Interaktionsformen zur flexiblen Anbindung von Fenstersystemen

Wolf-Gideon Bleek, Thorsten Görtz, Carola Lilienthal, Martin Lippert,
Stefan Roock, Wolfgang Strunk, Ulfert Weiss, Henning Wolf

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik
1999

Danksagung

Wir möchten allen danken, die bei der Ausarbeitung der hier vorgestellten Konzepte mitgewirkt haben. Namentlich sind dies:

- Frank Fröse
- Holger Koschek
- Keno Hamer.

Inhaltsverzeichnis

1 EINLEITUNG UND ZIELE DER ARBEIT ... 1

2 MOTIVATION ... 2

3 BEGRIFFSBILDUNG ... 5

4 PROBLEME KONVENTIONELLER ANBINDUNG AN BENUTZUNGSOBERFLÄCHEN ... 8

5 INTERAKTIONSFORMEN ZUR FLEXIBLEN ANBINDUNG AN BENUTZUNGSOBERFLÄCHEN ... 11

5.1 INTERAKTIONSFORMEN ... 11
5.2 PRÄSENTATIONSFORMEN ... 12

6 BESCHREIBUNG KONKRETER INTERAKTIONSFORMEN ... 13

6.1 KONSOLIDIERTE INTERAKTIONSFORMEN ... 13
6.1.1 IAFBase ... 13
6.1.2 IAFActivator ... 13
6.1.3 IAF1FromNSelection ... 14
6.1.4 IAFStatic1FromNSelection ... 14
6.1.5 IAFDynamic1FromNSelection ... 15
6.1.6 IAFMFromNSelection ... 15
6.1.7 IAFStaticMFromNSelection ... 16
6.1.8 IAFDynamicMFromNSelection ... 16
6.1.9 IAFFillIn ... 17
6.1.10 IAFStringFillIn ... 17
6.2 KOMPONENTEN ZUR KOMPOSITION VON INTERAKTIONSFORMEN ... 17
6.2.1 GUI-Manager ... 17
6.2.2 IAFsContext ... 18
6.3 WEITERGEHENDE INTERAKTIONSFORMEN ... 18
6.3.1 IAFTable ... 18

7 MODALE FENSTER ... 20

7.1 IAFMESSAGEREQUEST ... 21
7.2 IAFCHOICEREQUEST ... 21

8 SPEZIFIKATION DER PÄSENTATION ... 23

8.1 INTERAKTIONS-, PRÄSENTATIONSFORMEN UND INTERFACE BUILDER ... 23
8.2 RESSOURCE-DATEIEN UND PERSISTENTE OBERFLÄCHEN-WIDGETS ... 23
8.3 FESTCODIERTE OBERFLÄCHEN ... 25
8.4 GENERIERTER OBERFLÄCHEN-QUELLCODE ... 25
8.5 DYNAMISCHE OBERFLÄCHEN ... 25

9 IMPLEMENTATIONSANSÄTZE ... 27

9.1 REALISIERUNG IN C++ ... 27
9.1.1 Interaktionsformen ... 27
9.1.2 Präsentationsformen ... 27
9.1.3 Diskussion der Architektur von IAF und PF ... 29
9.1.4 Dynamik zur Laufzeit ... 31
9.1.5 Präsentationsformen im Kontext eines Werkzeuges ... 32
9.2 REALISIERUNG DES GUI-RAHMENWERKES IN JAVA ... 32
9.2.1 Ausgangspunkt Toolkit: Das AWT ... 33
9.2.2 Interaktionsformen in Java ... 33
9.2.3 Präsentationsformen in Java ... 34
9.2.4 Der Kontext der Interaktions- und Präsentationsformen ... 37
9.3 ÜBERSICHT ÜBER DIE INTERAKTIONSFORMEN UND PRÄSENTATIONSFORMEN ... 38

10 ZUSAMMENFASSUNG UND AUSBLICK ... 40

11 ANHANG 41
11.1 LEGENDE DER ABBILDUNGEN 41
11.1.1 Klassen- und Objektdiagramme 41
11.1.2 Interaktionsdiagramme 41
12 LITERATUR 42

Abbildungsverzeichnis

Abbildung 1: Systemvision für Datei-Such-Werkzeug 2
Abbildung 2: Variante 1 der Benutzungsoberfläche 3
Abbildung 3: Variante 2 der Benutzungsoberfläche 4
Abbildung 4: IAF-Klassenhierarchie der C++-Implementation 27
Abbildung 5: Teil eines PF-Klassenbaumes mit abgeleiteten Klassen zur Toolkit-Anbindung 28
Abbildung 6: Benutzung eines PFKlassenadapters 29
Abbildung 7: PF-Objektadapter und Klassenadapter mit drei Basisklassen 30
Abbildung 8 Die Java-Klassenhierarchie 33
Abbildung 9: Eine Hierarchie von Interaktionsformen (rechts), die die entsprechenden Präsentationsform-Interfaces benutzen (mitte), die von den Präsentationsform-Klassen implementiert werden (links). Die grauen Linien trennen die Schichten des Frameworks. 35
Abbildung 10: Die Präsentationsform-Klassen erben von den AWT-Klassen (links). 36
Abbildung 11: Interaktionsdiagramm für die Konstruktion der Interaktionsformen und deren Anbindung an die Präsentationsformen und die IAK. 37
Abbildung 12: Aktueller Stand der Hierarchie der Interaktionsformen. *) zu der 1fromNSelection werden noch Ergänzungen um Mehrfach-Selektion und statische und dynamische Selectionen hinzukommen. **) Der Name "ifFillInBoolean" ist nur eine vorübergehende Benennung. 38
Abbildung 13: Aktueller Stand der Hierarchie der Präsentationsformen. *) zu der 1fromNSelection werden noch Ergänzungen um Mehrfach-Selektion und statische und dynamische Selectionen hinzukommen. **) Der Name "pfFillInBoolean" ist nur eine vorübergehende Benennung 39
Abbildung 14: Übersicht über die Präsentationsform-Klassen und welche PF-Interfaces sie implementieren. (zu * und ** siehe oben) 39
Abbildung 15 Klassen- und Objektdiagramme 41
Abbildung 16 Interaktionsdiagramme 41

1 Einleitung und Ziele der Arbeit

Im September 1996 haben wir am Arbeitsbereich Softwaretechnik eine kleine Gruppe aus wissenschaftlichen Mitarbeitern und Studenten gebildet, die sich mit der Anbindung von objektorientierten Programmen an Fenstersysteme beschäftigte. Seitdem hat sich diese Gruppe 14 mal getroffen und verschiedene Ansätze und Konzepte entwickelt und diskutiert. Am vielversprechendsten erscheint uns zur Zeit die Anbindung über sogenannte Interaktionsformen. Dieses Konzept wurde von uns während der Treffen entwickelt und in Studien- und Diplomarbeiten, dem Java-Rahmenwerk (siehe [JWAM]) des Arbeitsbereiches Softwaretechnik sowie in industriellen Projekten umgesetzt. Dabei kamen sowohl C++ wie auch Java zum Einsatz.

Jetzt - zwei Jahre nach dem ersten Treffen - scheint uns die Zeit reif zu sein, um die erarbeiteten Konzepte in konsolidierter Form zu präsentieren. Die hier vorgestellten Ansätze sind in der Praxis erprobt und können als „funktionstüchtig“ angesehen werden.

Das Konzept der Interaktionsformen hat eine lange Geschichte. Eine der ersten Erwähnungen findet sich in der Dissertation von Reinhard Budde und Heinz Züllighoven ([BZ90]). Sie haben damals eine erste Begriffsdefinition gegeben und einige Interaktionsformen aufgeführt. Jahre später hat Sven Grand den Ansatz in seiner Diplomarbeit ([Gra95]) wieder aufgenommen. Er hat erstmals Interaktionstypen mit Hilfe von Oberflächenelementen eines GUI-Toolkits realisiert und dabei von der technisch motivierten Schnittstelle dieser Elemente abstrahiert. In den Arbeiten von Frank Fröse ([Frö96], [StrFrö96]) wurden Möglichkeiten vorgestellt, die Abhängigkeit von einem konkreten GUI-Toolkit durch eine entsprechende Kapselung zu reduzieren. Seinen vorläufigen Höhepunkt fand die konzeptionelle Ausarbeitung schließlich in der Gruppe am Arbeitsbereich Softwaretechnik. In diesem Kreis sind die Kernkonzepte für die Interaktionsformen beschrieben, wie sie Martin Lippert in seiner Studienarbeit ([Lip97]) für Java und Thorsten Görtz in seiner Diplomarbeit ([Gör98]) für C++ beschreibt.

Die Implementation der Interaktionsformen von Martin Lippert war einer der Ausgangspunkte für das JWAM-Rahmenwerk des Arbeitsbereiches Softwaretechnik. Das JWAM-Rahmenwerk unterstützt die Konstruktion interaktiver Anwendungen nach der Werkzeug- und Material-Metapher (siehe [Zül98]) mit Java. Es ist seit 1½ Jahren im Bereich Lehre und Forschung erfolgreich im Einsatz und wird zur Zeit bei verschiedenen Kooperationspartnern aus der Industrie erprobt. Das JWAM-Rahmenwerk steht im Internet zur Verfügung:

`http://swt-www.informatik.uni-hamburg.de/Software/JWAM`

Der nachstehende Text gliedert sich in folgende Abschnitte: Zuerst wird anhand eines kurzen Beispiels motiviert, warum die konventionelle Anbindung an Benutzungsoberflächen die Anwendungssoftware unnötig stark an die konkrete Benutzungsoberfläche bindet. Im Abschnitt "Begriffsbildung" wollen wir unser Verständnis der Begriffe näher erläutern und auch eine Grundlage für die folgenden Ausführungen legen. Danach zeigen wir auf, welche Probleme wir bei der konventionellen Anbindung von Benutzungsoberflächen sehen. Wir schlagen im nachfolgenden Kapitel eine Alternative dazu vor und geben unsere konzeptionelle Trennung vor. Dies wird dann in einer konkreten Umsetzung verdeutlicht. Da modale Fenster eine Sonderrolle einnehmen, haben wir ihnen ein gesondertes Kapitel gewidmet. Anschließend geben wir ein Konzept für Präsentationsformen an. Im Abschnitt "Implementationsansätze" werden Besonderheiten von C++ und Java diskutiert. Eine Zusammenfassung und der Ausblick liefern einen Eindruck von unserem Kenntnisstand bzw. den zukünftigen Arbeiten.

