation“ genannt, da sie der „Container-Applikation“, also jenem Programm, das das Container-Dokument bearbeitet, seine Editierfunktionen zur Verfügung stellt.
Nach der Aktivierung eines Teils des Dokumentes übernimmt die Server-Applikation die Kontrolle über das Fenster der Container-Applikation. Das Dokument als ganzes bleibt weiterhin sichtbar, aber der Rest des Fensters wird von der Server-Applikation umgebaut. Es werden neue Menüs und Symbolleisten eingefügt und bestehende verändert. Die Status- und Titelleiste werden möglicherweise der Server-Applikation angepasst und weitere benötigte Fenster erscheinen auf dem Desktop.
Obwohl jetzt also der spezielle Editor aktiv ist, sieht der Anwender nach wie vor sein Dokument und merkt, wenn sich die Entwickler von Container- und Server-Applikation an einige Konventionen gehalten haben, nichts von dem Wechsel in ein anderes Programm. Tatsächlich sollen nach [Meyer 94] bei durch Microsoft durchgeführten Tests, nur wenige Benutzer diesen Wechsel bemerkt haben. Mir scheint das angesichts der Abbildungen 3.2 und 3.3 eher unwahrscheinlich. Der Benutzer muss eine derart drastische Veränderung merken.

3.2.2 OpenDoc

OpenDoc wurde von den Compound Integration Laboratories (CI Labs), einem Zusammenschluß von Apple, IBM, Wordperfect (Corel) und anderen Firmen, entwickelt und kann als deren Antwort auf Microsofts OLE 2 verstanden werden ([Burdack 97], [Schürig 97]). OpenDoc hat jedoch nur wenig oder jedenfalls nicht genügend Unterstützung erfahren, was dazu führte, dass die Entwicklung OpenDocs eingestellt wurde.
Laut [Burdack 97] und [Schürig 97] soll es eine Neuimplementierung von OpenDoc auf Basis von JavaBeans geben. Dies scheint jedoch nur ein Gerücht zu bleiben, denn bis heute, also mehr als 3 Jahre nach der Veröffentlichung von [Burdack 97] und [Schürig 97], ist davon nichts Konkretes zu erfahren.

---

[6] OLE = Object Linking and Embedding
[7] COM = Component Object Model. COM+ ist eine Erweiterung von COM
[8] DCOM = Distributed COM. DCOM ist die verteilte Variante von COM.
[9] Visual Editing ist ein von Microsoft eingetragenes Markenzeichen. In früheren Publikationen wurde der treffendere Begriff des „In-Place-Activation“ benutzt.

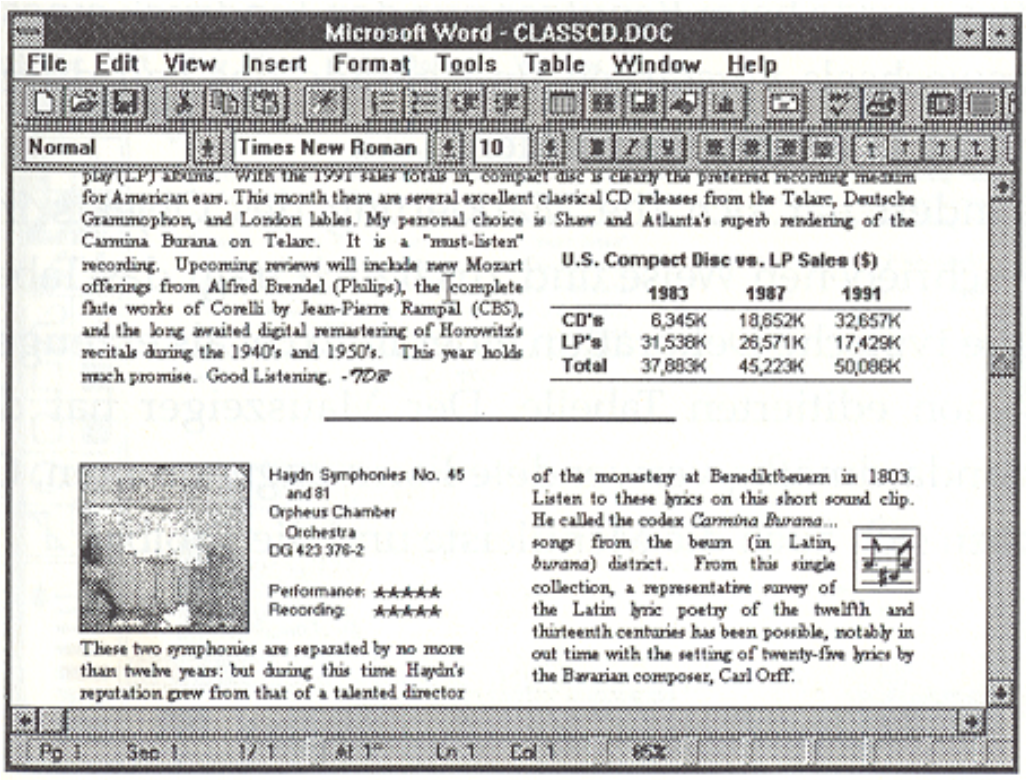

Abbildung 3.2 Ein Word-Dokument unter Bearbeitung in Word (aus [Meyer94])

Nichtsdestotrotz ist OpenDoc ein gutes Beispiel für einen dokumentenorientierten Ansatz, weshalb er hier kurz vorgestellt werden soll.
OpenDoc war eine vollkommene Neuentwicklung und hatte somit nicht das Problem der Altlasten, mit dem OLE 2 zu kämpfen hatte. OpenDoc hat somit ein viel klareres Objekt-Model als das bei OLE 2 möglich gewesen ist. Da OpenDoc von einem Zusammenschluß verschiedenster Firmen entwickelt wurde, ist es plattformunabhängig definiert. Es gibt von OpenDoc zum Beispiel Implementierungen für den Macintosh und für Windows.
In OpenDoc nimmt der Begriff „Part[10]“ eine wichtige Rolle ein. Es gibt Container-Parts und Leaf-Parts[11]. Container-Parts sind hierbei mit den Container-Dokumenten aus OLE 2 vergleichbar. Sie können andere Parts enthalten. Leaf Parts sind Parts, die keine anderen Parts enthalten können. Parts können, wie auch in OLE 2, Textdokumente, Grafikdokumente, Tondokumente etc. sein.
Ein Dokument besteht bei OpenDoc aus Parts, die von Part-Editoren bearbeitet oder von Part-Anzeigern angezeigt werden können. Während die einzelnen Teile eines Dokumentes in OLE 2 immer einen rechteckigen Bereich einnehmen, können Sie in OpenDoc verschiedenste Formen annehmen. Die in Abbildung 3.4 sichtbaren Augen sind Parts von ovaler Form.
Wird ein Part aktiviert, bekommt der entsprechende Part-Editor Zugriff auf das Menü und darf dort an bestimmten, jedoch im Unterschied zu OLE 2 nicht an allen Stellen, neue Einträge einführen.
Die Aktivierung eines Parts erfolgt wie auch bei OLE 2 durch einen Klick auf den entsprechenden Part. Eine weitere Verbesserung gegenüber OLE 2 ist, dass man sich nicht durch Hierarchien von

[10] engl.: part = Teil
[11] engl.: leaf = Blatt

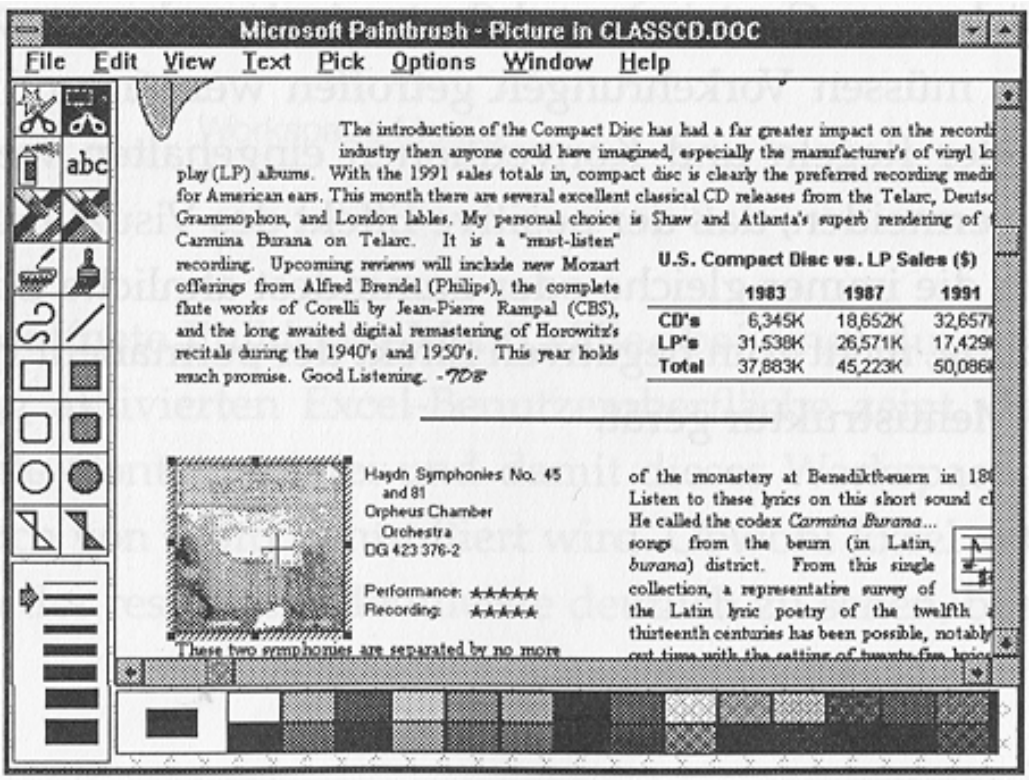

Abbildung 3.3 Das selbe Dokument wie in Abbildung 3.2 mit einem aktivierten Bitmap (aus [Meyer94])

Parts durchklicken muss, um schließlich den gewünschten Part zu aktivieren. Ein Part wird sofort aktiviert, unabhängig davon, an welcher Stelle in der Part-Hierarchie er sich befindet.
Die Kommunikation zwischen Part-Editoren und Part-Anzeigern wird nicht über Methodenaufrufe, wie dies etwa bei OLE 2 der Fall ist, sondern über einen Ereignis-Mechanismus realisiert. Da diese Ereignisse durch das „Message Interface“ OpenDocs geleitet werden, ist es möglich diese mit einem Macro-Recorder aufzunehmen und zu einem späteren Zeitpunkt wieder abzuspielen ([Burdack 97]). Eine Veranschaulichung dieses Umstandes ist in Abbildung 3.5 zu sehen.
Bei OLE 2 muß immer eine Anwendung gestartet werden, die OLE 2 unterstützt und die dann die ganze OLE-2-Maschinerie zum Bearbeiten eines Container-Dokumentes anlaufen lässt. Dabei wählt man immer die Anwendung aus, die die Art von Container-Dokument bearbeiten kann, das man gerade erstellen will. Bei Open Doc gibt es die „Document Shell“, die als Startpunkt für alle OpenDoc Aktivitäten dient. Die Funktion der Document Shell kann von einem eigenständigen Programm oder idealerweise vom Betriebssystem übernommen werden. Abbildung 3.6 zeigt eine Document Shell auf einem Macintosh-Rechner, wobei das Betriebssystem die Rolle der Document Shell übernimmt.

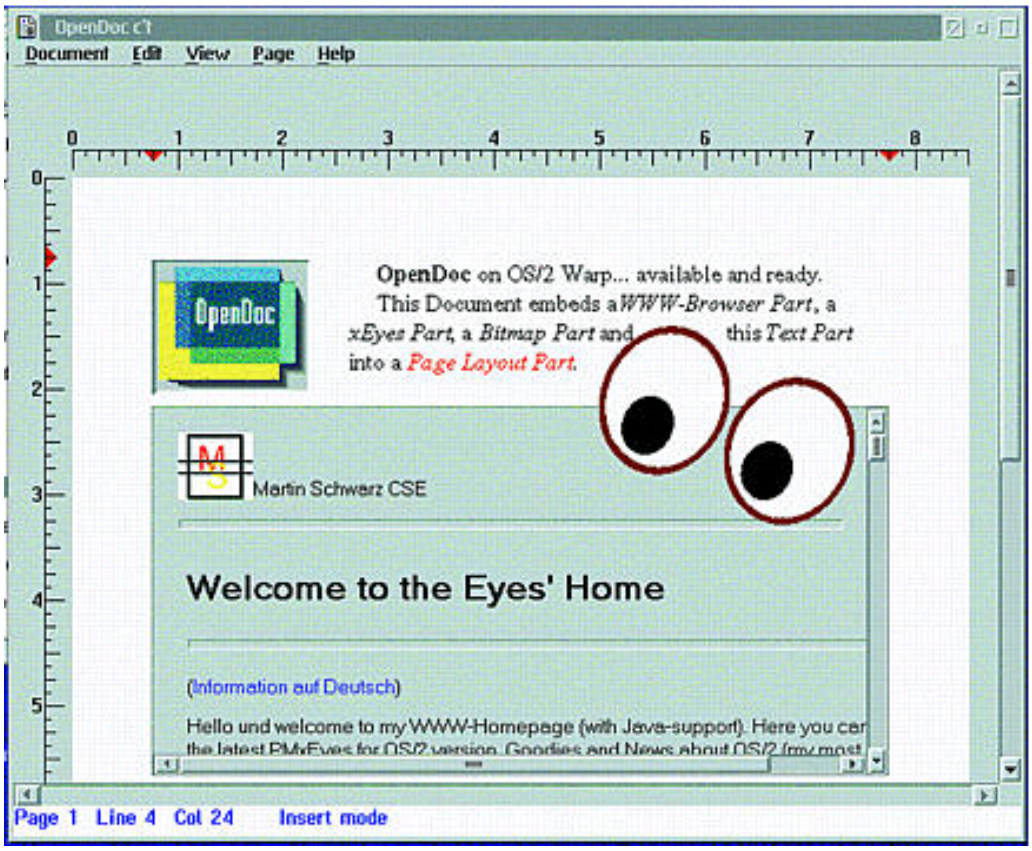

Abbildung 3.4 Ein Part mit einer nicht rechteckigen Form (aus [Schürig97])

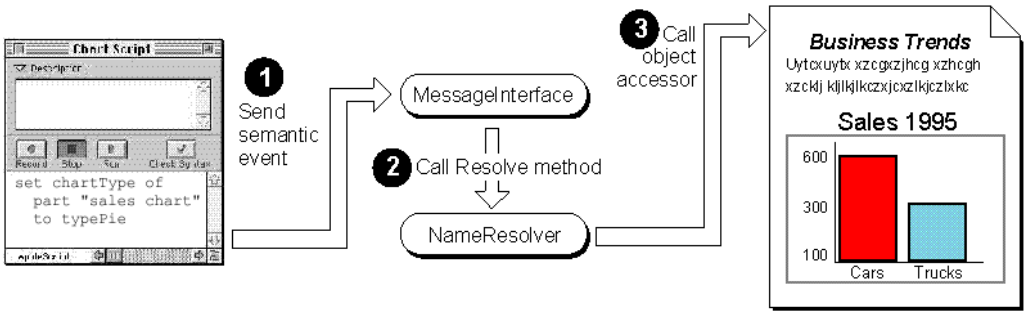

Abbildung 3.5 Der Event-Mechanismus OpenDocs mit Scripting-Funktionalität (aus [Burdack97])

3.2.3 KDE KParts

KParts ist das Komponenten-Rahmenwerk von KDE 2[12]. Die Verwendung dieses Rahmenwerkes

[12] KDE (K Desktop Environment) ist eine Suite von Programmen und Technologien für den Einsatz unter XWindows, der grafischen Benutzeroberfläche für UNIX-Systeme. KDE enthält neben einem Windowmanager andere Programme, die alle das gleiche „Look and Feel“ haben und ein gemeinsames Rahmenwerk benutzen. Der Einsatz des im KDE enthaltenen Rahmenwerkes schafft Möglichkeiten der Kommunikation zwischen den einzelnen Anwendungen,