2 Motivation

In diesem kurzen Kapitel wollen wir das Konzept der Interaktionsformen an einem konkreten Beispiel motivieren. Anhand dieses Beispiels wollen wir zeigen, wie unterschiedlich die Oberflächengestaltung für ein fachliches Werkzeug ausfallen kann, ohne dabei seine Funktionalität zu verändern. Mit Funktionalität ist in diesem Sinne der mögliche Umgang des Anwenders mit dem Werkzeug gemeint, der den Anwender bei der Erledigung einer Aufgabe unterstützt.

Das Beispiel ist ein Werkzeug zum Suchen von Dateien auf Festplatten. Ähnliche Werkzeuge existieren auf fast allen Plattformen (z.B. Finder beim Apple-Macintosh, Explorer bei MS-Windows 95).

Zunächst entwickeln wir eine Vorstellung von dem möglichen Umgang mit dem Werkzeug. Diese Vorstellung beschreiben wir in der Systemvision in Abbildung 1.

Der Benutzer arbeitet an einem Textdokument und möchte Teile einer vorhandenen Präsentation in den Text übernehmen. Er kann sich nicht mehr genau erinnern, wo er die Präsentation unter welchem Namen abgespeichert hat.

Daher öffnet er das Datei-Such-Werkzeug. Das Werkzeug bietet dem Anwender die Möglichkeit, Dateien nach unterschiedlichen Kriterien (Dateiname, Ort, Typ, Änderungsdatum) zu suchen.

Der Anwender kann sich erinnern, die Präsentation in der letzten Woche geändert zu haben. Er weiß außerdem, daß er die Datei auf Laufwerk D: gespeichert und daß die Datei vom Typ „Präsentation“ ist.

Also wählt der Anwender in dem Datei-Such-Werkzeug das Laufwerk D: aus und stellt als Dateityp „Präsentation“ ein. Außerdem grenzt er das Änderungsdatum auf die letzte Woche ein.

Nachdem er alle Einstellungen vorgenommen hat, aktiviert er die Suche. Die Dateien, die den Suchkriterien entsprechen, werden dem Anwender präsentiert. Er wählt anhand des Dateinamens die gesuchte Datei aus und öffnet sie.

Abbildung 1: Systemvision für Datei-Such-Werkzeug

Diese Systemvision kann durch unterschiedliche Benutzungsoberflächen realisiert werden. Die in Abbildung 2 dargestellte Oberfläche des Softwarewerkzeuges ist *eine* mögliche Umsetzung der Systemvision.

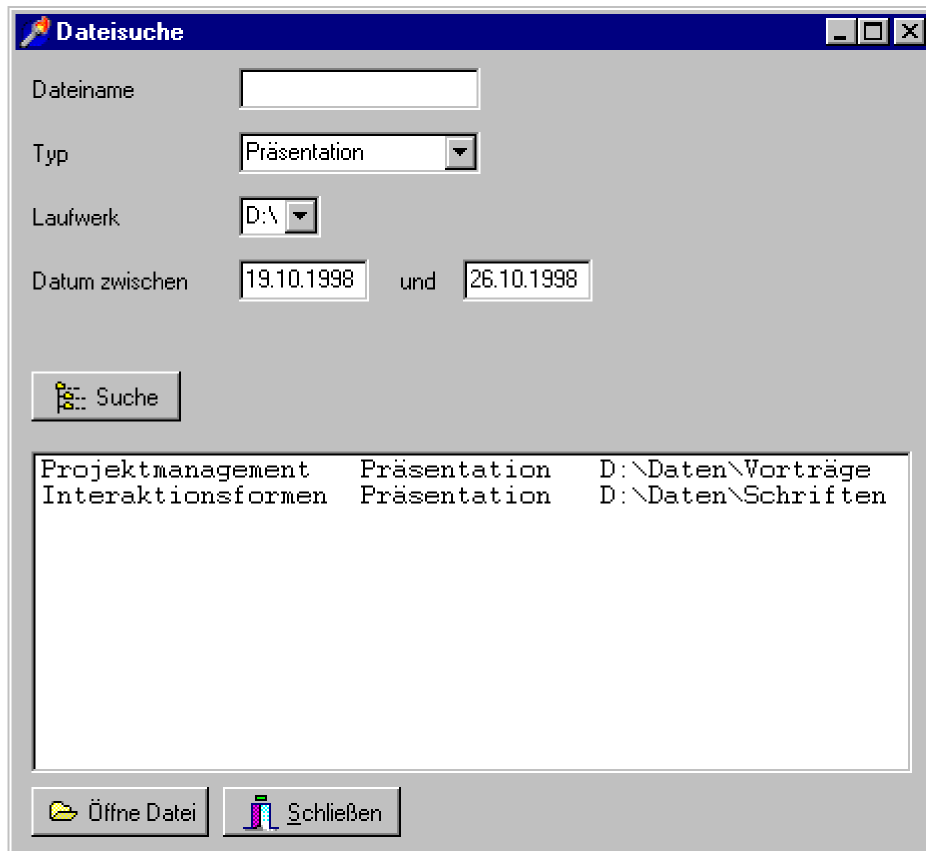

Abbildung 2: Variante 1 der Benutzungsoberfläche

Es sind verschiedene Varianten dieser Benutzungsoberfläche denkbar. Insbesondere könnten die folgenden Variationen sinnvoll sein:

- Das Laufwerk wird nicht über eine Combo-Box ausgewählt, sondern über eine Radio-Group.
- Die Datumsangaben werden nicht in ein Eingabefeld eingetippt, sondern aus einer Kalenderansicht ausgewählt.
- Die gewünschte Datei wird nicht über den Knopf „Öffne" ausgewählt, sondern durch einen Doppelklick auf die Liste mit dem Suchergebnis.

An der Form des prinzipiellen Umgangs des Anwenders mit dem Werkzeug ändert sich nichts auf der Ebene der Interaktion: Der Anwender selektiert, füllt aus und aktiviert. Die Interaktion kann also unverändert bleiben, auch wenn sich die Präsentation – in gewissen Grenzen – ändert. In Abbildung 3 ist die Benutzungsoberfläche mit den beschriebenen Änderungen dargestellt.

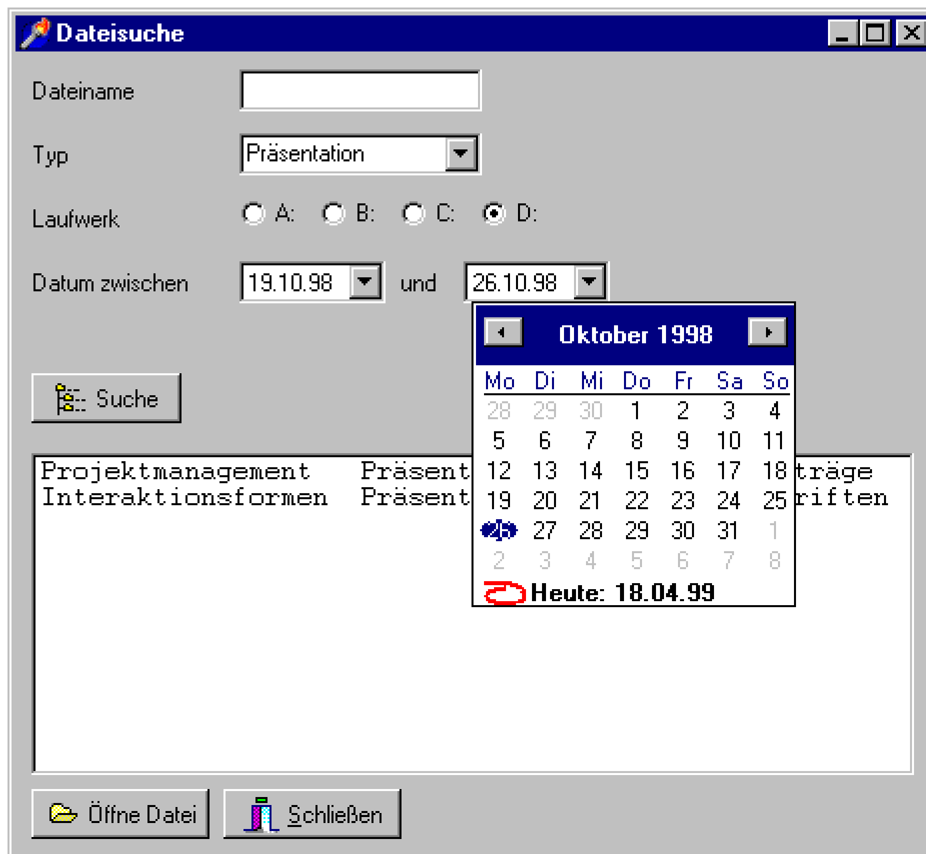

Abbildung 3: Variante 2 der Benutzungsoberfläche

Diese Variabilität der Benutzungsoberfläche unter Beibehaltung des Umgangs mit dem Softwarewerkzeug ist die Grundlage für das Konzept der Interaktionsformen: Für das Softwarewerkzeug soll die Benutzungsoberfläche nicht konkret „sichtbar“ sein, sondern nur die möglichen Interaktionen mit dieser. Das Programm fokussiert also auf die Interaktion und nicht auf die Präsentation. Die Präsentation kann in den Grenzen variiert werden, die durch die Interaktion festgelegt werden.

3 Begriffsbildung

In diesem Abschnitt wollen wir grundlegenden Begriffe für die weitere Arbeit einführen. Wir gehen davon aus, daß Begriffe wir Objektorientierung, Geheimnisprinzip, Kapselung etc. nicht weiter eingeführt werden müssen. Näheres dazu findet sich z.B. in [Mey88].

Seit der Entwicklung der Programmiersprache Smalltalk mit den dazugehörigen Klassenbibliotheken und dem zugrundeliegenden Fenstersystem, haben grafische Benutzungsoberflächen Einzug in alle Bereiche der Softwaretechnik gehalten. Alle verbreiteten Betriebssysteme bieten heute grafische Benutzungsoberflächen für die Interaktion mit dem Benutzer an. Wir benutzen im folgenden den Begriff *Fenstersystem* für diese Gattung von Benutzungsoberflächen. Damit machen wir zugleich deutlich, daß es für uns unerheblich ist, ob dieses grafisch realisiert wird, oder durch eine zeichenorientierte Darstellung.

Benutzer müssen mit Softwaresystemen interagieren, um diese zu steuern und deren Zustand in Erfahrung zu bringen. Dazu muß das Softwaresystem Daten präsentieren und Benutzeraktionen entgegennehmen und in ausführbare Kommandos umsetzen. Heute ist zu diesem Zweck die Verwendung grafischer Benutzungsoberflächen (GUI - Graphical User Interface) Stand der Kunst. Alle relevanten Betriebssysteme bieten grafische Benutzungsoberflächen (Apple Macintosh, OS/2, Windows 3.1, 95, NT, Unix XWindows).