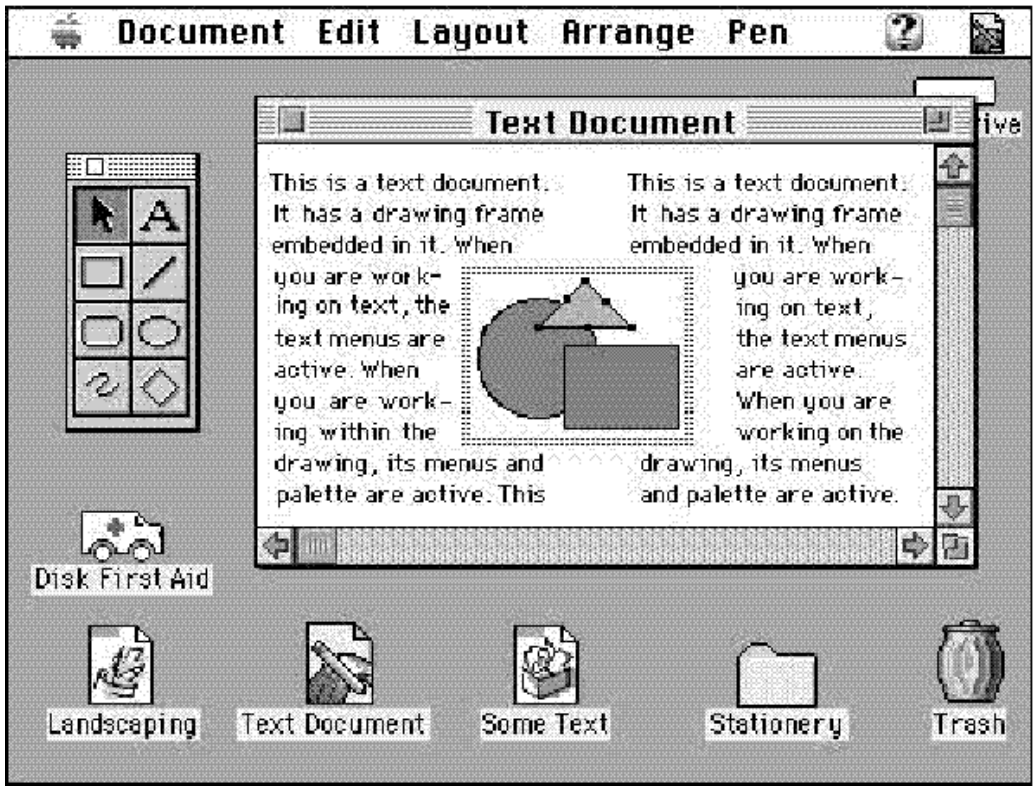

Abbildung 3.6 Document Shell mit Text-Part und aktiviertem Grafik-Part (aus [Burdack97])

ist in allem oder fast allem, was KDE 2 ausmacht, sichtbar. Das fängt mit dem zentralen Werkzeug KDEs, Konquerer, an. Konquerer ist eigentlich ein Dateimanager. Nebenbei ist er aber auch ein Web-Browser und ein Anzeiger für alle möglichen Dateiformate. Die Spitze der Leistungsfähigkeit erlebt KParts zur Zeit in KOffice, einer speziell für KDE entwickeltem Office-Suite. KParts ist hier die dahintersteckende Technologie.
KParts ist, wie auch OpenDoc und OLE 2, ein dokumentenorientierter Ansatz für die Komponentenentwicklung. Obwohl es auch ohne größere Probleme zur aufgabenorientierten Komponentenentwicklung benutzt werden kann, kann es seine volle Mächtigkeit erst beim Umgang mit Dokumenten beweisen.
In KParts werden Komponenten Parts genannt. [Sweet 00] sagt dazu :„A KDE component is called a part, and it encasulates three things: a widget, the functionality that comes with it, and the user interface“
Ein großer Unterschied zu OpenDoc ist, dass nicht die Dokumente die Parts sind, sondern die Komponenten, die die Dokumente anzeigen.
Um ein Part zu erstellen braucht man also drei Dinge

die in XWindows nicht vorgesehen sind. KDE 2, die zweite Version von KDE, erweitert dieses Framework unter anderem um einige komponentenorientierte Konzepte. Das Ziel der KDE-Entwickler ist die Schaffung eines Zuganges zu UNIX-Systemen und XWindows, der in der Benutzerfreundlichkeit Microsoft Windows oder MAC OS gleicht. Mehr Informationen zu KDE sind unter www.kde.org verfügbar.

- Das Widget, also die Oberfläche, die der Benutzer sehen wird.
- Die Funktionalität, die hinter der Oberfläche steckt
- Das „User Interface“, das die Oberfläche mit der Funktionalität verbindet.

Hier benutzt [Sweet 00] den Begriff „User Interface“ in einer ungewöhnlichen Form. Während vielerorts User Interface als Synonym für das benutzt wird, was ich hier mit „die Oberfläche“ bezeichnet habe, ist es nach [Sweet 00] nur ein kleiner, wenn auch bedeutender, Teil der Oberfläche. Mit User Interface werden hier die Einträge des Parts in den Menüs und Symbolleisten des einbettenden Programmes bezeichnet.
Eine Besonderheit ist, dass die Bereitstellung des User Interfaces nicht etwa dadurch erreicht wird, dass das Part dieses in die Menüs und Symbolleisten des einbettenden Programmes selbst integriert, sondern dass es eine XML-Datei zur Verfügung stellt, in der genau beschrieben ist, in welches Menü oder in welche Symbolleiste welche Aktion hineinzustellen ist.
Ein aus [Sweet 00] entnommenes und von mir erweitertes Beispiel soll dies dokumentieren:

```
<!DOCTYPE kpartgui>
<kpartgui name="NotepadPart" version="1">
 <MenuBar>
  <Menu name="Edit"><Text>&amp;Edit</Text>
   <Action name=selectall"/>
   <Separator/>
   <Merge/>
  </Menu>
 </MenuBar>
 <ToolBar fullWidth="true" name="mainToolBar"><Text>Main</Text>
   <Action name=selectAll"/>
 <ToolBar>
</kpartgui>
```

In diesem Beispiel wird für ein Part mit dem Namen „NotepadPart“ bestimmt, dass es in einem Menü mit dem Namen „Edit“ und einer Symbolleiste „mainToolBar“ die Aktion „selectAll“ anbietet. Mit `<Separator/>` wird in ein Menü oder in eine Symbolleiste ein Trenner eingefügt, während `<Merge/>` angiebt, dass an der Stelle dieses Tags etwaige Einträge von Sub-Parts zu diesem Menü oder dieser Symbolleiste eingefügt werden sollen. Das Rahmenwerk parst zur Laufzeit alle XML-Dateien der betroffenen Parts und erstellt daraufhin Menüs und Symbolleisten.
Die Aktionen werden im Konstruktor des entsprechenden Parts (Programme für das KDE werden ausschließlich in C++ entwickelt) durch ein Konstrukt der folgenden Art mit der Funktionalität verbunden (ebenfalls aus [Sweet 00])

```
(void) new KAction( i18n("Select All"), 0, this,
       SLOT( slotSelectAction()), actionCollection(), selectAll" );
```

Dabei ist slotSelectAction() die Operation die zur Ausführung einer Aktion aufgerufen wird.
Nun kann ein anderes Part oder eine Anwendung dieses Part benutzen, in dem es dieses Part instanziiert. Dabei gibt das Eltern-Part beziehungsweise die Eltern-Anwendung dem Kind-Part ein Widget mit, in das sich das Part einbetten soll. Dieses mitgegebene Widget ist sinnigerweise schon im Eltern-Part beziehungsweise der Eltern-Anwendung eingebettet.
Dieses Vorgehen bietet sich an, wenn man KParts zur aufgabenorientierten Komponentenentwicklung benutzen will. Soll jedoch dokumentenorientiert entwickelt werden, ist dies nicht der richtige Weg.

Hier bietet sich eine dynamischere Herangehensweise an. KDE hält eine Liste aller möglichen MIME-Typen[13] mit den zu deren Anzeige und Bearbeitung favorisierten Parts bzw Anwendungen bereit, die vom Benutzer des KDE verändert werden kann.

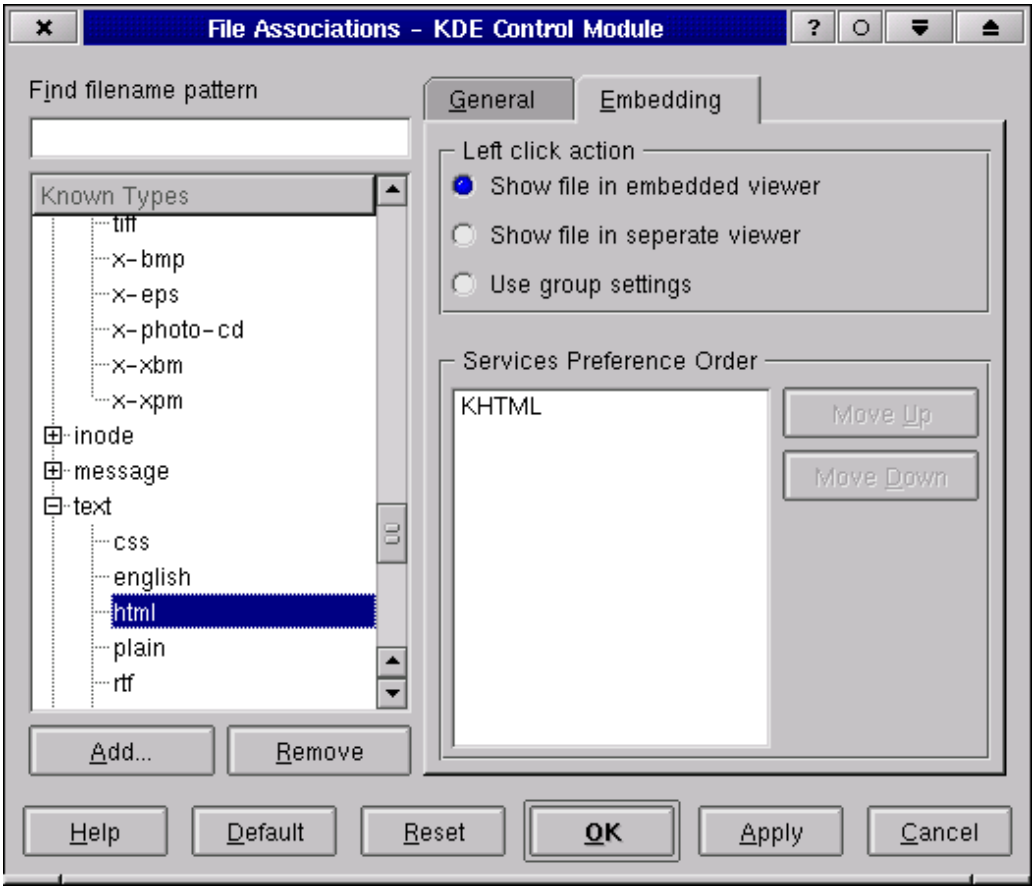

Abbildung 3.7 KDE Part-zu-MIME-Type-Zuordnung

Ein Part, das einen geeigneten Part zum Anzeigen eines in das eigene Dokument eingebetteten Dokumentes sucht, wendet sich an eine im KDE laufende Fabrik, hält dieser den MIME-Typ hin und erhält daraufhin eine spezialisierte Fabrik, die ein passendes Part erzeugen kann.

3.3 Zusammenfassung und Bewertung

In diesem Kapitel habe ich einige Konzepte vorgestellt, die sich mit dem mir gestellten Problem der Einbettung von Subwerkzeugen in der einen oder anderen Weise befassen.
Diese habe ich in zwei Gruppen von Ansätzen eingeteilt.

[13] MIME (Multipurpose Internet Mail Extensions) macht das Verschicken und Empfangen von binären e-mail-Anhängen benutzerfreundlicher. Es sind verschiedenen MIME-Typen definiert, so dass E-Mail-Programme auf bestimmte Anhänge in einer bestimmten Weise reagieren können. So zeigen heutige E-Mail-Programme zum Beispiel Bilder, die als Anhang mit einer e-mail verschickt werden, automatisch an, da sie über den MIME-Typ wissen, dass es sich bei diesen Binärdaten um Bilder handelt. Da MIME ein Standard ist und es für sehr viele Binärdatenformate Typdefinitionen gibt, bietet es sich an, diese auch in anderen Kontexten zu verwenden.

Die eine Gruppe ist die Gruppe der aufgabenorientierten Ansätze, bei denen die Komponenten mit dem Augenmerk auf die Aufgaben, die sie erledigen sollen, entwickelt werden. Solche Komponenten können zum Beispiel eine Textverarbeitung, ein Kalendar, ein Komponente zum Anzeigen von Bilder und vieles mehr sein. Ein Entwickler der seine Software aufgabenorientiert entwickelt, entwickelt nicht die ganze Funktionalität selbst, sondern benutzt für einzelne Aufgaben, die in der Benutzung seiner Software auftreten spezialisierte Komponenten anderer Entwickler. Bei den aufgabenorientierten Ansätzen habe ich die ActiveX-Steuerelemente (siehe dazu Abschnitt 3.1.1), eine Weiterentwicklung der Visual-Basic-Steuerelemente, und die Diplomarbeit „Komponentenorientierte Werkzeugkonstruktion“ von Frank Fröse (siehe dazu Abschnitt 3.1.2) vorgestellt.
Die zweite Gruppe ist die Gruppe der dokumentenorientierten Ansätze. Bei diesen, von denen ich hier OLE 2 (siehe Abschnitt 3.2.1), OpenDoc (siehe Abschnitt 3.2.2) und KDE KParts (siehe Abschnitt 3.2.3) vorgestellt habe, verschiebt sich die Konzentration weg von Programmen zu den Dokumenten, die mit diesen Programmen erstellt und bearbeitet werden. Das Dokument ist der zentrale Begriff dieser Ansätze. Es geht hier nicht darum, mit welchem Programm etwas bearbeitet wird, sondern was bearbeitet wird. Eine Klasse von Dokumenten – die Container Dokumente – nehmen hier eine besondere Rolle ein. In diese Dokumente können andere Dokumente anderer oder gleicher Art eingebettet werden. Die Komponenten in den dokumentenorientierten Ansätzen können immer eine bestimmte Art von Dokumenttyp bearbeiten oder auch nur anzeigen. Die Komponente, die ein Container Dokument bearbeiten kann, benutzt für die Bearbeitung eingebetteter Dokumente die für diese Dokumente spezialisierten Komponenten.
Bei beiden Ansatzgruppen ist eine starke Komponentenorientierung zu sehen. Beide führen bei richtiger Anwendung zu kürzeren Entwicklungszeiten bei tendenziell höherwertigerer Software.
In beiden Ansatzgruppen werden Komponenten in einander eingebettet, die mit den Werkzeugen im WAM-Ansatz vergleichbar sind. Damit bieten beide Lösungsideen für das mir gestellte Problem. Nun stellt sich die Frage, ob die aufgabenorientierten oder die dokumentenorientierten Ansätze besser in die WAM-Welt passen.
Ich greife hier auf die in Abschnitt 1.3.2 gegebene Beschreibung des Werkzeug-und-Material-Ansatzes zurück. Ich habe dort beschrieben, dass ein Softwareentwickler der nach WAM entwickelt bei der Analyse Gegenstände – Materialien und Werkzeuge – identifiziert. Beim Stichwort „Material“ könnte man jetzt sehr schnell eine Verbindung zu den Dokumenten in den dokumentenorientierten Ansätzen sehen. Diese Verbindung ist jedoch nur auf den ersten Blick vorhanden. Entwickler, die nach WAM entwickeln, entwickeln Software mit dem Ziel, die späteren Benutzer bei der Erledigung einer Aufgabe zu unterstützen. Hier kommt also auch der Begriff der „Aufgabe“ hinzu, der also auch eine Verbindung zu den aufgabenorientierten Ansätzen herstellt. Obwohl die Materialien im WAM-Ansatz eine wichtige Rolle einnehmen, stehen sie nicht alleine an der Spitze, was sie aber tun müssten, um in WAM einen dokumentenorientierten Ansatz sichtbar machen zu können. Mindestens ebenbürtig sind hier die Werkzeuge, also jene Gegenstände, mit denen die Materialien bearbeitet werden. Hier können verschiedene Werkzeuge mit verschiedenen Auswirkungen auf einem Material arbeiten. Genausowenig gleichgültig wie, ob ein Stück Holz mit einer Axt oder einer Pfeile bearbeitet wird, ist es hier gleichgültig, mit welchem Werkzeug ein Material bearbeitet wird. Dies bewegt den WAM-Ansatz weg von den dokumentenorientierten Ansätzen zu den aufgabenorientierten Ansätzen, denn auch hier sind die Werkzeuge mindestens so wichtig wie die Materialien.
Ein letztes Argument gibt es noch, dass klarstellt, das WAM kein dokumentenorientierter Ansatz sein kann, sondern ein aufgabenorientierter Ansatz ist. Die Dokumente in den dokumentenorientierten Ansätzen können beliebig in einem Container Dokument zusammengestellt werden. Dieses Verhalten wird man in einer WAM-Anwendung so gut wie nie zu sehen bekommen. Die Materialien repräsentieren hier Gegenstände aus der Anwendungswelt. Das können Bankkonten, Mappen, Belege und weiteres mehr sein. Die beliebige Zusammenstellung dieser Materialien zu einem größeren

Ganzen macht keinen Sinn. So wie in der Anwendungswelt eine Mappe niemals in ein Bankkonte gelegt wird, so wird es auch in einer WAM-Anwendung nie derartige Möglichkeiten geben. Materialien in WAM sind keine kreativen Erzeugnisse. Sie sind Erzeugnisse, die bei der Erledigung einer Aufgabe entstehen und verändert werden. Materialien können einen gültigen oder ungültigen Zustand haben. Bei Dokumenten in den dokumentenorientierten Ansätzen macht es keinen Sinn, zwischen gültigen und ungültigen Dokumenten zu unterscheiden. Dokumente existieren einfach.
Nun habe ich also festgestellt, dass WAM ein aufgabenorientierter Ansätz ist. Das bedeutet nun natürlich nicht, dass die dokumentenorientierten Ansätze uninteressant für meine Aufgabe werden. Zum Beispiel ist die Idee, die Menüstruktur eines Werkzeuges in eine XML-Datei auszulagern, wie dies bei KDE KParts gemacht wird, auch für einen aufgabenorientierten Ansatz von Interesse. Jedoch werde ich im folgenden die beiden aufgabenorientierten Ansätze näher betrachten, die ich in den Abschnitten 3.1.1 und 3.1.2 dieses Kapitels vorgestellt habe.
Die ActiveX-Steuerelemente sind ein aufgabenorientierter Ansatz, der durch diese Tatsache schon besser zu WAM passt als die dokumentenorientierten Ansätze. ActiveX-Steuerelemente sind, wie schon früher in diesem Kapitel beschrieben, Steuerelemente, die in die Oberflächen anderer Programme eingebettet werden können. Ein interessantes Feature, dass sicher auch zu WAM und JWAM passen würde, ist der Entwurfsmodus von ActiveX-Komponenten. Es würde ein Baukastensystem für JWAM ermöglichen, mit dem sich Werkzeugkomponenten zusammenstecken lassen würden. Jedoch hat ActiveX einen nicht zu vernachlässigenden Nachteil. ActiveX-Komponenten sind, wie schon gesagt, Steuerelemente. Steuerelemente können nicht alleine, unabhängig laufen. ActiveX-Steuerelemente alleine genügen also nicht, um Werkzeuge entwickeln zu können. ActiveX-Steuerelemente müssen also zu guter Letzt in Programme eingebettet werden, die keine ActiveX-Steuerelemente sind. Würden in JWAM Werkzeugkomponenten mit ActiveX-Steuerelementen entwickelt werden, gäbe es einen Bedarf für ein Konzept von Werkzeugen, die keine Werzeugkomponenten sind. Das ist meiner Meinung nach unbefriedigend. Meiner Meinung nach sollten Werkzeugkomponenten sowohl eingebettet werden können als auch die Fähigkeit haben alleine zu laufen. Außerdem haben Werkzeuge, die nach WAM gebaut werden, auch eine Menüleiste und möglicherweise eine Symbolleiste (siehe Abschnitt 2.1 „Die Anatomie eines Werkzeuges nach WAM“). ActiveX-Steuerelemente können weder Menüleisten noch Symbolleisten haben, was dadurch begründet ist, dass sie eben nur Steuerelement sind.
Die Diplomarbeit „Komponentenorientierte Werkzeugkonstruktion“ von Frank Fröse nimmt eine besondere Rolle ein. Sie wurde im Hinblick auf WAM geschrieben. Eine Diskussion, ob das dort beschriebene Konzept zu WAM passt, scheint mir unnötig zu sein. Was mir jedoch diskussionswürdig scheint, ist, ob die Umsetzung in JWAM hineinpasst und empfehlenswert ist. Ein Teil des von Frank Fröse vorgestellten Konzeptes und seiner Realisierung ist durch die Studienarbeit von Johanna Felski und Erdal Özkan (siehe [Felski 00]) in JWAM eingeflossen. Nur der Teil, der sich mit der Einbettung von Subwerkzeugen in Kontextwerkzeuge befasst, fehlt. Die Frage ist nun, ob es sinnvoll ist diesen Teil auch in JWAM zu integrieren, oder ob es besser ist, ein eigenes Konzept aufzustellen. Ich denke, dass die Schaffung und Implementierung eines eigenen Konzeptes für diesen Teil vorteilhafter ist. Ein Grund dafür ist, dass es auch in Frank Fröses Konzept ausgezeichnete Werkzeuge gibt, die einen Abschluss bilden, die also alleine für sich ohne ein Kontextwerkzeug über ihnen laufen können. Diese ausgezeichneten Teile sind sogenannte **RootToolComponent**s und haben einen Top-Level-GUI-Container, in dem sich ihre Oberflächen und die eingebetteten Oberflächen ihrer Subwerkzeuge befinden. Ein zweiter Grund ist, dass Frank Fröses Werkzeugkomponenten ausführbare Aktionen anbieten, die Objekte einer Subklasse von `javax.swing.Action` sind. Hiermit wird die Realisierung vom Swing-Toolkit abhängig. Da bei der Entwicklung von JWAM immer darauf geachtet wird, das Rahmenwerk nicht von einem bestimmten GUI-Toolkit abhängig zu machen, ist das nicht wünschenswert.

Ich werde im folgenden Kapitel ein Konzept vorstellen, von dem ich denke, dass es das Gute aus den vorgestellten Konzepten in einer JWAM angemessenen Weise zusammenstellt und erweitert.

4 Konzeption

In diesem Kapitel werde ich ein von mir entwickeltes Konzept beschreiben und begründen. Im Laufe dieses Kapitels werde ich Probleme bei der Werkzeugkonstruktiin vorstellen, die mit dem Thema der Einbettung von Subwerkzeugen zu tun haben, werde daraus erwachsende Ziele und Lösungen vorstellen, die zum Erreichen der Ziele führen.

4.1 Einbettung der Oberflächen

Das primäre Ziel dieser Arbeit ist, die **Einbettung von Oberflächen** zu erleichtern und so gut wie möglich zu automatisieren. Dabei geht es nicht um beliebige Oberflächen, sondern um die Präsentationen von Subwerkzeugen, die in die Oberflächen ihrer Kontextwerkzeuge eingebettet werden. Ich habe dieses Problem in Abschnitt 2.4 schon kurz beleuchtet und werde es hier erneut aufgreifen und von mehreren Seiten eingehender betrachten. Hierbei werde ich das Problem in mehrere Teilprobleme unterteilen. Nach der Betrachtung dieser Teilprobleme werde ich meine konzeptionelle Lösung vorstellen.

4.1.1 Die Probleme

Die geplante Vereinfachung und Automatisierung soll vor allem dazu dienen, den Komponentencharakter von Werkzeugkomponenten stärker herauszuarbeiten. Eine Vorstellung, die ich habe, ist, dass es die Möglichkeit geben soll, einen Markt für JWAM-Werkzeugkomponenten aufzubauen. Es ist kein Ziel dieser Arbeit, diesen Markt aufzubauen, sondern viel mehr ihn zu ermöglichen. Die Möglichkeit eines solchen Marktes wäre ein Indiz dafür, dass der Komponentencharakter der Werkzeugkomponenten die nötige Reife erreicht hat.
Was einen solchen Markt zur Zeit verhindert ist die Tatsache, dass ein Entwickler, der eine Werkzeugkomponente eines anderen Entwicklers einsetzen will, sehr viel über diese Komponente wissen muss. Er muss wissen, was für Exemplare welcher GUI-Klassen er erzeugen muss. Er muss wissen, welche der Klassen der Komponente die GUI-Klassen sind und wie die fertige Präsentation der Komponente übergeben werden muss. Genügt es einfach die Präsentation zu erzeugen und über eine nicht-standardisierte Prozedur an die Komponente zu überreichen? Oder muss die Präsentation verpackt in einem Exemplar der Klasse `IAFContextImpl` übergeben werden? Oder erwartet die Komponente, dass das Kontextwerkzeug selbst einen `IAFContextImpl` benutzt, an dem sie ihre eigenen Interaktionsformen abfragen kann? Das sind Fragen, die bisher nicht von JWAM beantwortet werden können. Dies sind auch die Fragen die den genannten Markt verhindern.
Dabei birgt der letzte Fall, dass eine Werkzeugkomponente über den `GUIContext`[14] des Kontextwerkzeuges, von dem sie – ohne eine entsprechende Zusicherung zu haben– annimmt, dass es sich dabei um ein Exemplar der Klasse `IAFContextImpl` handelt, auf die eigenen Interaktionsformen zugreift, eine große Gefahr.
Es sei angenommen, dass das Kontextwerkzeug tatsächlich ein Exemplar der Klasse `IAFContextImpl` als sein `GUIContext` benutzt. Weiterhin sei angenommen, dass das Kontextwerkzeug die Präsentation der zu benutzenden Werkzeugkomponente so eingebettet hat, dass die in ihr enthaltenen Interaktionsformen von dem `IAFContextImpl`-Exemplar auch gefunden werden können. Trotzdem kann ein weiteres Problem auftauchen, dass ich anhand des aus dem Abschnitt 2.4 bekannten Werkzeuges Device-Editor zeigen werde. Die Präsentation dieses Werkzeuges ist in Abbildung 4.1 zu sehen ist.

[14] Klassen, die die Schnittstelle `GUIContext` implementieren, werden in JWAM zwischen die Präsentation und die Interaktionskomponente gelegt. Die wichtigste Implementation findet sich in `IAFContextImpl`. Da aber das Rahmenwerk einige Aufgaben wie zum Beispiel das Anzeigen und das Verstecken der Präsentation bei Vorhandensein eines `GUIContext` selbst erledigen kann, gibt es auch eine Implementation `SwingContextImpl`, die verwendet werden kann, wenn das Muster „Trennung von Handhabung und Präsentation“ nicht verwendet werden soll.

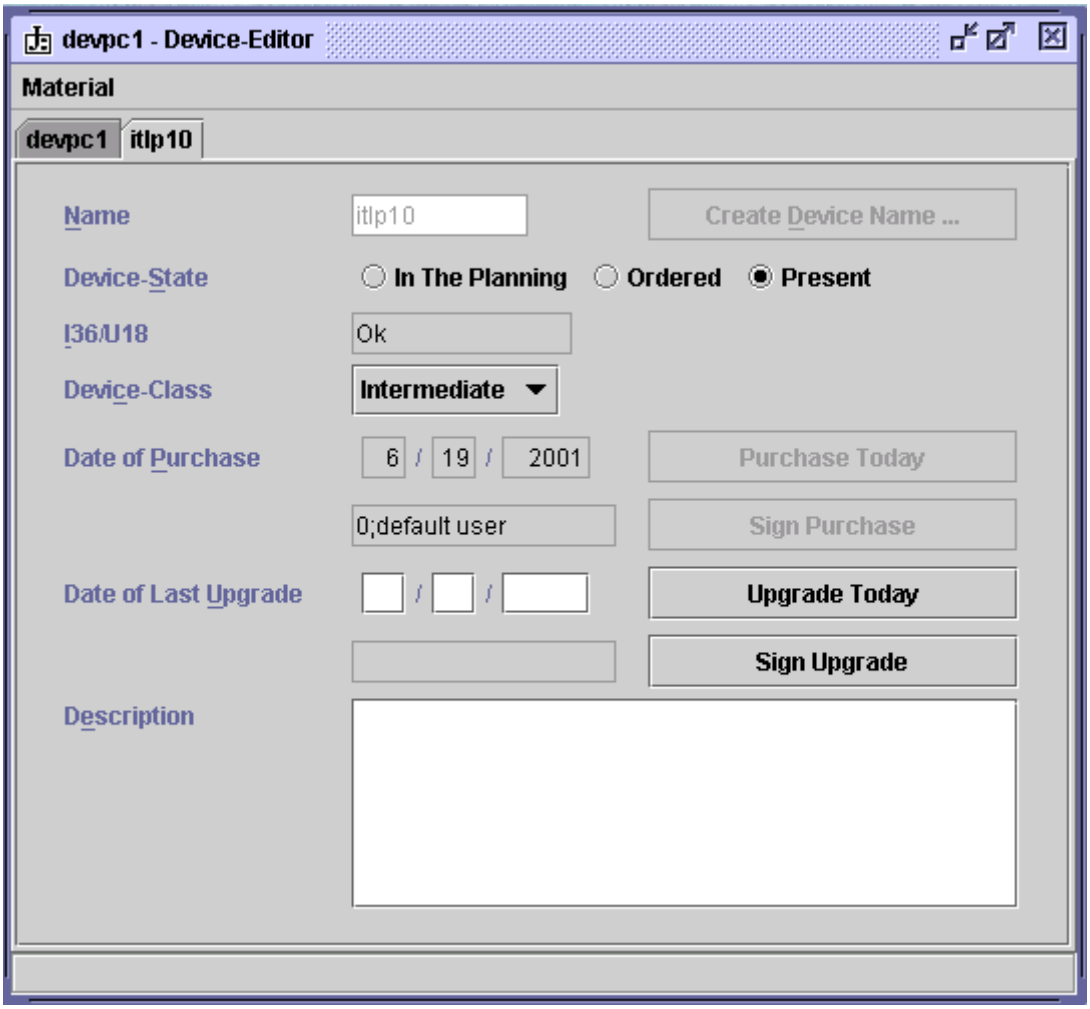

Abbildung 4.1 Die Präsentation des Werkzeuges Device-Editor

Die Präsentation des Kontextwerkzeuges hat nur sehr wenige Elemente. Nur die Menüleiste und die Tabs gehören dazu. Das meiste der Präsentation gehört zu den Subwerkzeugen, von denen sich je eines unter einem Tab befindet. Bei jedem der Subwerkzeuge handelt es sich um einen Single-Device-Editor, einem Werkzeug, das ein Material des Typs `Device` bearbeiten kann.
In der in Abbildung 4.1 dargestellten Situation werden mit dem Werkzeug Device-Editor zwei Materialien bearbeitet. Zu diesem Zweck benutzt das Werkzeug zwei Subwerkzeuge. Das eine ist sichtbar. Das andere würde durch einen Klick auf den Tab mit der Aufschrift „devpc1“ sichtbar werden. Da beide Subwerkzeuge über den `GUIContext` ihres Kontextwerkzeuges auf ihre Interaktionsformen zugreifen, entsteht hier ein Konflikt. Wird ein Objekt des Typs `IAFContextImpl` nach einer Interaktionsform gefragt, so sucht es in der an ihm gesetzten Präsentation nach dieser Interaktionsform. Da es in JWAM spezielle Widget-Klassen gibt, die einerseits von Toolkit-Widget-Klassen erben und andererseits Interaktionsform-Schnittstellen implementieren, wird dabei an jedem Widget geprüft, ob es eine bestimmte Interaktionsform-Schnittstelle implementiert und wenn dem so ist, ob es den richtigen Interaktionsformnamen hat. Wenn jetzt das Subwerkzeug unter dem Tab „itlp10“ den Kontext nach einem Aktivator[15] mit dem Namen „signpurchase“ fragt, sollte es eine