Ein **Fenstersystem** basiert meist auf der Schreibtischmetapher, die dem Benutzer bestimmte Elemente (z.B. Knöpfe und Eingabefelder) in Fenstern präsentiert. Der Benutzer kann diese Fenster über einander anordnen, verschieben, vergrößern, verkleinern etc. und die Interaktion mit dem Programm ausführen, indem er Knöpfe drückt, Eingaben in Textfeldern vornimmt usw. Beispiele für Fenstersysteme sind: X-Windows, Motif, Apple Desktop (MacOS), MS-Windows, Presentation Manager. (Siehe dazu auch [RW96]). Anstelle von Fenstersystem wird häufig auch die Abkürzung GUI (Graphical User Interface) verwendet.

Die in den Fenstern eines Fenstersystems dargestellten Elemente werden Widgets[1] oder Controls genannt. Sie dienen der Präsentation von Daten an der Benutzungsoberfläche sowie dem Entgegennehmen von Benutzeraktionen. Wir verwenden den Begriff **Widget** für die weitere Arbeit. Beispiele für Widgets sind: Fenster, Knopf, Eingabefeld, Listbox.

Ein **Styleguide** (in diesem Kontext) ist eine Richtlinie für die Gestaltung von Benutzungsoberflächen. Sie regelt, in welcher Art und Weise die Widgets eines Fenstersystems komponiert werden sollen und wie sich Programme für den Benutzer verhalten sollen.

Das **API** (Application Programming Interface) definiert die Schnittstelle eines Bestriebssystemdienstes für Anwendungsprogramme. In unserem Zusammenhang ist insbesondere das API des Fenstersystems von Interesse.

Ein **GUI-Builder** ist ein spezielles Programm, mit dem der Entwickler die Benutzungsoberfläche seines Programmes erstellen kann. Dies erfolgt in der Regel visuell nach dem WYSIWYG[2]-Prinzip.

Eine **Klassenbibliothek** ist eine Ansammlung von wiederverwendbaren Klassen. Diese können in unterschiedlichen objektorientierten Anwendungen verwendet werden. Beispiele für Klassenbibliotheken sind Container-Bibliotheken (z.B. STL - Standard Template Library für C++).

Ein **Rahmenwerk** ist eine Sammlung kooperierender Klassen, welche die grundlegende Architektur der Anwendung vorgeben und bereits den Kontrollfluß der Anwendung festlegen. Häufig wird auch der englischsprachige Begriff *Framework* verwendet. (Siehe auch [Wei97] und [Bäu98]).

Da die heute verfügbaren Betriebssysteme in der Regel in der Programmiersprache C implementiert sind, liegt das API zum Fenstersystem ebenfalls in C vor. Soll mit einer objektorientierten Programmiersprache (z.B. C++) auf diese Schnittstelle zugegriffen werden, so tritt beim Übergang von der Anwendung zur Benutzungsoberfläche ein Paradigmenbruch auf. Während die Anwendung objektorientiert implementiert wurde, liegt der Implementation der Benutzungsoberfläche ein imperatives Paradigma zugrunde.

Die Anbindung der Anwendung an die Benutzungsoberfläche über das entsprechende API führt weiterhin zu einer Abhängigkeit der Anwendung vom Fenstersystem.

Aus diesen beiden Gründen existiert eine große Anzahl von Klassenbibliotheken zum Zugriff auf das Fenstersystem-API. Solche Klassenbibliotheken werden häufig GUI-Toolkits genannt. Sie enthalten neben den Klassen häufig zusätzliche Werkzeuge, wie z.B. GUI-Builder. GUI-Toolkits kapseln den imperativen Zugang objektorientiert in Klassen. Auf diese Weise tritt der beschriebene Paradigmenbruch nicht auf. Weiterhin bieten die meisten GUI-Klassenbibliotheken Unterstützung für mehrere Fenstersystem-APIs. Greift eine Anwendung nur auf die Klassen der Klassenbibliothek zu, so ist sie mit den von der Klassenbibliothek

[1] Der Begriff stammt vom englischen Wort „Gadget" (dt. Dingsbums) und bildet sich aus dem Zusammenziehen der Worte Window und Gadget.

[2] WYSIWYG: What You See Is What You Get

unterstützten Fenstersystemen lauffähig. Beispiele für solche GUI-Klassenbibliotheken sind Zinc Application Framework, zApp, wxWindows, StarView etc.

Ein **GUI-Toolkit** ist eine Ansammlung von wiederverwendbaren Code-Fragmenten und spezialisierten Werkzeugen zur Anbindung von Anwendungen an →Fenstersysteme. Die Code-Fragmente müssen nicht zwangsweise objektorientiert entworfen sein oder in einer objektorientierten Programmiersprache vorliegen. In der Regel liegt einem GUI-Toolkit mindestens ein →GUI-Builder bei.

Der Begriff des Fenstersystems ist eng mit dem *reaktiver Systeme* verbunden. Ein reaktives System zeichnet sich durch die sogenannte „Invertierung des Kontrollflusses“ aus. Während in frühen Softwaresystemen der Kontrollfluß alleine dem Programm oblag, so liegt dieser jetzt beim Benutzer. Der Benutzer wählt selbständig das Fenster aus, in dem er die nächste Eingabe machen will und das Programm muß darauf in geeigneter Weise reagieren.

Wir sprechen von einem **reaktiven System**, wenn der Programmablauf nicht vom Programm sondern vom Benutzer gesteuert wird. Das Programm hat „lediglich“ die Aufgabe, auf die Benutzeraktionen in geeigneter Weise zu *reagieren*. (Siehe dazu auch [RW96]).

4 Probleme konventioneller Anbindung an Benutzungsoberflächen

Dieses Kapitel soll erklären, wie stark der Einfluß konventioneller Anbindung von Benutzungsoberflächen auf die Architektur von Anwendungssystemen ist. In unserem Ansatz vermeiden wir die hier aufgedeckten Schwächen.

Die Anbindung von Benutzungsoberflächen mit Hilfe von GUI-Toolkits hört sich in der Theorie ganz einfach an, bereitet in der Praxis jedoch häufig Probleme. Ausgefeilte GUI-Toolkits wie MacApp bieten nicht nur Klassen für die Entwicklung von Benutzungsoberflächen, sondern geben praktisch die komplette Struktur einer Anwendung vor. Soll statt dessen eine eigene Architektur umgesetzt werden, arbeitet man praktisch gegen das Rahmenwerk. Mit Maßgaben, z.B. als Rahmenwerke für dokumentenzentrierte Anwendungen zu dienen, werden die Einsatzmöglichkeiten zusätzlich begrenzt. Oftmals verschmelzen GUI-Toolkits und Rahmenwerke zu einer Einheit.

Die monolithische Struktur dieser Rahmenwerke läßt sich häufig nicht verbergen. Oft entsteht laut Rosenstein ([Ros95]) das Problem, daß durch die Etablierung des Rahmenwerks die Schnittstellen konstant gehalten werden müssen und Strukturänderungen, die in der Frühphase noch eher möglich waren, unterbleiben müssen. Ein optimales Design auf Basis aktueller Erkenntnisse könne so praktisch nicht mehr einfließen. In neueren GUI-Rahmenwerkes werde dies aber zunehmend berücksichtigt, indem man die monolithische Struktur aufbricht und Subrahmenwerke entwirft.

Auch wenn nur die Möglichkeiten für die Implementierung einer Oberfläche genutzt werden sollen, müssen an vielen Stellen die fachlichen Anteile mit den technischen Klassen der Rahmenwerke vermischt werden. So entsteht eine Abhängigkeit, die in vielen Fällen gar nicht erwünscht wird. Probleme entstehen, wenn aus irgendwelchen Gründen die Möglichkeiten des verwendeten GUI-Rahmenwerkes nicht mehr ausreichen und für den Einsatz eines neuen Toolkits die Struktur der Anwendung in weiten Teilen geändert werden muß. In fast allen Fällen ist es rein technisch unmöglich, zwei (oder gar mehrere) verschiedene GUI-Toolkits in einem Programm zu verwenden.

Außerdem fällt die Mächtigkeit der angebotenen Funktionen auf, die über die Unterstützung der Entwicklung von Benutzungsschnittstellen z.T. weit hinausgehen, dabei aber mehr oder weniger stark in das Rahmenwerk eingebettet sind. So besteht wieder die Gefahr, daß die anwendungsspezifischen Anteile eines Programmes zu sehr mit den technischen Möglichkeiten eines Rahmenwerkes vermischt werden. Das Toolkit sollte mit den fachlichen Komponenten nur lose gekoppelt werden.

Ein spezielles Phänomen tritt bei GUI-Toolkits auf, welche verschiedene Fenstersysteme unterstützen. Obwohl sich die von den Fenstersystemen bereitgestellten Widgets sehr ähnlich sind, unterscheiden sich die Fenstersystem-APIs zum Teil recht erheblich. Weiterhin existieren häufig größere Unterschiede in den Styleguides. Um ein vom Fenstersystem unabhängiges GUI-Toolkit zu entwickeln, müssen alle diese Unterschiede vor dem Anwendungsentwickler verborgen werden. Im wesentlichen realisieren die heute verfügbaren GUI-Toolkits zwei Varianten:

- **Kleinster gemeinsamer Nenner:** Es wird nur das unterstützt, was allen Fenstersystemen gemeinsam ist. Dies führt zu einer relativ einfachen Implementation des GUI-Toolkits. Allerdings können dann spezielle Widgets einzelner Fenstersysteme nicht mehr verwendet werden. Dies resultiert wiederum in Verstößen gegen die Styleguides wenn gerade die Benutzung dieser nicht unterstützten Widgets gefordert ist. So existiert z.B. das Widget

TreeView zur Baumdarstellung unter MS-Windows aber nicht unter X-Windows. Den Weg des kleinsten gemeinsamen Nenners geht z.B. das AWT[3] des Java-JDKs[4].

- **Angleichung:** Es wird die Obermenge der Widgets aller Fenstersysteme gebildet. Wo einzelne Widgets nicht existieren, werden diese durch das GUI-Toolkit nachgebildet. Dies führt zu einer erheblich aufwendigeren Konstruktion des GUI-Toolkits. Weiterhin sind auch bei diesem Vorgehen Verstöße gegen die Styleguides vorprogrammiert. Schließlich resultiert die Verwendung des TreeView unter MS-Windows in der Verwendung eines nachgebildeten TreeViews unter X-Windows. Neben diesen Verstößen gegen die Styleguides sind nachimplementierte Widgets in der Regel fehlerhafter und langsamer als die vom Fenstersystem bereitgestellten. Die meisten heute verfügbaren GUI-Toolkits implementieren Widgets nach (z.B. Zinc Application Framework, Java-Swing). Allerdings ist hier selten die volle Palette aller Widgets aller Benutzungsoberflächen anzutreffen. In der Regel werden nur die wichtigsten Widgets nachimplementiert.

Offensichtlich bietet keine der beiden Varianten eine befriedigende Lösung für die fenstersystemunabhängige Anwendungsentwicklung.

Zusammenfassend bleibt festzuhalten: moderne GUI-Rahmenwerke sind mittlerweile so ausgefeilt, daß sie die Komplexität prozeduraler Toolkits oder unüberschaubarer Fenstersystem-APIs erheblich verringern. Eine Unterstützung durch GUI-Builder verringert den Entwicklungsaufwand weiter. Um umfangreiche Benutzungsschnittstellen zu realisieren, besteht zu ihnen praktisch keine Alternative. Dabei kann die Abhängigkeit vom technischen Basissystem verringert werden, indem z.B. vom konkreten Fenstersystem abstrahiert wird. So können mit einem GUI-Toolkit Oberflächen erstellt werden, die sich ohne eine Änderung am Quelltext auf andere Plattformen portieren lassen. Gleichzeitig entsteht aber nur durch die Benutzung solch eines Rahmenwerkes eine neue Abhängigkeit, die ebenso wenig erwünscht sein kann. Außerdem lassen es viele Rahmenwerke nicht zu, daß ein Entwickler seine Vorstellungen einer Anwendungsarchitektur umsetzen kann.

Diese Betrachtung mündet in vier konkrete Forderungen, die sich für den Einsatz von GUI-Rahmenwerken aufstellen lassen:

- **Portabilität**: Das Rahmenwerk muß vom konkreten Fenstersystem abstrahieren, um die Portierung einer Anwendung auf andere Fenstersysteme zu ermöglichen. Dabei sollte die Möglichkeit bestehen, die verschiedenen Styleguides zu berücksichtigen.
- **Austauschbarkeit**: Die Klassen eines GUI-Rahmenwerkes müssen so entworfen werden, daß die Abhängigkeiten zu den fachlichen Klassen einer Anwendung minimal gehalten werden können. So kann das Rahmenwerk leichter gegen ein anderes ausgetauscht werden.
- **Architekturneutralität**: Der Entwickler soll die Architektur seiner Anwendung selber bestimmen können. Das GUI-Rahmenwerk muß sich diesem Anspruch unterordnen und sollte deshalb keine eigenen Vorgaben für die Architektur treffen.
- **Mächtigkeit**: Die vom verwendeten Fenstersystem bereitgestellten Möglichkeiten sollten ausgenutzt werden können.

Die vier Forderungen lassen erkennen, daß es das „perfekte" GUI-Rahmenwerk wohl nicht geben kann. Wie Kilberth et al. in [KGZ94] schreiben, wird heute niemand mehr ohne größere Not ein eigenes Fenstersystem für die Implementation interaktiver Anwendungen programmieren. Das läßt sich sicherlich auch auf die GUI-Rahmenwerke übertragen. Es gilt

[3] AWT - Abstract Windowing Toolkit
[4] JDK - Java Development Kit

also, einen „defensiven" Ansatz zu finden, bei dem die Nachteile der Toolkits in einem gewissen Maße in Kauf genommen werden. Die fachlichen Anteile einer Anwendung sollten dabei vom GUI-Toolkit möglichst vollständig entkoppelt werden. So könnte das Toolkit eingesetzt werden, das den gegebenen (oder sich ändernden) Anforderungen am ehesten entspricht und vor allem mit geringem Aufwand auch gegen ein anderes ausgetauscht werden kann.

Wir werden im folgenden einen Ansatz anbieten, der es ermöglicht, ein fremdes GUI-Toolkit anzuschließen, ohne daß dadurch für den selbstgeschriebenen Programmtext unnötige Abhängigkeiten entstehen. Unser Konzept fokussiert dabei eher auf die Umgangsformen.

5 Interaktionsformen zur flexiblen Anbindung an Benutzungsoberflächen

In diesem Kapitel beschreiben wir die Komponenten, mit deren Hilfe wir flexibel anbindbare Benutzungsoberflächen konstruieren wollen: Interaktions- und Präsentationsformen.

5.1 Interaktionsformen

Wie wir bereits anhand des Beispiels gezeigt haben, ist es möglich, verschiedene Präsentationen für die Realisierung einer Interaktion zu verwenden. Darauf basiert das Konzept der Interaktionsformen. Wir konzentrieren uns deshalb beim Programmentwurf auf die Interaktionen und wählen erst später passende Präsentationen aus. So können Präsentationen im Verlauf noch geändert werden, ohne daß der Entwurf geändert werden muß. Dieser basiert schließlich nur auf den Interaktionen.

> *Eine Interaktionsform definiert die abstrakte Handhabung eines Werkzeuges. Ein Werkzeug gebraucht Interaktionsformen, um die mögliche Interaktion mit dem Werkzeug zu beschreiben.*
>
> Eine *Interaktionsform* repräsentiert eine fachliche Umgangsform mit einem Werkzeug und hat keine Seiteneffekte, die sich auf das Werkzeug auswirken.
>
> *Interaktionsformen* werden benutzt, um das Werkzeug mit einer konkreten Umsetzung der Handhabung durch ein GUI-Toolkit zu verbinden.
>
> *Interaktionsformen* machen ein Werkzeug unabhängig von der technischen Präsentation durch das Toolkit, die beliebig ausgetauscht werden kann.

Bei der Konstruktion eines Werkzeuges soll sich der Entwickler nicht mehr an der Vorstellung einer konkreten Oberfläche orientieren, wie er es bisher bei der Auswahl der Widgets getan hat. Statt dessen wählt er aus der Menge der Interaktionsformen diejenigen aus, die gemäß der Funktionalität eines Werkzeuges benötigt werden. Ein Werkzeug wird also orientiert am intendierten Umgang beschrieben und nicht an der konkreten Realisierung eines GUI. Das GUI wird erst später mit dem Werkzeug verbunden und interpretiert durch seine Widgets die Handhabung.

Ein Widget soll aus der Sicht eines Werkzeuges nur noch unter dem Aspekt der Handhabung betrachtet werden. Die Umsetzung der Präsentation muß komplett außerhalb der Interaktionskomponente verwaltet werden. Das Ziel ist es, die Anbindung eines GUI-Toolkits nicht nur zu kapseln, sondern tatsächlich vollständig zu verbergen, so daß es nicht einmal durch eine implizite Benutzung erkannt werden kann. Die Interaktionskomponente darf die Oberfläche eines Werkzeuges nicht mehr implementieren, indem sie Widgets benutzt, vielmehr soll eine Anbindung der Widgets erfolgen, indem sie über den Typ ihrer Handhabung mit dem Werkzeug verbunden werden. Dem Werkzeug muß also ein Ausdrucksmittel zur Verfügung gestellt werden, um die Handhabung zu definieren.

Der wesentliche Unterschied der Interaktionsformen zu den bisher bekannten Widgets besteht darin, daß ein Interaktionsform-Objekt nicht mehr für die Kapselung eines einzelnen Widgets zuständig ist. Eine Interaktionsform repräsentiert vielmehr ein „virtuelles GUI-Element", das tatsächlich genau eine Art der Handhabung umsetzt, dabei aber keinerlei Annahmen über die Präsentation an der Benutzungsschnittstelle macht.

Tatsächlich ist es möglich, nur mit den Interaktionsformen „Aktivieren", „Wert eingeben oder darstellen", „Wert selektieren" sowie „Werte auflisten und markieren" die Handhabung jeder denkbaren Benutzungsschnittstelle zu beschreiben, die mit den üblichen Widgets implementiert werden kann.

5.2 Präsentationsformen

Eine Präsentationsform realisiert in der Benutzungsschnittstelle eines Werkzeuges die konkrete Präsentation und Handhabung einer fachlichen Umgangsform, die durch eine Interaktionsform vorgegeben wird.

Präsentationsformen kapseln die verschiedenen Widgets eines GUI-Toolkits und erfüllen dabei ein Protokoll, welches die Kopplung mit den Interaktionsformen erlaubt.

Für jedes Widget existiert eine spezielle Präsentationsform, die vorgibt, mit welcher Interaktionsform dieses Widget verbunden werden kann.

Präsentationsformen werden außerhalb eines Werkzeuges verwaltet und mit den Interaktionsformen innerhalb des Werkzeuges verbunden.

In den konkreten Präsentationsformen wird letztlich die Anbindung an das verwendete GUI-Toolkit durchgeführt. Die Präsentationsform-Objekte werden mit passenden Interaktionsform-Objekten innerhalb eines Werkzeuges gekoppelt, so daß die Handhabung eines Werkzeuges mit seiner Präsentation verbunden wird. Widgets, die keinen konkreten fachlichen Umgang mit dem Werkzeug realisieren, können mit keiner Interaktionsform verbunden werden. Sie sollen aber trotzdem als Präsentationsformen bezeichnet werden. Beispiele für solche Präsentationsformen sind z.B. statische Texte.

Fachlich bieten die Interaktionsformen dem Entwickler ein neues Ausdrucksmittel. Er orientiert sich nun nicht mehr an den Widgets, die durch die Erzeugung der Interaktionstypen in der Benutzungsschnittstelle erscheinen, sondern benutzt Interaktionsformen gemäß der Umgangsformen, die das Werkzeug bietet.

Durch die Aufteilung in Interaktion und Präsentation kommt in der Anbindung des GUI eine weitere – konzeptionelle - Indirektion zwischen Werkzeug und Toolkit hinzu, die im Vergleich mit der Architektur mit Widgets erhebliche Änderungen nach sich ziehen kann. Siehe dazu Kapitel 9.

6 Beschreibung konkreter Interaktionsformen

In diesem Abschnitt stellen wir die konsolidierten Interaktionsformen vor. Im Anschluß präsentieren wir die Interaktionsformen, die wir zwar diskutiert, die aber noch nicht in einer konsolidierten Form vorliegen.

6.1 Konsolidierte Interaktionsformen

Alle Interaktionsformen erben von einer abstrakten Oberklasse, in der das Basisprotokoll der Interaktionsformen vorgegeben ist. Diese Oberklasse trägt den Namen `IAFBase`.

6.1.1 IAFBase

Anwendbarkeit

`IAFBase` ist eine abstrakte Oberklasse zu den Interaktionsformen. Sie kann nicht direkt verwendet werden. In der Oberklasse `IAFBase` wird mit den beiden Methoden `Enable` und `Disable` ein Protokoll definiert, um IAF-Objekte (de-)aktivieren zu können. So läßt sich der Zustand des Werkzeuges berücksichtigen, indem einzelne Handhabungsmöglichkeiten gezielt gesperrt werden können.