Referenz auf den Knopf „Sign Purchase“ erhalten. Da in die Präsentation des Kontextwerkzeuges die Präsentationen beider Subwerkzeuge eingebettet sind, sind auch alle Interaktionsformen der Subwerkzeuge doppelt vorhanden. Es gibt dort also eine Interaktionsform `signpurchase` für das Subwerkzeug unter dem Tab „devpc1“ und eine für das Werkzeug `itlp10`. Es ist also nicht sichergestellt, dass das fragende Subwerkzeug auch wirklich seine Interaktionsform erhält und nicht die des anderen Subwerkzeuges. Da der `IAFContextImpl` die an ihm gesetzte Präsentation sequenziell durchsucht, wird das erste Widget, dass die Anforderungen erfüllt, zurückgegeben. In der in der Abbildung dargestellten Situation stehen die Chancen nicht schlecht, dass das Werkzeug unter dem Tab „itlp10“ den Aktivator des Werkzeuges unter dem Tab „devpc1“ bekommt, da dieses anscheinend früher in das Kontextwerkzeug eingebettet wurde und deshalb seine Präsentation auch früher in der Hierarchie der Widgets auftaucht.
Die Frage, die sich hier also stellt, ist, wie sich mehrere Subwerkzeuge des gleichen Typs in einem Kontextwerkzeug einbetten lassen. Es gibt da zwei Möglichkeiten.
Wenn der Entwickler den Sourcecode der Werkzeugkomponente hat, von der er mehrere Exemplare als Subwerkzeuge benutzen will, kann er ihn kopieren und in der Kopie die Namen der Interaktionsformen ändern. Er wird dann ein Exemplar des Originals einsetzen und ein Exemplar der Kopie. Dies ist nun aber ganz und gar nicht ein komponentenorientiertes Vorgehen.
Eine andere – und der Komponentenorientierung nähere – Möglichkeit ist, dass der Entwickler der Werkzeugkomponente voraussieht, dass mehrere Exemplare dieser Werkzeugkomponente auf das selbe Exemplar der Klasse `IAFContextImpl` zugreifen, und die Angabe eines Präfixes vorsieht. Der Entwickler, der diese Werkzeugkomponente einsetzt, wird nun zwei Exemplare der Präsentation dieser Komponente erzeugen und einbetten, wobei er bei der Erzeugung jedem Exemplar einen anderen Präfix mitgibt. So wird dann der Aktivator in dem einen Exemplar zum Beispiel den Namen „ex1signpurchase“ und in dem anderen „ex2signpurchase“ erhalten. Hiernach wird er auch an jedem Exemplar der Werkzeugkomponente den richtigen Präfix setzen, so dass die Interaktionskomponente jedes Exemplares auf die richtigen Interaktionsformen zugreift.
Das Problem der doppelten Interaktionsformen ist – wenn auch notdürftig – gelöst. Diese Lösungen lassen sich auch einfach auf das Problem anwenden, dass auftritt, wenn das Kontextwerkzeug in seiner eigenen Präsentation Interaktionsformen hat, die die gleichen Namen haben wie die Interaktionsformen seiner Subwerkzeuge.
Ein weiteres Problem kann schon bei der Einbettung der Präsentation einer Werkzeugkomponente, die als Subwerkzeug eingesetzt wird, auftauchen. Wenn der Entwickler der Werkzeugkomponente diese zum Einbetten vorgesehen hat, die GUI-Klasse also nicht von `javax.swing.JFrame` oder einer anderen Top-Level-GUI-Container-Klasse erbt, ist das kein Problem. Wenn sie es aber nun doch tut, ist es ein Problem, denn es ist zum Beispiel nicht möglich einen `JFrame` in ein `JPanel` einzubetten.
Da die Präsentationen der Subwerkzeuge auch weiterhin in den Kontextwerkzeugen erzeugt werden muss, kann dabei der Inhalt des `JFrame` genommen und dieser eingebettet werden. Auch das ist nicht wirklich komponentenorientiert, aber es funktioniert.
Umsichtige Entwickler von Werkzeugkomponenten können zu jeder Werkzeugkomponente, die sie konstruieren, zwei Teile ausliefern. Der eine Teil kann eine Werkzeugkomponente enthalten, die zur Einbettung gedacht ist und der andere Teil kann eine Werkzeugkomponente enthalten, die die andere einbettet und einen Top-Level-GUI-Container wie zum Beispiel ein `JFrame` als Präsentation hat. Der zweite Teil ist dann für die Situationen gedacht, wenn die Werkzeugkomponente als eigenständiges Werkzeug oder als Subwerkzeug in einem eigenen Fenster laufen soll.

[15] Ein Aktivator ist eine der möglichen Interaktionsformen. Ein Aktivator ist etwas, dass aktiviert werden kann. Das kann ein Knopf, ein Menüeintrag oder ähnliches sein.

4.1.2 Die Lösung

Ich habe im vorigen Abschnitt einige Probleme und Lösungen dieser Probleme vorgestellt. Diese Lösungen sind einsetzbar, wenn mit JWAM 1.5 Werkzeugkomponenten erstellt beziehungsweise benutzt werden sollen. Diese Lösungen sind teilweise notdürftig und nicht elegant. Vor allem führen sie nicht zu einer komponentenorientierten Werkzeugkonstruktion.
Als eines der Probleme habe ich den Bruch der Kapsel um Werkzeugkomponenten identifiziert, der notwendig wird, wenn Werkzeugkomponenten als eingebettete Subwerkzeuge verwendet werden sollen.
Eine gute Lösung muss also sicherstellen:

> Eine Werkzeugkomponente erzeugt ihre Präsentation selbst. Sie kann diese Präsentation zurückgeben und ermöglicht so die Einbettung derselben.

Da der `GUIContext` die Präsentation des Werkzeuges hält, muss dieser also entsprechend erweitert werden, dass er diese Präsentation auch zurückgeben kann.
Wenn jede Werkzeugkomponente ihre Präsentation zurückliefern kann, dann stellt sich auch die Frage nach der Verpackung dieser Präsentation. Ist die Präsentation einer Werkzeugkomponente zum Einbetten vorgesehen, so wird sie nicht in einen Top-Level-GUI-Container verpackt sein. Ist sie nur dazu gedacht, in einem eigenen Fenster zu laufen, dann wird sie in einem Top-Level-GUI-Container verpackt sein.
Ein Dilemma entsteht, wenn die Werkzeugkomponente sowohl zum Einsatz als Subwerkzeug als auch zum Einsatz als eigenständiges Werkzeug vorgesehen ist. Im ersten Fall muss die Präsentation einbettbar sein. Das Kontextwerkzeug kann sie dann einbetten oder sich um ein eigenes Fenster für die Komponente kümmern. Im zweiten Fall muss die Werkzeugkomponente selbst für ein eigenes Fenster sorgen und dort die benötigten Menüs und Symbolleisten unterbringen.
Frank Fröses Arbeit sieht eine explizite Unterscheidung dieser beiden Arten von Werkzeugkomponenten vor. Entweder eine Werkzeugkomponente kann eingebettet werden, oder sie kann eigenständig laufen (siehe hierzu Abschnitt 3.1.2). Hier gibt es ein `RootToolComponent`, dass den Abschluss für ein Werkzeuggeflecht bildet. Werkzeugkomponenten dieser Art erzeugen ein eigenes Fenster, in dass sie die eigene Präsentation und die Präsentationen der Subwerkzeuge einbetten.
Würde man diesem Lösungsvorschlag folgen, müsste man bei Werkzeugkomponenten, die sowohl als Subwerkzeuge als auch eigenständig laufen sollen, eine als Subwerkzeug lauffähige Werkzeugkomponente konstruieren und eine sehr dünne, praktisch funktionslose, die die Werkzeugkomponente einbettet und sie so eigenständig lauffähig macht. Nach außen hin wären es aber zwei Werkzeugkomponenten. Man könnte nicht von der gleichen Komponente zwei Exemplare erzeugen und eines davon einbetten, während das andere eigenständig liefe. Auch, wenn die praktisch funktionslose, den Werkzeugabschluss bildende, Werkzeugkomponente sehr dünn wäre, wäre sie trotzdem vorhanden.
Diese Vorgehensweise würde die Entwickler der Werkzeugkomponente immer zum Abwägen zwingen, was für die eigene Werkzeugkomponente sinnvoll ist. Drei Wege stünden immer zur Wahl:

- Die Werkzeugkomponente ist ausschließlich zum Einsatz als Subwerkzeug vorgesehen. Es gibt also keinen Werkzeugabschluss.
- Die Werkzeugkomponente ist ausschließlich zum eigenständigen Einsatz vorgesehen. Es gibt einen Werkzeugabschluss, aber es gibt keinen einbettbaren Teil.
- Es ist sowohl möglich die Werkzeugkomponente einzubetten als auch sie eigenständig laufen zu lassen.

Eine Entscheidung für eine der beiden ersten Varianten schließt möglicherweise sinnvolle Einsatzmöglichkeiten aus und hindert Entwickler, die die Werkzeugkomponente einsetzen wollen, einen größeren Nutzen aus der Komponente zu ziehen.
Das führt mich zu einer weiteren Forderung:

> Alle Werkzeugkomponenten haben einbettbare Präsentationen. Die Werkzeugkomponenten erzeugen keine eigenen Fenster. Wenn eine Werkzeugkomponente als eigenständiges Werkzeug gestartet wird, wird sie von der Umgebung in ein Fenster eingebettet. Es macht für die Werkzeugkomponente keinen Unterschied, ob sie eigenständig läuft oder als Subwerkzeug.

Es muss also per Nachbedingung sichergestellt werden, dass der Kontext beim Zurückgeben der Präsentation keinen Top-Level-GUI-Container zurückgibt.
Ich habe die Vision geäußert, dass das Einbetten der Subwerkzeuge größtenteils automatisch erledigt wird. Ich werde nun die Frage beantworten, wie diese Vision umgesetzt werden könnte.
Dazu muss es eine Schnittstelle geben, die ich `ToolComponentViewContainer` nenne. Sie stellt Operationen zum Feststellen, ob sie der Container für die Präsentaion eines bestimmten Subwerkzeuges ist, und zum Einbetten der Präsentation, für die sie der Container ist.
Klassen, die diese Schnittstelle erfüllen, werden von Widgetklassen erben, die die Möglichkeit bieten, andere Widgets in Exemplare von ihnen hineinzupacken. Objekte dieser Klassen werden in der Präsentation eines Kontextwerkzeuges an den Stellen platziert, an denen später die Präsentationen der Subwerkzeuge auftauchen sollen.
Beim Anfordern einer neuen Sub-FP legt die Kontext-FP fest, unter welchem Namen das Subwerkzeug eingebettet werden soll. Der Rest kann von den Klassen des Rahmenwerkes erledigt werden. Die Oberklasse aller Tool-Klassen – ToolImpl – enthält die notwendigen Algorithmen, so dass das Kontext-Tool-Objekt dem entsprechend erweiterten `GUIContext` alle einzubettenden Präsentationen übergibt. Der `GUIContext` sucht in allen Elementen seiner Präsentation nach Objekten vom Typ `ToolComponentViewContainer`, unter denen er dann den passenden Container sucht.

4.2 Werkzeugabschluss

Ich habe im vorigen Abschnitt ein Konzept beschrieben, bei dem die Umgebung für das Erzeugen eines Fensters für eigenständig laufende Exemplare von Werkzeugkomponenten zuständig ist. Das ist unproblematisch, bis der Entwickler einer Werkzeugkomponente für den Fall, dass diese als eigenständig laufendes Werkzeug benutzt wird, das Fenster um Menüs und Symbolleisten erweitern will. Da Menüs und Symbolleisten zur Anatomie eines eigenständig laufenden Werkzeuges gehören (siehe Abschnitt 2.1), wird dieser Fall sehr oft vorkommen.
Die Anforderung lautet also:

> Es muss eine Möglichkeit geben, dass die Werkzeugkomponenten in dem Fall, dass sie eigenständig in einem eigenen Fenster laufen, Zugriff auf dieses Fenster bekommen und Menüs und Symbolleisten an diesem Fenster setzen können. Diese Ausweitung der Möglichkeiten darf nicht dazu führen, dass zur Nutzung dieser Möglichkeiten die Konstruktion einer speziellen Werkzeugkomponente notwendig wird, die vor die eigentliche Werkzeugkomponente vorgeschaltet wird.

Die Werkzeugkomponenten müssen also in dem Fall, dass sie in einem eigenen Fenster laufen, von der Umgebung einen Zugriff auf dieses Fenster erhalten. Dazu muss die Schnittstelle `Tool` um eine Operation erweitert werden, über die die Umgebung diesen Zugriff herstellen kann. Die Umgebung soll hier aber nicht das Fenster selbst übergeben, da dann hier eine Abhängigkeit zu einem GUI-Toolkit eingegangen wird. Es bietet sich hier an, einen speziellen Kontext zu verwenden, der ähnliche Aufgaben übernimmt, die der `GUIContext` für die Kommunikation zwischen der Interaktionskomponente und der Präsentation übernimmt. Ein solcher Kontext kann `FrameContext` heißen und Operationen für das Setzen von Menüleisten, Symbolleisten und ähnlichem anbieten.

4.3 Ausführbare Aktionen

Werkzeugkomponenten bieten dem jeweiligen Benutzer, abhängig davon, ob sie eigenständig oder als Subwerkzeug laufen, verschiedene Handhabungsmöglichkeiten an.
Eine zu JWAM gehörende Werkzeugkomponente mit dem Namen „Container-Browser“ (siehe Abbildung 4.2), die den Inhalt eines JWAM-Containers anzeigt, bietet zum Beispiel im unteren Bereich des Fensters drei Knöpfe an, mit denen das Werkzeug geschlossen, das selektierte Element auf den Desktop gelegt oder gelöscht werden kann. Diese drei Knöpfe machen sicherlich Sinn, wenn das Werkzeug als eigenständiges Werkzeug läuft. Weniger Sinn machen sie, wenn es als eingebettetes Subwerkzeug läuft.

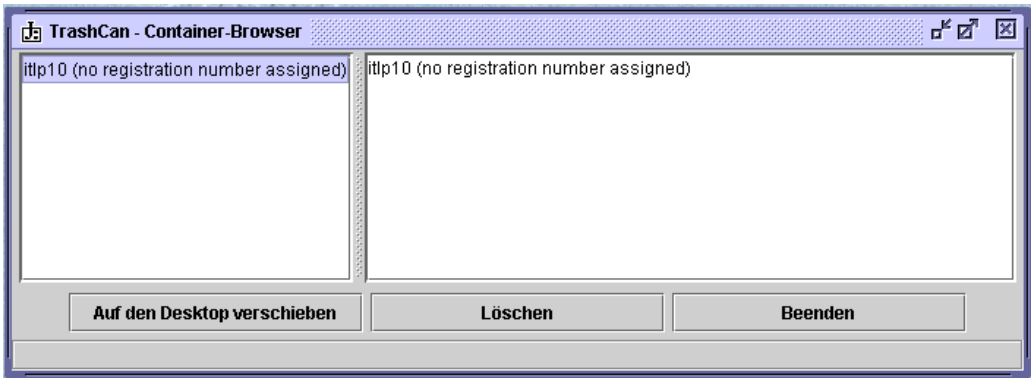

Abbildung 4.2 Das Werkzeug Collection-Browser

Solche Knopfleisten befinden sich immer am unteren Rand eines Fensters (siehe Abschnitt 2.1) und es sollte auch nur eine von ihnen pro Fenster geben. Es würde auch sehr merkwürdig aussehen, wenn jedes eingebettete Subwerkzeug einen eigenen „Beenden“-Knopf hätte. Ein Subwerkzeug sollte es dem Benutzer nicht anbieten, dieses Subwerkzeug schliessen zu können. Der Benutzer benutzt ein Werkzeug. Es ist für den Benutzer nicht interessant, aus wie vielen Subwerkzeugen dieses eine zusammengebaut ist. Wenn der Benutzer fertig ist mit der Benutzung dieses Werkzeuges, dann schließt er es, worauf das Werkzeug alle seine Subwerkzeuge schließt. Der Benutzer nimmt die Dienstleistungen des Werkzeuges in Anspruch und das Werkzeug die seiner Subwerkzeuge. Wenn der Entwickler eines Werkzeuges, das zum Beispiel die Container-Browser-Werkzeugkomponente als Subwerkzeug benutzt, entscheidet, dass es eine Möglichkeit geben soll, ein ausgewähltes Element zu löschen, dann wird er dafür sorgen, dass der Benutzer einen entsprechenden Knopf vorfindet.
Die Elemente einer Präsentation einer Werkzeugkomponente können in zwei Kategorien eingeordnet werden:

- Es gibt Elemente, die für die Benutzbarkeit der Werkzeugkomponente sowohl in der Benutzung als eigenständiges Werkzeug als auch in der Benutzung als Subwerkzeug notwendig sind.
- Es gibt Elemente, die nur dann sinnvoll sind, wenn die Werkzeugkomponente als eigenständiges Werkzeug benutzt wird.