Schnittstelle

```
public class IAFBase extends object
IAFBase (IAFsContext context, java.lang.String name);
void Enable ();
void Disable ();
bool IsEnabled ();
```

Präsentationsformen

6.1.2 IAFActivator

Anwendbarkeit

Die Klasse `IAFActivator` steht für die Handhabungsform „Aktivieren". Technisch wird die Aktivierung mit Kommando-Objekten durchgeführt, die höhere Ereignisse vom Widget zur IAK melden.

Schnittstelle

```
public class IAFActivator extends IAFBase

ifActivator(IAFsContext context, java.lang.String name)
          // create an interaction form for an activation
void attachActivateCommand(cmdActivate cmd)
          // attach the command for an activation at the user interface
void pfActivated(cmdObject cmd)
          // callback method for the command which is attached to the
          // connected pfs by this if
```

Präsentationsformen

Ein `IAFActivator` kann mit Hilfe eines Knopfes, eines Menüeintrags oder eines Elementes in einer Liste präsentiert werden.

6.1.3 IAF1FromNSelection

Anwendbarkeit

`IAF1FromNSelection` dient zur Realisierung der Handhabung „Auswahl 1 aus n" wie sie bei der Auswahl aus einer Liste vorgenommen wird. Die Klasse `IAF1FromNSelection` ist eine abstrakte Oberklasse.

Schnittstelle

```
public class IAF1FromNSelection extends IAFBase

IAF1fromNSelection(IAFsContext context, java.lang.String name)
          // create an interaction form to select one from many

void attachSelectCommand(cmdSelect cmd)
          // attach the command for a selection
void pfSelected(cmdObject cmd)
          // callback method for the command which is attached to the
          // connected pfs by this if

java.lang.String selection()
          // returns the name of the selected element
Identificator selectionID()
          // returns the id of the selected element
int selectionIndex()
          // returns the index of the selected element

int elementCount()
          // returns the count of the elements in the selection
Identificator idOfIndex(int index)
           // returns the id of the element with the given index.
int indexOfID(Identificator id)
           // returns the index of the element with the given id.

void select(Identificator id)
          // select the element by the id
void select(int index)
          // select the element by the index
```

6.1.4 IAFStatic1FromNSelection

Anwendbarkeit

`IAFStatic1FromNSelection` dient zur Realisierung der Handhabung „Auswahl 1 aus n" wie sie bei der Auswahl aus einer Liste vorgenommen wird. Die Klasse `IAFStatic1FromNSelection` wird verwendet, wenn alle Elemente der Auswahl vor Darstellung der Elemente bekannt sind.

Schnittstelle

```
public class IAFStatic1FromNSelection extends IAF1FromNSelection
```

```
void set(conContainer[5] items)
         // set the elements of the selection. (old elements will be
         // removed before)
void set(conExtOrdCol items, conExtOrdCol ids)
         // set the elements of the selection. (old elements will be
         // removed before)
```

Präsentationsformen

Die Interaktionsform `IAFStatic1FromNSelection` kann durch die Präsentationsformen `PFList` oder `PFChoice` (eine Gruppe von Radiobuttons) präsentiert werden.

6.1.5 IAFDynamic1FromNSelection

Anwendbarkeit

`IAFDynamic1FromNSelection` dient zur Realisierung der Handhabung „Auswahl 1 aus n" wie sie bei der Auswahl aus einer Liste vorgenommen wird. Die Menge, aus der ausgewählt wird, ist dynamisch und kann vor Beginn der Auswahl nicht angegeben werden oder soll nach einer ersten Darstellung verändert werden. Dies ist insbesondere dann der Fall, wenn die Menge zu groß ist, um komplett an die Interaktionsform weitergegeben zu werden, oder wenn die Menge durch eine andere Interaktionsform beeinflußt wird.

Schnittstelle

```
public class IAFDynamic1FromNSelection extends IAF1FromNSelection

void add(java.lang.String item)
         // add an element to the selection
void add(java.lang.String item, Identificator id)
         // add an element to the selection
void remove(int index)
         // remove an element from the selection
void removeAll()
         // remove all elements from the selection
```

Präsentationsformen

Die Interaktionsform `IAFDynamic1FromNSelection` wird durch die Präsentationsform `PFList` präsentiert.

6.1.6 IAFMFromNSelection

Anwendbarkeit

Wenn die Möglichkeit gegeben sein soll, aus einer Menge von Auswahlmöglichkeiten mehrere Elemente zu wählen (multiselect), wird ein `IAFMFromNSelection` eingesetzt.

Schnittstelle

```
public class IAFMFromNSelection extends IAFBase
```

[5] Wir verwenden in der jetzigen Implementation als Container eine im JWAM-Framework integrierte Container-Bibliothek (die JConLib). Selbstverständlich lassen sich auch andere Behälterklassen verwenden.

```
void attachSelectCommand(cmdSelect cmd)
         // attach the command for a selection

conContainer selections()
         // returns the collection of names of the selected elements
conContainer selectionIDs()
         // returns the ids of the selected elements
conContainer selectionIndexes()
         // returns the indexes of the selected elements

void select(Identificator id)
         // select the element by the id
void select(int index)
         // select the element by the index
```

6.1.7 IAFStaticMFromNSelection

Anwendbarkeit

`IAFStaticMFromNSelection` erlaubt eine Mehrfachauswahl „m aus n". Die Menge aus der ausgewählt wird steht vor Beginn der Auswahl fest und kann komplett angegeben werden.

Schnittstelle

```
public class IAFStaticMFromNSelection extends IAFMFromNSelection
```

Präsentationsformen

Die Interaktionsform `IAFStaticMFromNSelection` kann durch die Präsentationsformen `PFList` oder `PFCheckGroup` (eine Gruppe von Check-Boxen) präsentiert werden.

6.1.8 IAFDynamicMFromNSelection

Anwendbarkeit

`IAFStaticMFromNSelection` erlaubt eine Mehrfachauswahl „m aus n". Die Menge aus der ausgewählt wird ist dynamisch und kann vor Beginn der Auswahl nicht angegeben werden oder soll nach einer ersten Darstellung verändert werden. Dies ist insbesondere dann der Fall, wenn die Menge zu groß ist, um komplett an die Interaktionsform weitergegeben zu werden, oder wenn die Menge durch eine andere Interaktionsform beeinflußt wird.

Schnittstelle

```
public class IAFDynamicMFromNSelection extends IAFMFromNSelection

void add(java.lang.String item)
         // add an element to the selection
void add(java.lang.String item, Identificator id)
         // add an element to the selection
void remove(int index)
         // remove an element from the selection
void removeAll()
         // remove all elements from the selection
```

Präsentationsformen

Die Interaktionsform `IAFDynamicMFromNSelection` wird durch die Präsentationsform `PFList` präsentiert.

6.1.9 IAFFillIn

Anwendbarkeit

`IAFFillIn` definiert eine Schnittstelle für die Handhabung des „Ausfüllens", d.h. Eingabe eines Wertes. Zusätzlich werden spezielle Interaktionsformen angeboten, die den Typ des Wertes festlegen, der eingegeben werden kann. Objekte der Klasse `IAFFillIn` oder einer Unterklasse werden auch eingesetzt, wenn ein Wert nur dargestellt und nicht verändert werden darf.

Schnittstelle

```
public class IAFFillin extends IAFBase

void clear ();
```

6.1.10 IAFStringFillIn

Anwendbarkeit

`IAFStringFillIn` wird für die Eingabe oder Anzeige eines Text-Wertes verwendet.

Schnittstelle

```
public class IAFStringFillin extends IAFFillin


IAFStringFillin (IAFsContext context, java.lang.String name)
         // create an interaction form to fill in some text

void attachTextChangeCommand(cmdTextChange cmd)
          // attach the command for a text change in the fill in
void pfEdited(cmdObject cmd)
         // callback method for the command which is attached to the
         // connected pfs by this if

void set(java.lang.String value)
          // set the value for the fill in
java.lang.String value()
          // the value of the fill in
```

Präsentationsformen

`IAFStringFillIn` wird mit Hilfe der Präsentationsform `PFString` dargestellt.

6.2 Komponenten zur Komposition von Interaktionsformen

6.2.1 GUI-Manager

Anwendbarkeit

Die Programmkomponente „GUI-Manager“ wird eingesetzt, um mit Hilfe der Ressourcen für eine Interaktionsform die gewünschte Präsentationsform zu erzeugen. Über den GUI-Manager kann sich eine Interaktionskomponente den oder die zu ihr passenden `IAFsContext` Objekte geben lassen.

Der GUI-Manager interpretiert die Ressourcen und erzeugt die zu den Ressourcen passende PF. Er dient außerdem dazu, die Ereignisschleife des Fenstersystems zu aktivieren. Auf der Realisierungsebene handelt es sich bei dem GUI-Manager um ein Fabrik-Singleton.

Schnittstelle

```
public class GUIManager

boolean existsIAFsContext(java.lang.class IPClass, java.lang.String
                          IPName))

IAFsContext iafsContext(java.lang.class IPClass, java.lang.String IPName);

void Run();
```

6.2.2 IAFsContext

Anwendbarkeit

Der Anwendungsentwickler muß vor dem Anlegen einer Interaktionsform einen `IAFsContext` haben, in den die Interaktionsform eingeordnet wird.

Er benutzt `IAFsContext` von dem Werkzeug aus, um Teile einer Oberfläche ein- oder ausblenden zu können. Er kann sich mit Hilfe von `IAFsContext` die zu einer Interaktionsform gehörenden Präsentationsformen geben lassen, um so z.B. direkt auf Eigenschaften wie Farbe und Schrifttyp zugreifen zu können.

Ein `IAFsContext` wird vom Anwendungsentwickler nicht selbst erzeugt.

Schnittstelle

```
void show()
         // Shows the user interface represented by this IAFsContext
void hide()
         // Hides the user interface represented by this IAFsContext

java.lang.String name()
         // the name of the IAFsContext

conSet pfs(PFDescriptor description, java.lang.String name)
         // returns the presentation form components out of the user
         // interface of this IAFsContext matching to the pfDescriptor

boolean hasSubIAFsContext(java.lang.String name)
         // Has this IAFsContext a sub context with the specified name?
IAFsContext subIAFsContext(java.lang.String name)
         // Returns the sub context with the specified name
void registerSubIAFsContext(IAFsContext context)
         // Registers the specified context as sub context of this context
```

6.3 Weitergehende Interaktionsformen

6.3.1 IAFTable

Anwendbarkeit

Die Interaktionsform `IAFTable` dient der Darstellung einer 2-dimensionalen Tabelle. Dabei kann zur Laufzeit zumindest die Anzahl der angezeigten Zeilen, häufig auch die Anzahl der

angezeigten Spalten verändert werden. Nach dem heutigen Stand der Technik können sich in den Tabellenzellen beliebige "einfache" Interaktionsformen befinden.

Diskussion

Für ein `IAFTable` ist nicht wie bei den anderen Interaktionsformen ein Handlungsabschluß mit dem dazu gehörenden Kommando zu definieren, vielmehr muß für jede einzelne Zelle ein Kommando registriert werden können. Der Umgang mit den Werten in der Tabelle soll möglichst an die schon vorhandenen Interaktionsformen angelehnt sein.

Beim `IAFTable` wird eine Musterzeile/spalte (im weiteren nur Zeile genannt) registriert, die den Aufbau und den Umgang mit den Zellen einer Zeile festlegt. Die Zeile ist folgendermaßen aufgebaut:

Zeile [1..n Spalten, je Spalte 1..m IAFs + ID der Spalte]

Für die Spaltenüberschriften ist eine eigene Musterzeile vorgesehen, weil evtl. der Umgang mit den Überschriften anders als mit dem Rest der Tabelle ist. Die `IAFTable` verwaltet die Musterzeilen. Der Zugriff auf die Zellen einer Zeile der Tabelle erfolgt über die Interaktionsformen der einzelnen Spalten in der Musterzeile. Die Musterzeile stellt einen Cursor auf die Tabelle dar und kann entsprechend bewegt werden, um alle Zellen der Tabelle zu füllen. Da evtl. der gesamte Tabelleninhalt nicht auf einmal dargestellt werden kann, muß die `IAFTable` *aktiv(!)* bei der Interaktionskomponente um neue Werte bitten, wenn in der Tabelle gescrollt wird. Eine Tabelle unterstützt verschiedene Selektionsformen, die nicht Bestandteil der `IAFTable` sind, sondern durch eigene Interaktionsformen abgedeckt werden.

7 Modale Fenster

Während wir bei Oberflächenkonstruktionen in der Regel von flexiblen Benutzungsmöglichkeiten ausgehen, die dem Anwender eines Systems möglichst wenig vorschreiben, in welcher Reihenfolge Aktionen auszuführen sind, beschreibt dieses Kapitel die wenigen Fälle, in denen eine Ablaufsteuerung bzw. Unterbrechung des freien Ablaufs notwendig wird. Nach der Motivation dieser Anwendungssituationen stellen wir das Konzept zur Realisierung innerhalb des Interaktionsformenkonzepts vor.

Reaktive Systeme ermöglichen dem Benutzer den wahlfreien Wechsel zwischen parallel verfügbaren Arbeitskontexten in verschiedenen Fenstern. Demgegenüber bestimmen modale Fenster ein Ablaufschema und legen den Benutzer in seinen Arbeitsformen fest.

> **Modale Fenster** dienen der Steuerung des Programmablaufes. Ein modales Fenster erwartet nach seiner Ausgabe eine Benutzerreaktion. Die Anwendung wird blockiert, bis diese Eingabe erfolgt ist. Der Benutzer kann keine Eingaben in anderen Fenstern vornehmen.

In gängigen Systemen werden vielfach modale Dialogfenster eingesetzt, um Einstellungen vorzunehmen oder ein Kommando, z.B. einen Druckauftrag, näher zu beschreiben. Andere modale Fenster sind z.B. Warnmeldung an den Benutzer und Sicherheitsrückfragen mit Auswahlmöglichkeit.

Die in reaktiven Systemen übliche Möglichkeit des Benutzers, die Reihenfolge seiner Aktionen selbst zu bestimmen, wird durch modale Fenster beschränkt. Bevor wieder andere Aktionen ausgeführt werden können, muß erst das entsprechende Fenster geschlossen worden sein. Derartige Einschränkungen sind nach Möglichkeit zu vermeiden bzw. auf ein Minimum zu beschränken. Konkrete Vorschriften in Styleguides müssen jedoch beachtet werden.

Modale Fenster verändern die Benutzung einer Anwendung und beeinflussen deren Implementation. Modalität aus Sicht des Programmes ist gegeben, wenn das Programm eine gezielte Rückfrage an den Benutzer initiiert. Dies sind meist Standarddialoge mit Anzeige einer Textmeldung, auf die der Benutzer mit typischen Antwortmöglichkeiten reagieren kann, beispielsweise

- die Rückfrage vor Beenden eines Programmes „Daten sind noch nicht gespeichert! Jetzt speichern ?" (Antworten „Ja/Nein/Abbruch")
- der wichtige Hinweis „Druckauftrag konnte nicht ausgeführt werden" (Antwort nur „OK").

Komplexe modale Dialoge sind in vielen Fenstersystemen vorzufinden (z.B. Standard-Dateiauswahldialog) und werden ggf. auch durch Styleguides für bestimmte Systeme vorgeschrieben. Für das Programm selbst besteht keine Notwendigkeit, diese Dialoge modal auszuführen, auch wenn die Präsentation für den Benutzer in Form eines modalen Fensters erfolgt. So wird bei Portierung der Anwendung auf ein anderes System lediglich die Präsentationsform ausgetauscht, ohne weitere Programmänderungen.

Aufgrund dieser Unterscheidung zwischen Modalität für das Programm bzw. nur für den Benutzer, werden beide Fenstertypen unterschiedlich realisiert:

- Muß das Programm an einer Stelle auf die Benutzereingabe warten, so ist das Fenster für das Programm modal. Das realisieren wir mit einem Funktionsaufruf (IAFRequest), dessen Rückgabewert nur bei mehreren Auswahlalternativen für den Benutzer interessiert.

- Wenn der Benutzer ein Fenster abschließen muß und alle anderen blockiert sind, ist es für den Benutzer modal. Diese Modalität ist Teil der Präsentation und ohne jeden Einfluß auf die Implementation des Programmes.

Während die zweite Art also lediglich zu einer „modalen Präsentation“ führt, soll die erste Art von der Interaktionskomponente wie ein Funktionsaufruf verwendet werden. Der Kontrollfluß des Programms geht wirklich erst weiter, wenn der Benutzer seine Auswahl gemacht hat.

Damit stellen Requests eine besondere Interaktionsform dar. Soweit keine Auswahlalternativen angeboten sind, stellt ein Request eine Empfangsbestätigung des Benutzers für wichtige Nachrichten dar. Das System stellt sicher, erst nach dieser Bestätigung weitere Arbeiten zuzulassen.

Requests mit Auswahlmöglichkeit beinhalten zusätzlich eine 1 aus N Selektion. Im Gegensatz zum IAFStatic1FromNSelection ist zwingend eine Auswahl zu treffen, die in der Regel direkt zu einer Aktion führt. Insofern ist ein IAFRequest mit Auswahlmöglichkeit eher einer Gruppe von IAFActivator-Objekten vergleichbar. Auch hier ist garantiert, daß das System erst nach Aktivierung einer Option weiterarbeitet.

Folgende Requests sind vorgesehen:

7.1 *IAFMessageRequest*

`IAFMessageRequest`: einfache Meldung, die der Benutzer bestätigen muß. Soweit keine statischen Meldungen im GUI-Builder festgelegt werden können, ist es möglich, einen Meldetext festzulegen. `IAFMessageRequest` wird durch Aufruf der Prozedur `StartRequest` verwendet. Der Programmablauf wird blockiert, bis der Benutzer die Meldung bestätigt hat.

```
class IAFMessageRequest extends IAFBase
void startRequest ();
String message ();
void setMessage (String msg);
```

7.2 *IAFChoiceRequest*

`IAFChoiceRequest`: Auswahl des Benutzers zwischen mehreren Alternativen. Ergänzend zum `IAFMessageRequest` werden im GUI-Builder die verschiedene Auswahlalternativen eingerichtet und können vor Aufruf von `startRequest` ggf. deaktiviert werden. Die Funktion `startRequest` aktiviert das modale Fenster und liefert den Index der getroffenen Auswahl zurück.

```
class IAFChoiceRequest extends IAFBase
int startRequest ();
void enableItem (int index);
void disableItem (int index);
String message ();
void setMessage (String msg);
```

Während `IAFMessageRequest` z.B. darauf hinweist, daß ein für die weitere Arbeit wichtiger Auftrag nicht ausgeführt werden kann, wird der `IAFChoiceRequest` u.a. vor Abbruch des Programmes eingesetzt, um noch eine Speicherung der Daten zu ermöglichen.

Eine Alternative Konstruktion im Rahmen des IAF-Ansatzes könnte spezialisierte Requests für verschiedene Ereignisse vorsehen, z.B. wären die zulässigen Optionen (OK/Abbruch bzw. JA/Nein/Abbruch) im konkreten Request fest vorgegeben.

Dieser Entwurf einfacher Requests geht davon aus, daß die Gestaltung der zugehörigen Präsentation im GUI-Builder erfolgt. So kann auf spezielle Requests für Fehlerhinweise im Vergleich zu Meldungen über erfolgreichen Abschluß einer Aktion verzichtet werden. Soweit Meldungstexte vom dynamischen Zustand des Programmes abhängen, können diese vor der Aktivierung gesetzt werden.

In der Verwendung unterscheiden sich Requests von anderen Interaktionsformen dadurch, daß eine Anzeige nur während Ausführung der Methode `startRequest` erfolgt. Die Methode `IsEnabled` aus der Oberklasse `IAFBase` liefert folglich immer „falsch“.

Die hier genannte Verwendung von Requests ist vielen Fällen erforderlich, in flexiblen, reaktiven System soll die Verwendung modaler Dialoge aber die Ausnahme bleiben. Dies gilt auch für die Alternative modaler Präsentation für den Benutzer entsprechend Styleguide, die im Programm nicht modal realisiert wird.

8 Spezifikation der Päsentation

In den vorigen Kapiteln haben wir das Konzept der Interaktionsformen beschrieben. Dabei haben wir darauf hingewiesen, daß die Präsentation unabhängig vom eigentlichen Anwendungsprogramm festgelegt werden soll. Durch die Trennung von Interaktion und Präsentation erreichen wir eine zusätzliche Abstraktion für die Werkzeugkonstruktion. Wie schon in den vorangegangenen Abschnitten beschrieben wurde, geht das Werkzeug nur noch mit Interaktionsformen um, die für das Werkzeug den Abschluß bilden. Die Präsentationsformen werden außerhalb des Werkzeugs verwaltet. Daher wird im Werkzeug nicht mehr festgelegt, wie man beispielsweise eine Interaktionsform „`IAFActivator`“ an der Oberfläche des Werkzeuges darstellen möchte. Da man einen `IAFActivator` völlig unabhängig vom Werkzeug entweder als Button oder als MenuItem (oder beides) präsentieren kann, muß diese Spezifikation, wie die Oberfläche tatsächlich aussehen soll, außerhalb des Werkzeuges erfolgen. In diesem Abschnitt beschreiben wir, wie die Präsentation spezifiziert werden kann.