Beim Container-Browser sind zum Beispiel die Liste und das Textfeld Vertreter der ersten Kategorie. Dieses Werkzeug würde ohne diese beiden Elemente nicht sinnvoll benutzbar sein. Die drei Knöpfe gehören aber ganz klar in die zweite Kategorie. Sicherlich sollte der Container-Browser auch als Subwerkzeug Elemente des angezeigten Containers zum Beispiel löschen können, aber in dem Fall, dass das Werkzeug als Subwerkzeug verwendet wird, sollte es dem Entwickler des Kontextwerkzeuges überlassen werden, wie er diese Funktionalität zur Verfügung stellt, wenn er sie denn überhaupt zur Verfügung stellen möchte.
In die zweite Kategorie werden immer Knöpfe und Menüeinträge fallen. Es scheint unwahrscheinlich, dass Eingabefelder oder Listen in die zweite Kategorie gehören.
Knöpfe – wie auch Menüeinträge – lösen Aktionen aus. In dem einen Fall – wenn die Werkzeugkomponente als eigenständiges Werkzeug benutzt wird – lösen die Benutzer durch Betätigen von Knöpfen und Aktivieren von Menüeinträgen diese Aktionen direkt aus. Im zweiten Fall – bei der Benutzung als Subwerkzeug – nimmt das Kontextwerkzeug die Dienstleistungen des Subwerkzeuges in Anspruch. Zu den Dienstleistungen sollte auch die Möglichkeit gehören, diese Aktionen auszulösen.
In JWAM 1.5 gibt es kein Konzept von ausführbaren Aktionen. Die Betätigung von Knöpfen wird von Interaktionskomponenten interpretiert und in Form von Befehlen an die Funktionskomponente weitergeleitet. Die Funktionskomponenten der Kontextwerkzeuge greifen auf die Funktionskomponenten der Subwerkzeuge zu und nehmen deren Dienstleistungen durch den Aufruf dafür vorgesehener Operationen in Anspruch.
Hieraus erwächst die Anforderung:

> Werkzeugkomponenten sollen ausführbare Aktionen anbieten und somit die möglichen Aktionen explizit machen. Diese Aktionen sollen von der Werkzeugkomponente benutzt werden, um Menüs und Symbolleisten am `FrameContext` zu befüllen, falls die Werkzeugkomponente als eigenständiges Werkzeug gestartet wird. Kontextwerkzeuge sollen diese Aktionen als einen Teil der von den Subwerkzeugen erbrachten Dienstleistung nutzen.

Da das Anbieten von ausführbaren Aktionen eine fachliche Angelegenheit ist, macht es Sinn die Schnittstelle `ToolFunctionality` um entsprechende Operationen zu erweitern.
Eine Aktion kann Auskunft über ihre Beschreibung, ihren Text und ihr Icon geben. Die beiden letzten – Text und Icon – sind der präferierte Text und das präferierte Icon, durch die die Aktion in einem Menü oder in einer Symbolleiste repräsentiert werden kann[16]. Ferner kann eine Aktion sagen, ob sie gerade ausführbar ist oder nicht. Eine Aktion zum Löschen des ausgewählten Eintrags wird zum Beispiel in unserem Container-Browser nur dann ausführbar sein, wenn auch ein Eintrag ausgewählt ist.

[16] Das Angeben eines Textes und eines Icons ist etwas, das nicht fachlich begründet ist. Deshalb passt es nicht so richtig in den Aufgabenbereich einer Funktionskomponente. Eine Aufteilung der Zusammenstellung von Aktionen auf die Funktionskomponente und die Interaktionskomponente schien mir das Konzept zu verkomplizieren, weshalb ich mich dazu entschlossen habe, der Funktionskomponente den ganzen Umgang mit Aktionen zu überlassen.

Da Aktionen ihren Status von „ausführbar“ zu „nicht-ausführbar“ und umgekehrt ändern können, muss die Funktionskomponente ein Ereignis anbieten, an dem sich alle Interessierten anmelden können und das bei der Änderung einer Aktion ausgelöst wird.

4.4 Präsentationen in Teilen – Maximierung der Einsetzbarkeit

Bisher bin ich bei der Diskussion der Präsentationen von Werkzeugkomponenten immer davon ausgegangen, dass die Präsentationen in einer Klasse definiert sind und so meistens in einem Stück verfügbar sind.
Das ist jedoch nicht zwingend erforderlich. Um eine maximale Flexibilität und Einsetzbarkeit zu erreichen, ist es sinnvoll über Präsentationen in Teilen nachzudenken.
Kehren wir wieder zu dem im vorigen Abschnitt vorgestelltem Werkzeug zurück. Der Container-Browser enthält eine Liste, in der er den Inhalt eines Containers darstellt und ein Textfeld, in dem er Detailinformationen zum ausgewählten Element anzeigt. In der Präsentation aus einem Guss stehen diese beiden Elemente Seite an Seite – der Auflister links und der Detailanzeiger rechts. Nun kann es sein, dass ein Entwickler gerne diese Werkzeugkomponente einsetzen will, er aber den Auflister über dem Detailanzeiger braucht. Die bisherige Konzeption bietet für diesen Fall keine Unterstützung.
Dem kann entgegengewirkt werden, indem die Schnittstelle `GUIContext` so erweitert wird, dass sie statt einer Wurzelkomponente beliebig viele akzeptiert und bei der Suche nach einem Widget auch alle diese Wurzelkomponenten durchsucht. Damit die Wurzelkomponenten im Falle der Verwendung der Werkzeugkomponente als eingebettetes Subwerkzeug an den richtigen Stellen eingebettet werden können, müssen sie unterscheidbar sein. Die einfachste Möglichkeit der Unterscheidung ist die Unterscheidung über Namen. Um auch hier keine Abhängigkeit zu bestimmten Toolkits einzugehen bietet sich die Definition einer Schnittstelle `ToolComponentViewPart`, die die einzige Operation `viewPartName` hat, an. Diese Operation gibt den Namen der Wurzelkomponente zurück[17]. Alle Wurzelkomponenten, die an einem `GUIContext` gesetzt werden, müssen diese Schnittstelle erfüllen. Damit ist auch eine Umbenennung von `ToolComponentViewContainer` in `ToolComponentViewPartContainer` und eine Anpassung an die Möglichkeit, dass eine Werkzeugkomponenten eine mehrteilige Präsentation hat, angebracht. Es muss möglich sein, für jeden Teil der Präsentation einen eigenen `ToolComponentViewPartContainer` zu haben.
Hiermit kann der Entwickler des Kontextwerkzeuges dann selbst entscheiden, wo welche Teil des Subwerkzeuges eingebettet wird. Daraus resultiert natürlich eine neue Aufgabe für den Werkzeugkomponentenentwickler. Es ist sicherlich nicht sinnvoll die Präsentationen der Werkzeugkomponenten so fein zu zerlegen, dass schließlich jedes Widget ein eigener unabhängig einbettbarer Teil wird. Hier ist für jede Werkzeugkomponente von neuem eine sinnvolle Aufteilung zu finden.
Damit der Umgang mit Werkzeugkomponenten durch diese Flexibilisierung nicht verkompliziert wird, muss der Entwickler einer Werkzeugkomponente auch immer eine Präsentation zur Verfügung stellen, die alle Teile der Präsentation in einem Guss bereitstellt.
Die Verwendung der Präsentation in Teilen ist dann eine Option.

4.5 Zusammenfassung

In diesem Kapitel habe ich ein Konzept zur Lösung des mir gestellten Problems, der Einbettung von Oberflächen bei der Werkzeugkomposition, vorgestellt und begründet. Dabei habe ich Probleme

[17] Schnittstellen mit nur einer Operation zeugen normalerweise von einem schlechten Design. In dieser Schnittstelle sind jedoch keine anderen Operationen notwendig. Sie dient allein der Unterscheidung der einzelnen Wurzelkomponenten und der Unabhängigkeit von bestimmten GUI-Toolkits.

bei der Werkzeugkonstruktion und der Einbettung von Subwerkzeugen im Speziellen aufgezeigt und im Laufe des Kapitels folgende Anforderungen gestellt.

- Eine Werkzeugkomponente erzeugt ihre Präsentation selbst. Sie kann diese Präsentation zurückgeben und ermöglicht so die Einbettung derselben.
- Alle Werkzeugkomponenten haben einbettbare Präsentationen. Die Werkzeugkomponenten erzeugen keine eigenen Fenster. Wenn eine Werkzeugkomponente als eigenständiges Werkzeug gestartet wird, wird sie von der Umgebung in ein Fenster eingebettet. Es macht für die Werkzeugkomponente keinen Unterschied, ob sie eigenständig läuft oder als Subwerkzeug.
- Es muss eine Möglichkeit geben, dass die Werkzeugkomponenten in dem Fall, dass sie eigenständig in einem eigenen Fenster laufen, Zugriff auf dieses Fenster bekommen und Menüs und Symbolleisten an diesem Fenster setzen können. Diese Ausweitung der Möglichkeiten darf nicht dazu führen, dass zur Nutzung dieser Möglichkeiten die Konstruktion einer speziellen Werkzeugkomponente notwendig wird, die vor die eigentliche Werkzeugkomponente vorgeschaltet wird.
- Werkzeugkomponenten sollen ausführbare Aktionen anbieten und somit die möglichen Aktionen explizit machen. Diese Aktionen sollen von der Werkzeugkomponente benutzt werden, um Menüs und Symbolleisten zu befüllen, falls die Werkzeugkomponente als eigenständiges Werkzeug gestartet wird. Kontextwerkzeuge sollen diese Aktionen als einen Teil der von den Subwerkzeugen erbrachten Dienstleistung nutzen.

Ich habe diese Punkte durch Vorschläge zu deren Realisierung ergänzt und werde eine mögliche Realisierung für JWAM im nächsten Kapitel vorstellen.

5 Realisierung

In diesem Kapitel werde ich anhand des bereits vorgestellten Werkzeuges Device-Editor die Realisierung des im vorigen Kapitel vorgestellten Konzeptes zeigen.

5.1 Neue Version des Device-Editor

Die verbesserte Werkzeugkonstruktion, die nun alle im vorigen Kapitel aufgestellten Anforderungen erfüllt, ermöglichte es, den Device-Editor komponentenorientiert neu zu entwickeln. Diese Komponentenorientierung ist im UML-Diagramm in Abbildung 5.1 zu sehen.

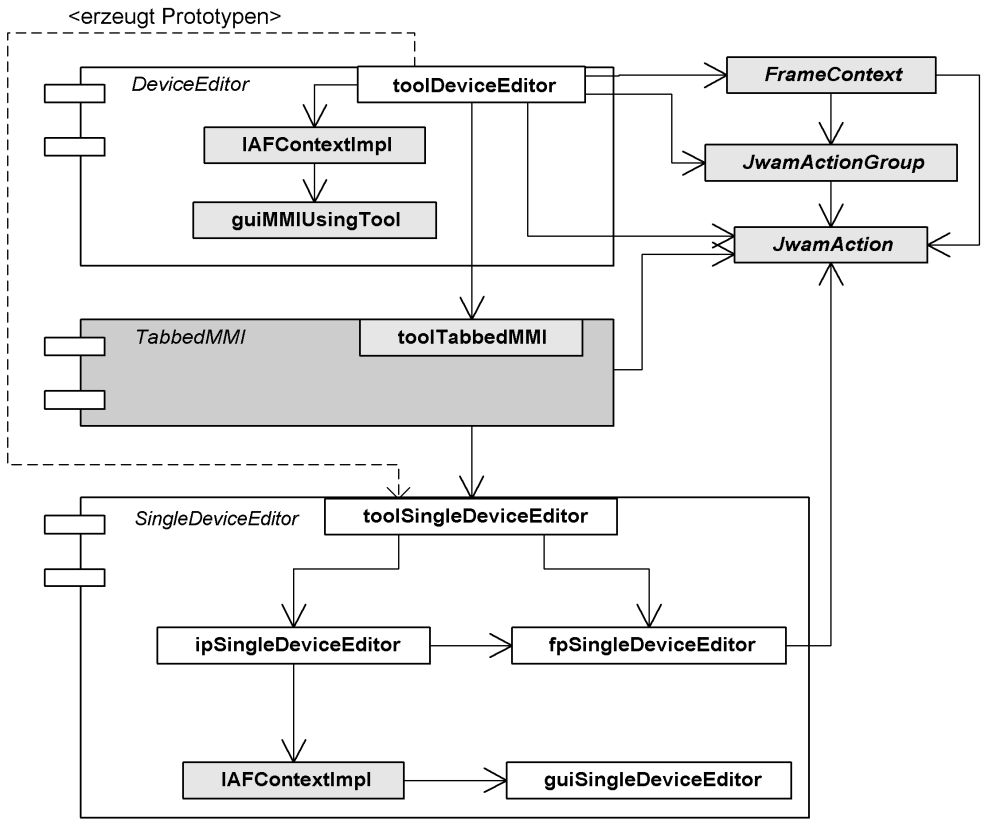

Abbildung 5.1 UML-Diagramm : Device-Editor

Auch in diesem UML-Diagramm sind nicht alle Beziehungen zu sehen. Nur die für das Zeigen der Realisierung wichtigen Verbindungen sind hier dargestellt.
Die drei beteiligten Komponenten **SingleDeviceEditor**, **TabbedMMI** und **DeviceEditor** sind hier graphisch hervorgehoben. Es ist zu sehen, dass von außerhalb einer Komponente keine Verbindungen in die Komponente selbst hineingehen. Nur die die Komponente vertretende Klasse, die jeweilige Tool-Klasse, wird von außen angesprochen. Es gibt nicht mehr, die in Abbildung 2.3 als die Komponentenkapsel brechend herausgestellten Verbindungen zwischen dem `GUIContext` des Kontextwerkzeuges und der Präsentation und dem `GUIContext` des Subwerkzeuges.
Während die Version des Werkzeuges Device-Editor, die ich in Abschnitt 2.4 zeigte ,aus dem Kontextwerkzeug Device-Editor und dessen Subwerkzeug Single-Device-Editor bestand, kommt in der in Abbildung 4.1 vorgestellten Version noch eine Werkzeugkomponente hinzu. Diese Werkzeugkomponente – genannt **TabbedMMI** – ist ein Bestandteil von JWAM. Diese Komponente wird mit einem Prototypen eines Werkzeuges bestückt – hier mit einem Single-Device-Editor – und startet dann für jedes an dieser Werkzeugkomponente gesetzte Material ein neues Exemplar des

als Prototypen gesetzten Werkzeuges, das dann unter einem neuen Reiter verfügbar ist. Diese Komponente macht es also leicht, ein Werkzeug zu konstruieren, dass mehrere Materialien zur gleichen Zeit bearbeiten kann. Das einzig notwendige an eigener Entwicklungsarbeit ist ein Werkzeug, das ein einzelnes Material bearbeiten kann – hier der Single-Device-Editor – und ein schlankes Hüllwerkzeug, dass das MMI-Werkzeug[18] konfiguriert.
Ich werde in den folgenden Abschnitten beschreiben, was im Vergleich zu der in Abschnitt 2.4 beschriebenen Version, neu und anders ist. Dabei werde ich die Realisierung des in Kapitel 4 erarbeiteten Konzeptes vorstellen.
Bei dieser Beschreibung werde ich nicht auf die innere Struktur der Werkzeugkomponente TabbedMMI eingehen. Diese Komponente wird hier wie eine Komponente eines anderen Entwicklers behandelt, deren Sourcecode nicht verfügbar ist. Dadurch wird auch die erreichte Verbesserung in Richtung Komponentenorientierung sichtbar werden.

5.2 Einbettung von Präsentationen

Der wichtigste Teil dieser Arbeit ist der, die Präsentationen von Subwerkzeugen in die der Kontextwerkzeuge einzubetten. Deshalb werde ich auch mit diesem Punkt anfangen.
Ich habe in Abschnitt 4.1.2 gefordert, dass Präsentationen von den Werkzeugkomponenten selbst, zu denen sie gehören, erzeugt werden und dass diese Werkzeugkomponenten diese Präsentationen dann zur Einbettung durch Kontextwerkzeuge anbieten. Ebenfalls im selben Abschnitt habe ich gefordert, dass alle Präsentationen einbettbare Präsentationen sind und dass die Umgebung für das Einbetten der Präsentationen von Kontextwerkzeugen zuständig sein soll. In Abschnitt 4.4 habe ich schließlich festgetellt, dass es für eine Unterstützung von Präsentationen in Teilen notwendig ist, dass jeder Teil einer Präsentation über einen Namen identifizierbar ist.
Zum Zweck der Identifizierbarkeit der Teile einer Präsentation, ist das Interface `ToolComponentViewPart` zu JWAM hinzugekommen.

```
public interface ToolComponentViewPart
{
  /**
   * @require result != null
   */
  public String viewPartName();
  public static final String MONO_VIEWPART_NAME = "mono";
}
```

Die Operation `viewPartName` gibt den Namen des Teils einer Präsentation zurück. Da es auch weiterhin möglich sein soll, die Präsentationen von Subwerkzeugen in einem Stück in Kontextwerkzeuge einzubetten, muss eine Zusammenstellung der Präsentationsteile einer Werkzeugkomponente auch als solche gekennzeichnet werden. Das Interface `ToolComponentViewPart` enthält die Konstante `MONO_VIEWPART_NAME`, die als Name für derartige Zusammenstellungen benutzt werden soll.
Damit ergiebt sich folgendes Aussehen für die Klasse `guiSingleDeviceEditor`:

```
public class guiSingleDeviceEditor extends JScrollPane
                                   implements ToolComponentViewPart
{
  public guiSingleDeviceEditor()
  {
```

[18] MMI steht hier in Anlehnung an MDI, was Multi Document Interface bedeutet, für Multi Material Interface

```
    int width;
    GridBagConstraints gcon=new GridBagConstraints();
    JPanel mainPanel = new JPanel();

    mainPanel.setLayout(new GridBagLayout());
    mainPanel.setBorder(BorderFactory.createEmptyBorder(12,12,12,12));

    //... weitere Zusammenstellung der Oberfläche
  }

  public String viewPartName()
  {
    return ToolComponentViewPart.MONO_VIEWPART_NAME;
  }
}
```

Diese Version der Klasse unterscheidet sich nur wenig von der vorigen Version. Das einzig neue ist, dass diese Klasse nun das Interface `ToolComponentViewPart` implementiert. Die Implementierung des Interfaces besteht darin, dass in der neuen Operation `viewPartName` die Konstante zurückgegeben wird, die für die Benennung von monolithischen Präsentationen gedacht ist.
Anders verhält es sich da mit der Präsentation des Werkzeuges Device-Editor.

```
public class guiMMIUsingTool extends pfJToolComponentViewPart
                               implements ToolComponentViewPartContainer
{
  {
    super();
    setLayout(new BorderLayout());
  }

  public boolean isDesignatedContainer(ToolComponentViewPart viewPart,
                                       String subToolName)
  {
    Contract.requireNotNull(viewPart, this, "viewPart");
    Contract.requireNotNull(subToolName, this, "subToolName");
    return viewPart.viewPartName().equalsIgnoreCase(
             ToolComponentViewPart.MONO_VIEWPART_NAME) &&
           subToolName.equalsIgnoreCase("mmi");
  }

  public void embed(ToolComponentViewPart viewPart, String subToolName)
  {
    Contract.requireNotNull(viewPart, this, "viewPart");
    Contract.requireNotNull(subToolName, this, "subToolName");
    Contract.require(isDesignatedContainer(viewPart, subToolName), this,
                     "isDesignatedContainer(viewPart, subToolName)");
    Contract.check(viewPart instanceof Component,
                   "viewPart is a java.awt.Component");
    add((Component) viewPart, BorderLayout.CENTER);
```

```
  }
}
```

Das Auffallendste an dieser Klasse ist zunächst ihr Name – `guiMMIUsingTool`. Das Werkzeug ist hier nur ein Hüllwerkzeug, dass das MMI-Werkzeug konfiguriert. Von seiner Oberfläche ist nur sehr wenig sichtbar, denn es handelt sich nur um ein Panel, in das das Subwerkzeug eingebettet wird. Da es Hüllwerkzeuge dieser Art häufig geben wird, wenn MMI-Werkzeuge zum Einsatz kommen, gibt es eine gui-Klasse für diese Werkzeuge, nämlich `guiMMIUsingTool`. Diese Klasse erbt, wie im obigen Codestück zu sehen, von `pfJToolComponentViewPart`, einer Klasse, die von `pfJPanel` erbt und die `ToolComponentViewPart` implementiert. `guiMMIUsingTool` enthält keine Implementation der Operation `viewPartName`. Das ist auch nicht nötig, da diese Operation in `pfJToolComponentViewPart` so implementiert ist, dass sie die Konstante `MONO_VIEWPART_NAME` des Interfaces `ToolComponentViewPart` zurückgibt.
Die Klasse `guiMMIUsingTool` implementiert ein Interface, dass ich bisher nocht nicht vorgestellt habe.

```
public interface ToolComponentViewPartContainer
{
  /**
   * @require viewPart != null
   * @require subToolName != null
   */
  public boolean isDesignatedContainer(ToolComponentViewPart viewPart,
                                       String subToolName);
  /**
   * @require viewPart != null
   * @require subToolName != null
   * @require isDesignatedContainerFor(viewPart, subToolName)
   */
  public void embed(ToolComponentViewPart viewPart, String subToolName);
}
```

`ToolComponentViewPartContainer` beschreibt den Umgang mit Objekten, die Präsentationen von Subwerkzeugen einbetten. Dazu enthält es eine Operation – `isDesignatedContainer` –, mit der das dieses Interface erfüllende Objekt gefragt werden kann, ob es der vorgesehene Container für einen bestimmten Teil der Präsentation eines Subwerkzeuges ist. Die andere Operation – `embed` – veranlasst in diesem Fall die Einbettung. `ToolComponentViewPart` ist hier also der Stecker und `ToolComponentViewPartContainer` die passende Steckdose.
Im Fall von `guiMMIUsingTool` ist das Interface derart implementiert, dass ein Viewpart mit dem Namen, der der Konstanten `MONO_VIEWPART_NAME` im Interface `ToolComponentViewPart` entspricht, eines Subwerkzeuges mmi zum Einbetten akzeptiert wird.
Wenn das Kontextwerkzeug auch eigene Elemente in der Oberfläche hat, wird die GUI-Klasse des Kontextwerkzeuges nicht `ToolComponentViewPartContainer` implementieren, sondern Objekte von Widgetklassen, die dieses Interface implementieren, in die eigene Präsentation aufnehmen. Ein Beispiel für eine solche Widgetklasse ist zum Beispiel `SwingToolComponentViewPartContainer`. Diese Klasse erbt von `JPanel` und implementiert `ToolComponentViewPartContainer`. Eine Verwendung dieser Klasse in einer gui-Klasse eines Kontextwerkzeuges könnte folgendermassen aussehen:

```
add(new SwingToolComponentViewPartContainer("lister", "browser"),
    BorderLayout.WEST);
add(new SwingToolComponentViewPartContainer("description", "browser"),
    BorderLayout.WEST);
add(new pfJButton("ok", "Ok"), BorderLayout.SOUTH);
```

In diesem Beispiel werden in eine Präsentation eines Kontextwerkzeuges nebeneinander Container für die Viewparts mit den Namen „lister“ und „description“ des Subwerkzeuges mit dem Namen „browser“ gepackt. Unter den beiden Container wird außerdem ein `pfJButton` mit der Aufschrift „Ok“ platziert.
Während in JWAM 1.5 die Präsentation eines Kontextwerkzeuges auch gleich die Präsentationen der Subwerkzeuge anordnete, wird hier nur der Platz für die Präsentationsteile eines Subwerkzeuges geschaffen und ausgewiesen. Es werden keine Annahmen darüber gemacht, was für Widgets letztendlich in diesen Containern eingebettet werden.
Nun taucht aber auch gleich die Frage nach dem Namen des Subwerkzeuges auf. Es ist nicht der Name, der in dem Werkzeug selbst definiert ist, denn dann könnten nicht mehrere Exemplare der gleichen Werkzeugkomponente als eingebettete Subwerkzeuge eines Kontextwerkzeuges verwendet werden. Bei diesem Namen handelt es sich um den Namen, den der Entwickler des Kontextwerkzeuges einem Subwerkzeug gibt. Dieser Name ist vom Entwickler des Kontextwerkzeuges frei wählbar. Das einzig wichtige daran ist, dass er auch den selben Namen beim Erzeugen des Werkzeuges in der Funktionskomponente angiebt.
Da das Werkzeug Device-Editor fast keine eigene Funktionalität hat, wurde es ohne die Trennung von Funktion und Interaktion entwickelt. Das heißt, dass sich das bißchen Funktionalität bei diesem Werkzeug auch in der Tool-Klasse befindet. Das führt auch dazu, dass das Subwerkzeug in der `doEquip`-Operation der Klasse `toolDeviceEditor` erzeugt und konfiguriert wird.

```
protected void doEquip()
{
  setContext(new IAFContextImpl(new guiMMIUsingTool()));
  _mmi = (toolTabbedMMI) createSubToolFunctionality(
            toolTabbedMMI.class, "mmi", null);
  _mmi.setToolPrototype(new toolSingleDeviceEditor());
  // ....
}
```

Die Klasse `guiMMIUsingTool`, die ich etwas früher vorgestellt habe, akzeptiert ein Viewpart eines Subwerkzeuges mit dem Namen „mmi“. Dieser Name wird, wie im obigen Codeausschnitt zu sehen, der Operation `createSubToolFunctionality` übergeben. Das führt schließlich dazu, dass in dieser Operation folgendes Codestück ausgeführt wird:

```
if (optionalEmbeddingName != null && hasContext() && tool.hasContext() &&
    context().hasDesignatedToolComponentViewPartContainers(tool.context(),
      optionalEmbeddingName))
{
  context().embed(tool.context(), optionalEmbeddingName);
  guiHasBeenPlaced = true;
}
```

Hier wird also am `GUIContext` des Werkzeuges Device-Editor die Operation `embed` aufgerufen. Als Parameter bekommt diese Operation das zum angegebenen Funktionalitätstyp – hier `toolTabbedMMI` – passend erzeugte Subwerkzeug und den Name, unter dem das Subwerkzeug in der GUI-Klasse bekannt ist. Der `GUIContext` sucht dann in den an ihm gesetzten Präsentationsteilen nach passenden Containern für die im `GUIContext` des Subwerkzeuges enthaltenen Präsentationsteile und bettet diese in ihre Container ein.

5.3 Ausführbare Aktionen

In Abschnitt 4.3 habe ich gefordert, dass Werkzeuge, die an ihnen ausfühbaren Aktionen explizit machen und nach außen anbieten. Das ist nun durch die Verwendung des Interfaces `JwamAction` und dessen Implementierung `JwamActionImpl` möglich. Um das Zusammenstellen von Aktionen zu Gruppen, die dann später auf Symbolleisten und Menüs abgebildet werden, zu ermöglichen stehen diese beiden nicht alleine. Alle zum Bereich der ausführbaren Aktionen gehörenden Interfaces und Klassen und deren Beziehungen zueinander sind in der Abbildung 5.2 zu sehen.

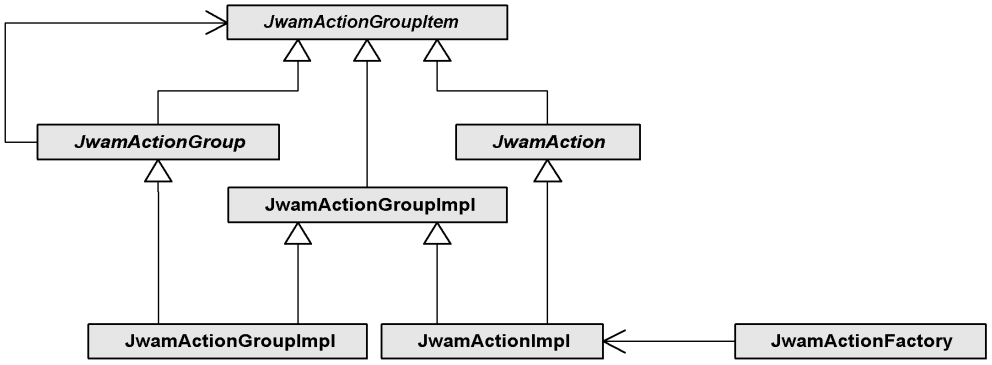

Abbildung 5.2 Die Klassen für ausführbare Aktionen

Die Interfaces in dieser Gruppe bieten ausschließlich einen sondierenden Zugriff an. Die Interfaces `JwamActionGroupItem` und `JwamAction` sehen zum Beispiel folgendermassen aus:

```
public interface JwamActionGroupItem extends Serializable
{
  /**
   * @ensure result != null
   */
  public dvIdentificator id();

  public boolean isEnabled();

  /**
   * @return the text of the item.
   * @ensure result != null
   */
  public String text();
}
```

```
public interface JwamAction extends JwamActionGroupItem
{
  public void perform();

  public boolean hasIconURL();

  /**
   * @require hasIconURL()
   * @ensure result != null
   */
  public URL iconURL();

  /**
   * @ensure result != null
   */
  public String actionType();
}
```

Damit ist sichergestellt, dass ein Kontextwerkzeug nicht verändernd auf die Aktionen eines Subwerkzeuges zugreift. Die Aktionen eines Werkzeuges gehören diesem Werkzeug und dieses Werkzeug selbst darf diese Aktionen auch verändern. Die Implementationen dieser Interfaces (zum Beispiel `JwamActionImpl`) bieten Möglichkeiten, den Zustand dieser Objekte zu ändern.
Es wäre natürlich auch möglich gewesen, ein Interface `JwamActionModifiable` zu definieren, das dann von `JwamActionImpl` implementiert werden könnte. Der Nutzen eines solchen Interfaces wäre jedoch sehr gering. An keiner Stelle im JWAM wird ein verändernder Zugriff auf die Aktionen benötigt. Dieser verändernde Zugriff wird erst in den Werkzeugen benötigt. Ob dort dann `JwamActionImpl` benutzt wird oder eine andere selbst geschriebene Klasse, ist belanglos. Ein solches Interface würde nur die Komplexität erhöhen.
Die Klasse `JwamActionFactory` steht etwas abseits der anderen. Diese Klasse kann einige Standardaktionen wie zum Beispiele jene zum Schließen des Werkzeuges und zum Laden eines Materials erzeugen.
Das Anbieten von ausführbaren Aktionen ist eine fachliche Angelegenheit. Deshalb gehört diese Aufgabe zur Funktionalität eines Werkzeuges. Die Klassen `ToolFunctionalityImpl` und `ToolMonoImpl` bieten von sich aus schon eine Aktion zum Schließen des Werkzeuges an. Alle weiteren benötigten Aktionen müssen von Klassen, die von diesen erben, angeboten werden.
In unserem Beispiel werden die Aktionen des Werkzeuges Single-Device-Editor in der Klasse `fpSingleDeviceEditor` verwaltet.

```
public class fpSingleDeviceEditor extends ToolFunctionalityImpl
{
  public fpSingleDeviceEditor(RequestHandler handler)
  {
    super(handler, "Single Device-Editor");

    _storeAction = JwamActionFactory.createdStoreAction();
    _storeAction.attachActionPerformer(
      new ActionPerformer()
      {
        public void run()
        {
```

```
          storingRequestedEvent().announce();
        }
      });
  }

  public JwamAction[] actions()
  {
    return new JwamAction[]{storeAction(), exitAction()};
  }

  public JwamAction storeAction()
  {
    return _storeAction;
  }

  private JwamActionImpl _storeAction;
}
```