8.1 Interaktions-, Präsentationsformen und Interface Builder

Prinzipiell ist für die Spezifikation der Präsentation die Programmkomponente `PfsContext` zuständig. Dieser Kontext der Präsentationsformen verwaltet die Widgets, aus denen sich die Oberfläche eines Werkzeuges zusammensetzt und verknüpft diese mit den entsprechenden Interaktionsformen bzw. stellt eine Schnittstelle zur Verfügung, damit eine andere Komponente (der IAFsContext) diese Verbindung zwischen Präsentationsformen und Interaktionsformen herstellen kann.

Nun gibt es eine Reihe von Möglichkeiten, um mit einem PFsContext die Präsentationsformen festzulegen:

- Der GUI-Manager liest eine Ressource-Datei ein und erzeugt daraus die Widgets für die Präsentationsformen. Mit diesen Widgets parametrisiert er einen neu erzeugten PFsContext.
- Der GUI-Manager benutzt persistente Widgets, um neu erzeugte PFsContext-Objekte damit zu parametrisieren.
- Der PFsContext wird beerbt und in der neuen Subklasse wird die Erzeugung der Widgets codiert.
- Der verwendete GUI-Builder generiert Quellcode, welcher die Benutzungsoberfläche erzeugt.

Zusätzlich dazu kann an dieser Stelle die Anforderung gestellt werden, dynamische Oberflächen (wie zum Beispiel einen Desktop mit einer variablen Anzahl von Icons) mit Präsentationsformen und Interaktionsformen zu realisieren. Auch hierzu soll ein Realisierungsvorschlag gemacht werden.

8.2 Ressource-Dateien und persistente Oberflächen-Widgets

Das PF-Kontextobjekt erzeugt bei seiner Instantiierung die komplette Menge seiner PF-Objekte. Dies könnte per Hand programmiert werden, indem sich alle benötigten PF-Kontextklassen bei einer zentralen Instanz im IAF-PF-Rahmenwerk anmelden, die bei Bedarf für die Erzeugung der Kontextobjekte benutzt werden. In den beiden

„Referenzimplementationen" in C++ (vgl. [Gör98]) und in Java (vgl. [Lip97]) sind die Rahmenwerke aber so angelegt, daß die PF-Kontextobjekte serialisierte Präsentationsformen einlesen, die in einem GUI-Builder definiert und abgespeichert wurden. Dies bietet den Entwicklern eine enorme Erleichterung, da nur noch die Einbindung der Interaktionsformen im Softwarewerkzeug per Hand kodiert werden muß. Änderungen an der konkreten Oberfläche werden im GUI-Builder vorgenommen und gesichert und können direkt ausprobiert werden, ohne eine einzige Klasse des Werkzeuges neu übersetzen zu müssen.

In größeren Projekten mit einer großen Anzahl zum Teil aufwendiger Oberflächen für Werkzeuge wird es zunehmend wichtig, einen GUI-Builder zu verwenden, um die Oberflächen für die Werkzeuge zu erstellen. Diese Vorgehensweise paßt sehr gut zu dem hier vertretenden Ansatz der Trennung von Präsentation und Interaktion. Während die Interaktion im Werkzeug festgelegt wird, kann mit einem GUI-Builder die Präsentation festgelegt und verändert werden, (völlig) unabhängig vom Werkzeug. Diese Trennung macht es zudem noch möglich, wesentlich einfacher auch eine personelle Trennung zwischen diesen Aufgaben zu reaslisieren.

Damit die Oberfläche eines Werkzeuges außerhalb des Programmcodes beispielsweise mit einem GUI-Builder erzeugt werden kann, muß ein grundsätzliches Format definiert werden, um die erzeugten Oberflächeninformationen (die Ressourcen) im Programm mittels eines GUI-Managers zur Laufzeit einzulesen.

Dieses „Ressourcen“-Format kann sehr unterschiedlich ausehen. Es ist grundsätzlich denkbar, als Ressourcen-Format eine spezielle Beschreibung der Oberfläche zu benutzen, wie dies von vielen GUI-Buildern gemacht wird. Dabei wird (teilweise als Klartext) nach einer speziellen Grammatik die Oberfläche in ihren Bestandteilen und Eigenschaften beschrieben. Aus dieser Beschreibung können zur Laufzeit die entsprechenden Widgets erzeugt werden. Der GUI-Manager hat in diesem Fall die Aufgabe, die Ressource-Dateien einzulesen, sie zu interpretieren und die entsprechenden Widgets zu erzeugen. Hat er die Widgets einmal erzeugt, kann er einen PFsContext erzeugen und ihn mit den neuen Widgets parametrisieren.

Dieser Ansatz hat den Nachteil, daß die Interpretation des Ressource-Formats bei komplexen Oberflächen sehr aufwendig werden kann. Zudem stellt sich die Frage, wer das Format der Ressource-Dateien bestimmt. Wenn hierbei das von GUI-Builder erzeugte Ressource-Format verwendet wird, muß zumindest für jeden GUI-Builder spezifisch ein GUI-Manager implementiert werden, der mit dem speziellen Ressourcen-Format des GUI-Builders umzugehen weiß. Anders herum ist es sehr schwierig, aus einem GUI-Builder ein standardisiertes Ressourcen-Format zu bekommen, wenn dies in der Programmiersprache bzw. dem Programmiermodell nicht standardisiert ist.

Ein anderer Ansatz ist, die Ressourcen als persistente Oberflächenobjekte aufzufassen. Das Ressourcen-Format wäre in diesem Fall das Format für persistente Objekte. Dieser Ansatz hat den Vorteil, daß der GUI-Manager es wesentlich einfacher hat. Er liest die persistenten Oberflächenobjekte einfach aus einer Datei ein und braucht keine Widgets zu erzeugen. Die Erzeugung der Objekte wird von der Persistenzimplementierung geleistet, die wir hier voraussetzen. Das einzige Problem ist, wie die Objekte im GUI-Builder persistent gemacht werden können. Die einfachste Lösung für dieses Problem ist ein Komponentenmodell, falls dieses in der Sprache oder der Sprachumgebung vorliegt. Damit kann eine spezielle Komponente entwickelt werden, die in den GUI-Builder eingebettet die erzeugten Oberflächen direkt aus dem GUI-Builder heraus persistent macht.

Liegt kein Komponentenmodell vor, läßt sich das persistente Abspeichern der Oberflächenkomponenten über einen kleinen Testrahmen realisieren.

In jeden Fall ist die Erzeugung der Oberflächen-Widgets vor dem GUI-Manager somit vor den Werkzeugen verborgen.

8.3 Festcodierte Oberflächen

In einigen Fällen ist es nicht möglich, Oberflächen mit einem GUI-Builder zu erstellen. Gründe hierfür liegen meist in der mangelnden Funktionalität von GUI-Buildern. In diesen Fällen muß es möglich sein, die Erzeugung der Widgets direkt im Programm zu codieren, ohne daß die Werkzeuge von der Präsentation abhängig werden.

Dazu ist es möglich, den PFsContext zu beerben. In der beerben Klasse kann die Oberfläche, beispielsweise im Konstruktor, direkt erzeugt werden. Diese Subklasse von PFsContext ist nun werkzeugspezifisch. Im PfsContext wird dabei direkt auf die Widgets zugegriffen. Zum einen kann der Entwickler so auf alle Eigenschaften der Widgets zugreifen, zum anderen wird der spezielle PfsContext bei diesem Vorgehen von dem verwendeten GUI-Toolkit abhängig. Allerdings bleibt diese Abhängigkeit auf einen relativ kleinen, klar begrenzten Teil der Anwendung beschränkt.

8.4 Generierter Oberflächen-Quellcode

Die meisten GUI-Builder moderner Entwicklungsumgebungen erzeugen statt oder zusätzlich zu Ressource-Dateien Quellcode, der ausgeführt eine entsprechende Oberfläche realisiert. Die direkte Verwendung dieses Quellcodes zur Implementierung eines Werkzeugs ist nicht zu empfehlen, weil dann der GUI-Builder die Anwendungsstruktur vorgibt. Trotzdem hat die Verwendung des generierten Quellcodes den Vorteil, daß keine Inkompatibilitäten bzgl. der Ressource-Dateien bzw. serialisierter Objekte auftreten können.

Daher kapseln wir generierten Oberflächen-Quellcode in einem speziellen PFsContext, der dann für die Anbindung von Interaktionsformen an die konkreten Präsentationsformen verantwortlich ist. Wichtig ist zu beachten, daß im GUI-Builder für die später an die Interaktionsformen anzubindenden Widgets die zur Verfügung gestellten Präsentationsformen verwendet werden.

8.5 Dynamische Oberflächen

In vielen Fällen reichen die beschriebenen Mechanismen zur Spezifikation der Präsentation vollkommen aus. In einigen Fällen ist jedoch eine dynamische Oberfläche gewünscht.

> Unter einer **dynamischen Oberfläche** verstehen wir eine Benutzungsoberfläche, deren Widgets sich zur Laufzeit des Programmes verändern können.

Beispiele für solch dynamische Oberflächen sind grafische Zeichenprogramme oder elektronische Schreibtische (Desktops). Im ersten Fall ändern sich die gezeichneten Objekte, im zweiten die Piktogramme auf dem elektronischen Schreibtisch.

Um dynamische Oberflächen mit Präsentationsformen zu realisieren, bietet sich die recht einfache Möglichkeit an, den PFsContext zu beerben und in der neuen Subklasse eine entsprechende Funktionalität hinzuzufügen. So ist dies beispielsweise bei der Implementation des Desktops im JWAM-Rahmenwerk geschehen, der eine variable Anzahl von Icons als Präsentationsformen handhaben muß.

Prinzipiell kann die Subklasse beim dynamischen Erzeugen der Präsentationsformen (bzw. Widgets) zwei unterschiedliche Strategien verfolgen, die sich an der Schnittstelle äußern:

Zum einen kann die Subklasse die Funktionalität für das Werkzeug transparent realisieren. In diesem Sinne stellt der neue PFsContext fest, wenn das Werkzeug eine Interaktionsform an eine Präsentationsform anbinden möchte und zu dieser Interaktionsform aber noch keine Präsentationsform existiert. Hat er dies erkannt, kann er daraufhin eine passende Präsentationsform erzeugen und diese an die Interaktionsform koppeln. Dieser Ansatz birgt den Vorteil, daß er vor dem Werkzeug verborgen ist. Es bemerkt nicht, ob und wann eine neue Präsentationsform erzeugt werden muß oder erzeugt wird. Zum anderen ist dieser Ansatz jedoch problematisch, da es dem PFsContext überlassen bleibt, welches konkrete Widget erzeugt wird. Stellt der PFsContext beispielsweise fest, daß zu einem IAFActivator noch kein passendes Widget existiert, kann er dafür entweder einen Button oder ein MenuItem erzeugen. Es mag durchaus vorkommen, daß diese Entscheidung durch den nun speziellen PFsContext sinnvoll gefällt werden kann, da er in der Regel speziell für ein bestimmtes Werkzeug implementiert wird. Wesentlich schwieriger wird es, wenn an eine Präsentationsform mehrere Interaktionsformen gebunden werden müssen. Hierfür muß eine angemessene algorithmische Lösung gefunden werden.