Das obige Codestück zeigt nur den Teil der Klasse `fpSingleDeviceEditor`, der sich mit den ausführbaren Aktionen befasst.
Der Umgang mit Aktionen ist recht einfach. Im Konstruktor wird die `JwamActionFactory` angewiesen, eine Speichern-Aktion zu erzeugen. An dieser Aktion wird dann ein `ActionPerformer` gesetzt. Das Interface `ActionPerformer` erweitert die Interfaces `Runnable` und `Serializable`. Die Einführung dieses Interfaces wurde durch die Tatsache erzwungen, dass sich Objekte von anonymen Innerclasses, die `Runnable` implementieren, nicht wegserialisieren lassen. Da Aktionen und damit auch alles was an ihnen hängt manchmal gespeichert werden müssen, war `Runnable` alleine nicht verwendbar. Ein `ActionPerformer` wird ausgeführt, wenn an der Aktion, zu der er gehört, `perform` aufgerufen wird.
Von der Operation `actions` werden alle Aktionen, die von diesem Werkzeug angeboten werden zurückgegeben. Diese Operation ist in `ToolFunctionality` definiert und muss implementiert werden, wenn mehr als nur die Beenden-Aktion angeboten werden soll.
Die Speichern-Aktion ist hier immer verfügbar. Es wäre aber auch denkbar, dass diese Aktion nur dann verfügbar ist, wenn etwas am Material verändert wurde. Für diesen Fall müsste im Konstruktor eine weitere Zeile dazukommen.

```
_storeAction.setEnable(false);
```

Damit wäre diese Aktion initial nicht aktiviert und zum Beispiel in einem Menü ausgegraut. An einer geeigneten Stelle würden die folgenden Zeilen die Aktion wieder aktivieren:

```
_storeAction.setEnable(false);
actionsChangedEvent().announce();
```

`actionsChangedEvent` ist eine Operation in der Oberklasse, die das Ereignis zurückgibt, das verkündet werden muss, wenn sich irgendetwas an den Aktionen geändert hat. Jemand, der sich für die Aktionen eines Werkzeuges (zum Beispiel ein Kontextwerkzeug) interessiert, wird dieses Ereignis empfangen und entsprechend reagieren.

5.4 Beziehung zwischen `JwamAction` und `ifActivator`

Seit ich zum ersten Mal die Idee der ausführbaren Aktionen erwähnte, bekam ich oft die Frage zu hören: „Ist eine `JwamAction` nicht eigentlich ein `ifActivator`?“.

Eine gewisse Verwandtschaft ist nicht zu leugnen. Auch das Aktivieren eines Aktivators löst auf seine Weise eine Aktion aus. Ein `ifActivator` ist aber etwas, dass von der Präsentation und von der Interaktionskomponente benutzt wird. Es ist also auf die Behandlung von Interaktionen des Benutzers ausgerichtet. Manchmal führt das Aktivieren eines Aktivators zum Auslösen einer Aktion an der Funktionskomponente. Das passiert aber nur, wenn die Interaktionskomponente einen entsprechenden Aufruf an der Funktionskomponente durchführt. Hinter einem Aktivator verbirgt sich auch immer eine Präsentationsform, die von der Interaktionskomponente bei Bedarf ermittelt werden kann.
Das Ausführen einer `JwamAction` führt immer zu einer Ausführung einer Aktion an der Funktionskomponente. Es gibt keine Interaktionskomponente, die das Ausführen interpretiert und in Befehle an die Funktionalität umwandelt. Das Auslösen einer `JwamAction` wird nicht immer direkt durch eine Interaktion eines Benutzers ausgelöst. Eine `JwamAction` kann auch durch ein Kontextwerkzeug an einem Subwerkzeug ausgelöst werden. Während ein `ifActivator` immer zu einer bestimmten Präsentationsform gehört – meistens zu einem Knopf –, kann eine `JwamAction` gleichzeitig zum Beispiel an einem Knopf und an einem Menüeintrag angemeldet sein. Gleichzeitig ist sie auch dem Kontextwerkzeug bekannt. Während eine Interaktionsform nur der sie realisierenden Präsentationsform und der Interaktionskomponente bekannt ist, kann eine `JwamAction` beliebig vielen bekannt sein.
Das Anbieten einer `JwamAction` erweitert die Dienstleistungen, die eine Werkzeugkomponente erbringen kann. Das Schaffen eines `ifActivator` ist hingegen von der Präsentation des Werkzeuges abhängig.
Deshalb denke ich, dass `ifActivator` und `JwamAction` bei einer gewissen Verwandtschaft doch zwei nebeneinander stehende Konzepte sind.

5.5 Werkzeugabschluss

In Abschnitt 4.1.2 habe ich gefordert, dass alle Präsentationen einbettbare Präsentationen sind. Dort habe ich auch gefordert, dass die Umgebung die Fenster für Kontextwerkzeuge erzeugt. Außerdem habe ich in Abschnitt 4.2 gefordert, dass Werkzeuge im Falle, dass sie in eigenen Fenstern laufen, einen eingeschränkten Zugriff auf diese Fenster zum Zweck der Bestückung von Menüs und Symbolleisten erhalten. Die Realisierung dieser Anforderungen werde ich nun vorstellen.
Wenn die Umgebung ein Werkzeug startet, wird, nachdem das Material am Werkzeug gesetzt wurde und weitere für den Start eines Werkzeuges wichtige Dinge erledigt wurden, geprüft, ob die Präsentation dieses Werkzeuges einbettbar ist. Ist das der Fall, so wird am `ToolComponentManager` die Operation `equipWithFrame` aufgerufen.

```
public void equipWithFrame(Tool tool)
{
  Contract.requireNotNull(tool, this, "tool");
  Contract.require(tool.canRunWithFrame(), this,
                   "tool <" + tool.name() +"> can run with a frame");
  Contract.require(!tool.hasFrameContext(), this,
                   "tool <" + tool.name() +"> has no frame context");
  tool.equipWithFrame(frameContext(tool));
  Contract.ensure(tool.hasFrameContext(), this,
                   "tool <" + tool.name() +"> has a frame context");
}
```

```
  public FrameContext frameContext(Tool tool)
  {
    Contract.requireNotNull(tool, this, "tool");

    FrameContext result = null;
    if(hasFrameContextProvider())
    {
      result = _frameContextProvider.frameContext(tool);
    }
    else
    {
      result = new SwingFrameContext(new JFrame());
    }

    Contract.ensureNotNull(result, this, "result");

    return result;
  }
```

In `equipWithFrame` ruft der `ToolComponentManger` an dem Werkzeug, das mit einem Fenster ausgestattet werden soll, die Operation `equipWithFrame` auf. Diese Operation ist in `Tool` definiert und in `ToolImpl` implementiert. Diese Operation erwartet als Parameter ein Objekt vom Typ `FrameContext`. Dieses Interface spielt hier eine ähnliche Rolle, wie `GUIContext` für die Präsentationen von Werkzeugen. `FrameContext` definiert den eingeschränkten Zugriff, den ein Werkzeug auf sein Fenster haben kann.

```
public interface FrameContext extends Serializable
{
  public void embed(final GUIContext context);
  public void close();
  public void attachExitAction(final JwamAction action);
  public void setMenuBar(final JwamActionGroup actionGroup);
  public void setMenuBarVisible(boolean visible);
  public boolean isMenuBarVisible();
  public void useIconsInMenus(final boolean useIcons);
  public boolean isUsingIconsInMenus();
  public void useTextsInMenus(final boolean useTexts);
  public boolean isUsingTextsInMenus();
  public void setToolBar(final JwamActionGroup actionGroup);
  public void setToolBarVisible(boolean visible);
  public boolean isToolBarVisible();
  public void useIconsInToolBars(final boolean useIcons);
  public boolean isUsingIconsInToolBars();
  public void useTextsInToolBars(final boolean useTexts);
  public boolean isUsingTextsInToolBars();
  public void setCommandButtonPanel(final JwamActionGroup actionGroup);
  public void setCommandButtonPanelVisible(boolean visible);
  public boolean isCommandButtonPanelVisible();
  public void useIconsInCommandButtonPanel(final boolean useIcons);
  public boolean isUsingIconsInCommandButtonPanel();
```

```
  public void useTextsInCommandButtonPanel(final boolean useTexts);
  public boolean isUsingTextsInCommandButtonPanel();
  public void setStatusBarVisible(boolean visible);
  public boolean isStatusBarVisible();
  public void addTextDisplayToStatusBar(final String textDisplayName,
                                        final int columns, boolean flexible);
  public void removeTextDisplayFromStatusBar(final String textDisplayName);
  public boolean hasTextDisplay(final String textDisplayName);
  public void updateTextDisplay(final String textDisplayName,
                                final String updatedContent);
  public void setFrameTitle(final String title);
  public void setFrameIcon(final Image icon);
  public void updateAction(final JwamAction action);
  public void updateActions(final JwamAction[] actions);
  public void packFrame();
  public void moveFrameToFront();
  public void setFrameSize(int width, int height);
  public void setFrameResizable(boolean resizable);
  public void show();
  public void hide();
}
```

In der obige Aufstellung der Schnittstellt von `FrameContext` fehlen die Angaben zu den Vor- und Nachbedingungen der einzelnen Operationen. Somit ist die obige Aufstellung nicht die vollständige Definition der Schnittstelle sondern nur ein Überblick über diese.
Beim Betrachten dieser Aufstellung, fällt sehr schnell auf, dass hier keine Abhängigkeiten zu einem bestimmten GUI-Toolkit bestehen. Dieses Interface kann zum Beispiel für das Swing-Toolkit, aber auch für das AWT oder andere Toolkits implementiert werden. In JWAM ist `SwingFrameContext` die zur Zeit einzige verfügbare Implementation.
Damit eine andere `FrameContext`-Implementation oder eine andere Art von Fenstern verwendet wird, muss am `ToolComponentManager` ein `FrameContextProvider` gesetzt werden.

```
public interface FrameContextProvider
{
  public FrameContext frameContext(Tool tool);
}
```

Wie weiter oben zu sehen wird ein solcher `FrameContextProvider` in der Operation `frameContext` des `ToolComponentMangers` gerufen, um einen neuen `FrameContext` für ein bestimmtes Werkzeug zu erhalten. Der `FrameContextProvider` kann dann entscheiden, was für ein Fenster er für das bestimmte Werkzeug erzeugt und in welchen `FrameContext` er dieses Fenster verpackt.
Die Desktopkomponente von JWAM meldet zum Beispiel einen eigenen `FrameContextProvider` am `ToolComponentManager` an. Dieser sieht wie folgt aus:

```
protected class FrameContextProvider implements
   de.jwam.handling.toolconstruction.basicconstruction.FrameContextProvider
{

  /**
   * @require tool != null
```

```
   * @ensure result != null
   */
  public FrameContext frameContext(Tool tool)
  {
    Contract.requireNotNull(tool, this, "Tool");
    FrameContext result = null;

    if (tool == _desktopTool)
    {
      result = new SwingFrameContext(new pfJFrame());
    }
    else
    {
      dvIdentificator parentToolID = null;

      JInternalFrame iFrame = new JInternalFrame("", true, true, true, true);
      ((guiDesktopInternalFrames) _desktopTool.context().monoRoot()).
                                                  addInternalFrame(iFrame);
        result = new SwingFrameContext(iFrame);
    }

    Contract.ensureNotNull(result, this, "result");
    return result;
  }
}
```

Wenn das Werkzeug, das ein Fenster bekommen soll, der JWAM-Desktop selbst ist, bekommt es einen `pfJFrame` als Fenster. Anderenfalls bekommt es einen `JInternalFrame` als Fenster. In Abbildung 5.3 ist der JWAM-Desktop ohne die Verwendung des eigenen `FrameContextProvider`s und damit ohne die Verwendung von `JInternalFrame`s für die Werkzeuge zu sehen.
Abbildung 5.4 zeigt den Desktop mit der Verwendung des eigenen `FrameContextProvider`s und damit mit der Verwendung von `JInternalFrame`s für die Werkzeuge. Dafür war keine spezielle Anpassung des Werkzeuges Device-Editor notwendig. Der einzige Unterschied ist hier die Verwendung des eignen `FrameContextProvider`s.
Ein letzter Punkt muss noch geklärt werden, um das Thema „Werkzeugabschluß“ abzuschließen. Abbildung 5.4 zeigt einen Device-Editor mit einem Menü. Ohne weiteres Zutun, erscheint dieses Menü jedoch nicht von selbst. Dafür ist eine Erweiterung in der Klasse `toolDeviceEditor` notwendig. Weiter oben habe ich gesagt, dass der `ToolComponentManger` beim Ausstatten eines Werkzeuges mit einem Fenster an diesem die Operation `equipWithFrame` aufruft, die in `ToolImpl` implementiert ist.

```
  public void equipWithFrame(FrameContext frameContext)
  {
    Contract.requireNotNull(frameContext, this, "frameContext");
    Contract.require(canRunWithFrame(), this,
                     "tool <" + name() + "> can run with a frame");

    if (hasContext() && !(context().monoRoot() instanceof Window))
    {
      if(!hasFrameContext())
```

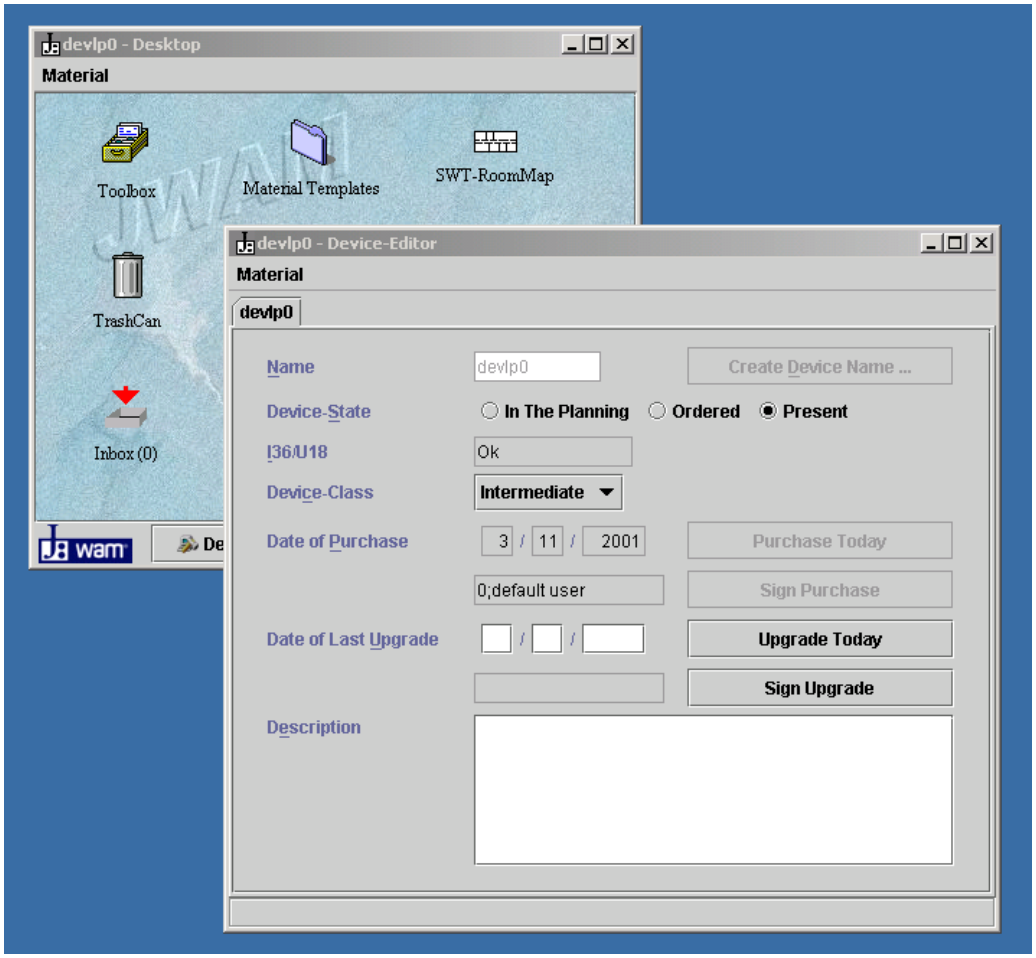

Abbildung 5.3 JWAM-Desktop ohne die Verwendung von `JInternalFrames`

```
{
  setFrameContext(frameContext);
}
frameContext().embed(context());
frameContext().attachExitAction(functionality().exitAction());
String title = "";
if(functionality().materials().length > 0 &&
   !VoidThing.class.isInstance(functionality().materials()[0]))
{
  title += functionality().materials()[0].name() + " - ";
}
title += name();
frameContext().setFrameTitle(title);
frameContext().setStatusBarVisible(true);
frameContext().addTextDisplayToStatusBar("stateDisplay", 20, true);
functionality().stateChangedEvent().register(
  new EventReaction()
```

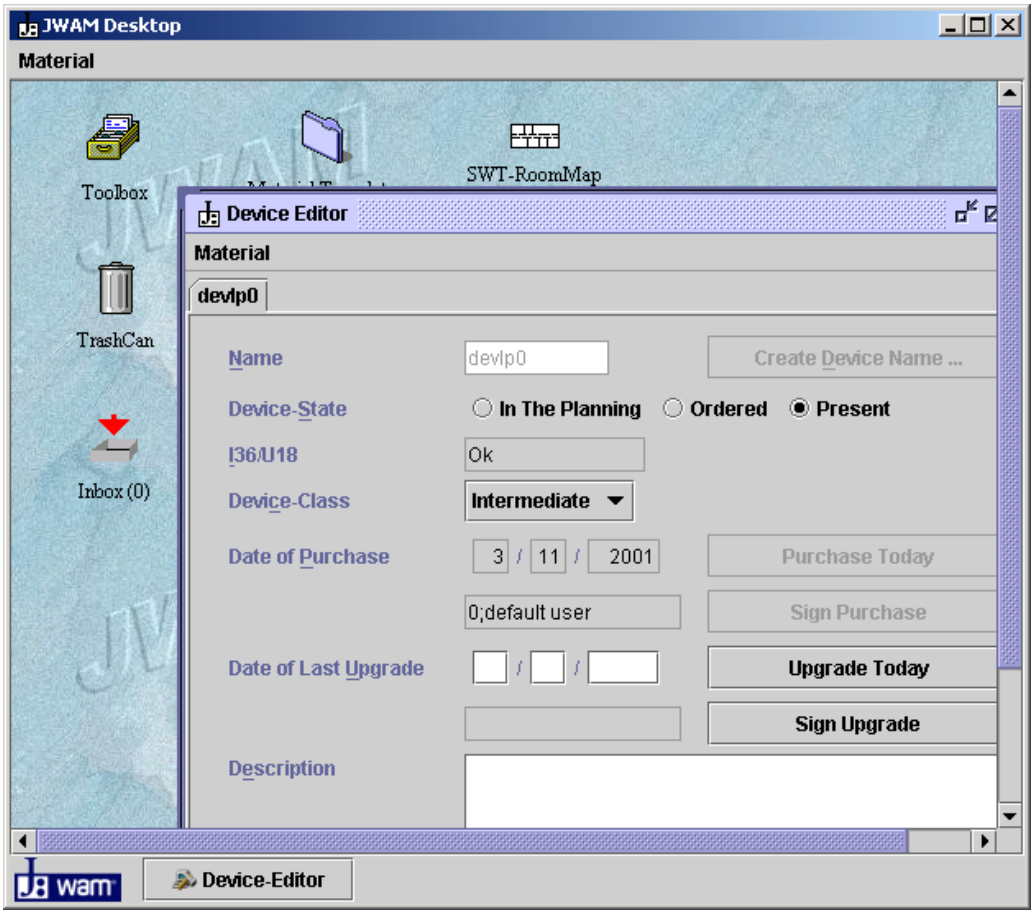

Abbildung 5.4 JWAM-Desktop mit der Verwendung von `JInternalFrames`

```
    {
      public void update (evtObject evt)
      {
        if( isEquipped() && frameContext().hasTextDisplay("stateDisplay"))
        {
          frameContext().updateTextDisplay("stateDisplay",
                                           functionality().stateString());
        }
      }
    });

  _updateTitle = new EventReaction()
  {
    public void update (evtObject evt)
    {
      String title2 = "";
      if(functionality().materials().length > 0 &&
         !VoidThing.class.isInstance(functionality().materials()[0]))
      {
```

```
          title2 += functionality().materials()[0].name() + " - ";
        }
        title2 += name();
        frameContext().setFrameTitle(title2);
      }
    };

    functionality().materialUpdatedEvent().register(_updateTitle);
    _registeredUpdateTitle = true;
  }

  Contract.ensure(hasFrameContext(), this,
                  "tool <" + name() + "> has a frame context");
}
```

Diese Implementation übernimmt einige Aufgaben, die sich generisch für alle Wekzeuge erledigen lassen. Hier wird die eigene Präsentation in das Fenster eingebettet, die ständige Aktualisierung der Statuszeile und Titelzeile in die Wege geleitet und die Beenden-Aktion am `FrameContext` angemeldet. Das letztere führt dazu, dass die Beenden-Aktion bei einem Klick auf das „x“ in der oberen rechten Ecke des Fensters ausgeführt wird[19]. Es werden jedoch keine Menüs erzeugt. Dazu muss die Operation `equipWithFrame` überschrieben werden.

```
public void equipWithFrame(FrameContext frameContext)
{
  Contract.requireNotNull(frameContext, this, "frameContext");
  Contract.require(canRunWithFrame(), this,
                   "tool <" + name() + "> can run with a frame");

  super.equipWithFrame(frameContext);

  JwamActionGroupImpl mainActionGroup = new JwamActionGroupImpl("main");
  JwamActionGroupImpl materialActionGroup  = new JwamActionGroupImpl("Material");
  materialActionGroup.addItem(newAction());
  materialActionGroup.addItem(storeAction());
  materialActionGroup.addItem(closeAction());
  materialActionGroup.addItem(exitAction());
  mainActionGroup.addItem(materialActionGroup);

  frameContext().setMenuBar(mainActionGroup);
  frameContext().setMenuBarVisible(true);

  // workaround for a Visual Age for Java 3.5 bug
  final FrameContext fContext = frameContext();
  final ToolFunctionality toolFunc = functionality();

  functionality().actionsChangedEvent().register(
    new EventReaction()
```

[19] Bei anderen Betriebssystemen als Windows oder Linux mit KDE kann dieser Knopf anders aussehen oder an einer anderen Stelle angebracht sein

```
    {
      public void update (evtObject evt)
      {
        for(int i = 0; i < toolFunc.actions().length; i++)
        {
          fContext.updateAction(toolFunc.actions()[i]);
        }
      }
    });

  frameContext().setFrameTitle("Device Editor");

  Contract.ensure(hasFrameContext(), this, "This tool has a frame context");
}
```

Hier wird zunächst die Implementation der Oberklasse aufgerufen und hiernach aus den einzelnen Aktionen eine `JwamActionGroupImpl` zusemmengestellt, die dann am `FrameContext` als Menüleiste gesetzt wird. Außerdem sorgt diese Implementation, dass bei jeder Verkündigung des Ereignisses `actionsChangedEvent` alle Aktionen aktualisiert werden.

5.6 Zusammenfassung

Ich habe in diesem Kapitel die Realisierung des im Kapitel 4 vorgestellten Konzeptes gezeigt.
Ich habe anhand des Beispieles Device-Editor gezeigt, wie ein Werkzeug mit Hilfe dieser Realisierung konstruiert werden kann. Ich habe am Anfang des Kapitels in Abschnitt 5.1 ein UML-Diagramm der neuen Version dieses Werkzeuges gezeigt, an dem die stärkere Komponentenorientierung ersichtlich wurde. Daraufhin habe ich in Abschnitt 5.2 gezeigt, wie `ToolComponentViewPart` und `ToolComponentViewPartContainer` zusammen die Einbettung von Subwerkzeugen in Kontextwerkzeuge vereinfachen, ohne dass die Komponentenkapsel des Subwerkzeuges verletzt wird. In Abschnitt 5.3 habe ich `JwamAction` und die zu diesem Bereich gehörenden Interfaces und Klassen vorgestellt, wobei ich anhand des Beispiels die Verwendung von ausführbaren Aktionen gezeigt habe. In Abschnitt 5.4 bin ich auf die oft gestellte Frage nach der Beziehung zwischen `ifActivator` und `JwamAction` eingegangen, wobei ersichtlich wurde, dass es sich bei beiden um verwandte aber doch unterschiedliche Konzepte handelt. Ich habe dieses Kapitel mit der Beschreibung der Realisierung des Werkzeugabschlußes abgeschlossen.

6 Zusammenfassung und Ausblick

In diesem Kapitel werde ich einen kurzen Rückblick auf die Arbeit geben und anschließend einen Ausblick auf das geben, was noch kommen könnte und was kommen sollte

6.1 Rückblick

Ich habe in dieser Arbeit zunächst in Kapitel 2 die Werkzeugkonstruktion nach WAM und deren Realisierung in JWAM 1.5 beschrieben. Ich bin dort auf den Teil eingegangen, der sich mit den Präsentationen von Werkzeugen befasst und habe einige für diesen Bereich wichtige Entwurfsmuster aus [Züllighoven 98] kurz vorgestellt.
In Kapitel 3 habe ich einige bereits existierende Konzepte vorgestellt, die sich mit der Einbettung von Subwerkzeugen in Kontextwerkzeugen befassen. Ich habe zwei Gruppen von Ansätzen in diesem Gebiet entdeckt. Die aufgabenorientierten Ansätze bringen Komponenten hervor, die zur Erledigung einer bestimmten Aufgabe gedacht sind. Die dokumentenorientierten Ansätze bringen Komponenten hervor, die zur Bearbeitung bestimmter Dokumenttypen vorgesehen sind. Ich habe hier festgestellt, dass WAM in die Gruppe der aufgabenorientierten Ansätze gehört.
In Kapitel 4 habe ich ein Konzept zur Lösung des mir gestellten Problemes vorgestellt, dass sich auf einigen Anforderungen stützt. Die Realisierung und damit auch die Erfüllung dieser Anforderungen habe ich in Kapitel 5 gezeigt.

6.2 Ausblick

Obwohl ich denke, einen nicht kleinen Beitrag zur komponentenorientierten Werkzeugekonstruktion in JWAM geleistet zu haben, ist diese noch nicht vollkommen.
Ein noch nicht gelöstes Problem der Werkzeugkonstruktion ist hängt mit der Tatsache zusammen, dass Werkzeuge sich über ihre Funktionalität definieren. Immer, wenn ein neues Subwerkzeug benötigt wird, wird das passende über den Typ der Funktionskomponente, die erwartet wird, ermittelt. Bei monolithischen Werkzeugen ohne die Trennung von Funktion und Interaktion, ist das immer die Tool-Klasse. Soll nun dieses Werkzeug zu einer späteren Zeit in eines umgewandelt werden, bei dem eine Trennung von Funktion und Interaktion besteht, ändert sich auch der Typ der Funktionalität, der von potentiellen Kontextwerkzeugen angeben wird. Die Benutzer einer Werkzeugkomponente bemerken zwangsläufig die Veränderung der eigentlich inneren Struktur der Komponente und haben dadurch einen gewissen Migrationsaufwand. Die Frage, die einer Beantwortung harrt, ist nun, wie sich eine solche Situation vermeiden lässt und ob es die Aufgabe eines Entwicklers ist, immer eine Funktionalitätsschnittstelle zu definieren, die dann im ersten Fall von der Tool-Klasse und im zweiten Teil von der Funktionskomponente implementiert wird, oder ob ein generischer Mechanismus im Rahmenwerk das Vereinfachen kann.
Ein zweiter wichtiger Punkt ist die Konfigurierbarkeit von Werkzeugkomponenten. So ist es bei einigen sinnvoll, sie mal als Editor und ein anderes Mal als Anzeiger für ein Material zu benutzen. Zur Zeit ist es dem Entwickler selbst überlassen, Operationen anzubieten, über die das Werkzeug konfiguriert werden kann. Es wäre jedoch eine standardisierte Art wünschenswert, über die eine solche Konfiguration durchgeführt werden könnte. Gäbe es da einen Standard, könnten Entwickler, die eine Werkzeugkomponente benutzen, schneller herausfinden, was an einer Werkzeugkomponente konfigurierbar ist.
Als einen letzen Schritt weiter in Richtung komponentenorientierte Werkzeugkonstruktion sehe ich die Schaffung eines Baukastensystems, in dem Entwickler interaktiv Werkzeugkomponenten zu Werkzeugen zusammenstecken könnten. Vielleicht wäre eine Realisierung in Zusammenhang mit JavaBeans möglich.
Diese Schritte würden meiner Meinung nach JWAM schließendlich zu einer perfekt komponentenorientierten Wekzeugkonstruktion verhelfen.

Bibliographie

[3Amigos 99] Grady Booch,Jim Rumbaugh, Ivar Jacobsen: *Das UML-Benutzerhandbuch.* Bonn: Addison Wesley Longman Inc., 1999

[Bleek 99] Wolf-Gideon Bleek, Thorsten Görtz, Carola Lilienthal, Martin Lippert, Stefan Roock, Wolfgang Strunk, Ulfert Weiss, Henning Wolf: *Interaktionsformen zur flexiblen Anbindung von Fenstersystemen.* Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1999

[Blish 00] James Blish *Cities In Flight.* Woodstock, New York: The Overlook Press, 2000

[Burdack 97] Henning Burdack: *OpenLDoc – Hauptseminar „Frameworktechnologien und ihr Einsatz bei Groupware“.* Technische Universität München, Fachbereich Informatik, Juni 1997

[Felski 00] Johanna Felski, Erdal Özkan: *Studienarbeit : Rahmenwerkbasierte Werkzeugkomponenten am Beispiel des JWAM-Rahmenwerks.* Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, August 2000

[Fröse 99] Frank Fröse: *Diplomarbeit: Komponentenorientierte Werkzeugkonstruktion.* Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, November 1999

[Gamma 98] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Entwurfsmuster.* Bonn: Addison Wesley Longman Inc., 1998

[Lippert 97] Martin Lippert: *Studienarbeit: Konzeption eines GUI-Frameworks in Java nach der WAM-Metapher.* Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, November 1997

[Lippert 99] Martin Lippert: *Diplomarbeit: Die Desktop-Metapher in Systemen nach dem Werkzeug-und-Material-Ansatz.* Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, Oktober 1999

[Meyer 94] Hanns Martin Meyer, Karl Obermayr: *Objekte integrieren mit OLE 2 – Microsofts Basistechnologie für objektorientierte Architektur.* Berlin, Heidelberg, New York: Springer-Verlag, 1994

[Schmitt 96] Dr. Hans-Jochen Schmitt *Bewegliche Ziele – ActiveX: Microsofts Antwort auf Java.* c't Magazin für Computertechnik 12/1996, ct'ROM 1996

[Schürig 97] Michael Schürig, Martin Schwarz, Dr. Jörn Loviscach *Teilweise – OpenDoc zwischen den Fronten.* c't Magazin für Computertechnik 3/1997, ct'ROM 1997

[Sweet 00] David Sweet et al. *KDE 2.0 Development* SAMS Publishing, voraussichtlich Oktober 2000

[Sturm 98] Thorsten Sturm: *Studienarbeit: Interaktionsformen für objektorientierte, reaktive Softwaresysteme unter Berücksichtigung der Werkzeug-Material-Metapher.* Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, Dezember 1998

[Szyperski 98] Clemens Szyperski: *Component Software – Beyond Object-Oriented Programming.* New York: Addison Wesley, 1998

[Weis 98] Torben Weis, Bernd Wuebben: *KDE KOM/Openparts.* developer.kde.org, August 1998

[Züllighoven 98] Züllighoven et al.: *Das objektorientierte Konstruktionshandbuch.* Heidelberg: dpunkt Verlag, 1998