Die zweite Strategie ist, daß an der Schnittstelle des speziellen PFsContext explizit Methoden zur Verfügung gestellt werden, um die Dynamik innerhalb der Oberfläche zu realisieren. Beispielsweise kann der PFsContext für einen Desktop eine Methode „addIcon“ bereitstellen, die eine entsprechende Präsentationsform erzeugt und in die Oberfläche eingliedert. Dadurch ist die Dynamik der Oberfläche nicht mehr transparent für das Werkzeug, bietet aber den Vorteil, daß die oben beschriebenen Nachteile der transparenten Realisierung nicht mehr auftreten.

Die beiden Strategien unterscheiden sich nur in ihrem Abstraktionsgrad. Es kann aber davon ausgegangen werden, daß es nicht wirklich eine Rolle spielt, welche Variante verwendet wird. Auch wenn im ersten Fall die konkrete Präsentationsform-Erzeugung für das Softwarewerkzeug transparent ist, kann man in beiden Fällen davon ausgehen, daß das zugehörige Softwarewerkzeug Annahmen darüber macht, welche Präsentationsformen erzeugt werden. Schließlich ist ein hier beschriebener PFsContext für dynamische Oberflächen exakt für ein Sotwarewerkzeug erstellt.

9 Implementationsansätze

In diesem Abschnitt wollen wir anhand unserer Implementationsansätze präsentieren, wie die in den vorigen Kapiteln vorgestellten Konzepte umgesetzt werden können. Dazu dienen uns zwei Implementationen, die im Rahmen einer Studien- und einer Diplomarbeit entstanden sind. Zunächst gehen wir auf die Implementation von Interaktions- und Präsentationsformen in C++ ein (siehe [Gör98]). Im Anschluß daran werden wir eine Java-Implementation vorstellen, wie sie im JWAM-Rahmenwerk umgesetzt wurde (siehe [Lip97]). Wir stellen in diesem Abschnitt nur die Ansätze dar und beschreiben den Entwurf, der für die jeweilige Implementation gewählt wurde. Quelltext der entsprechenden Implementation soll hier nicht gezeigt werden. Das würde den Rahmen sprengen. Der Quelltext der Java-Implementation kann im JWAM-Rahmenwerk begutachtet werden.

9.1 Realisierung in C++

Die C++-Implementation wurde im Rahmen einer Diplomarbeit angefertigt (siehe [Gör98]) und ist in ein C++-Rahmenwerk eingebettet. Als Grundlage der Implementation diente das wxWindows-Toolkit (vgl. [Sma97]). Wir gehen hier davon aus, daß die wesentlichen Grundlagen von C++ sowie von GUI-Toolkits wie z.B. wxWindows bekannt sind.

9.1.1 Interaktionsformen

Im C++-Rahmenwerk sind die Interaktionsformen in einem Vererbungsbaum angeordnet. Die hier beschriebenen Interaktionsformen orientieren sich an dem oben vorgestellten Katalog.

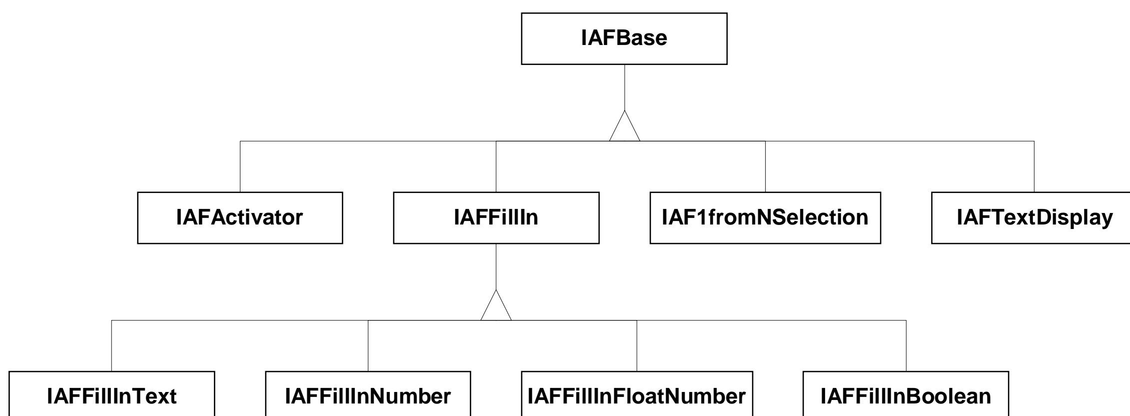

Abbildung 4: IAF-Klassenhierarchie der C++-Implementation

In der Oberklasse `IAFBase` wird mit den Methoden `Enable` und `Disable` ein Protokoll definiert, um IAF-Objekte (de-)aktivieren zu können. So läßt sich der Zustand des Werkzeuges berücksichtigen, indem einzelne Handhabungsmöglichkeiten gezielt gesperrt werden können. Die oben dargestellte Klassenhierarchie zeigt die im C++-Rahmenwerk realisierten Interaktionsformen. Zur Zeit ist dies nur eine Teilmenge der oben identifizierten Interaktionsformen.

9.1.2 Präsentationsformen

In den konkreten PF-Klassen wird letztlich die Anbindung an das verwendete GUI-Toolkit durchgeführt. Die PF-Objekte werden mit passenden IAF-Objekten innerhalb eines einbettenden Softwarewerkzeuges gekoppelt, so daß die Handhabung eines Werkzeuges mit

seiner Präsentation verbunden wird. Widgets, die keinen konkreten fachlichen Umgang mit dem Werkzeug realisieren (z.B. zu reinen Darstellungszwecken), können mit keiner Interaktionsform verbunden werden. Sie sollen aber trotzdem als Präsentationsformen bezeichnet werden.

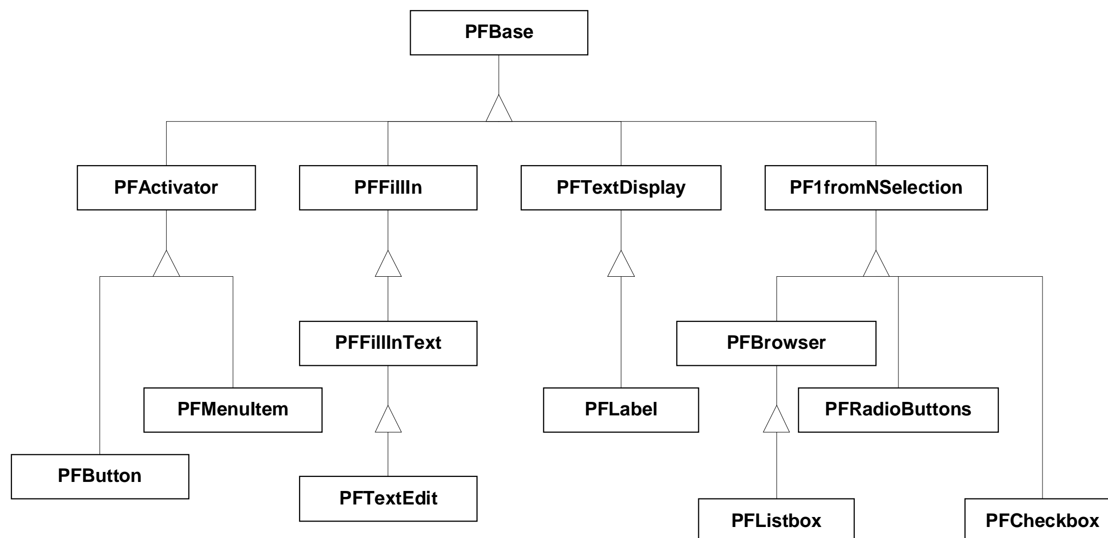

Abbildung 5: Teil eines PF-Klassenbaumes mit abgeleiteten Klassen zur Toolkit-Anbindung

Jedes PF-Objekt kapselt ein Widget. Die Konsequenz ist, daß für jedes Widget eine konkrete PF-Klasse existiert. Die Hierarchie der PF-Basisklassen entspricht exakt dem IAF-Klassenbaum. Dadurch wird eine mögliche Benutztbeziehung zwischen Interaktions- und Präsentationsform über die gleiche Position der PF-Basisklasse im deckungsgleichen Klassenbaum definiert. Das Rahmenwerk stellt eine Zuordnung zwischen IAF-Objekten und PF-Objekten her, wenn der Typ einer Interaktionsform zum Basistyp eines konkreten PF-Objektes paßt. Die Widgets werden in die PF-Hierarchie per Adaptermuster eingebettet, wie es im folgenden Abschnitt erläutert wird. So wird der PF-Klassenbaum jeweils für jedes spezielle GUI-Toolkit um zahlreiche konkrete PF-Klassen erweitert, die eine Verbindung zu den Klassen des GUI-Toolkits herstellen müssen.

Für PF-Objekte, deren Typ lediglich in die Kategorie „Layout und Erscheinungsbild" fällt, kann kein passendes IAF-Pendant gefunden werden. Es wird auch keine PF-Basisklasse angeboten, da diese nur für die Verknüpfung mit den Interaktionsformen benötigt werden. Damit werden diese Präsentationsformen vollständig vom einbettenden Werkzeug entkoppelt.

Insgesamt gibt es drei Möglichkeiten, zwischen IAF- und PF-Objekten eine Benutztbeziehung herzustellen:

- Im einfachsten Fall benutzt ein IAF-Objekt ein PF-Objekt. Ein Beispiel wäre die Zuordnung eines IAFFillIn mit einem PF-Objekt, das ein Widget für Texteingabe kapselt.
- Die zweite Möglichkeit besteht darin, daß ein IAF-Objekt mehrere PF-Objekte benutzt. Das IAF-Objekt ist beispielsweise ein IAFActivator, mit dem das Schließen des Werkzeuges initiiert werden soll. Dieses kann mit zwei PF-Objekten verbunden werden, die einen Menüeintrag „Schließen" und den Schließenknopf eines Fensters kapseln.
- Im dritten Fall können mehrere IAF-Objekte verschiedenen Typs auf ein PF-Objekt zugreifen. Dies ist aber nur möglich, wenn die PF-Klasse mehrere Handhabungskategorien abdeckt, für die jeweils das Protokoll einer abgeleiteten PF-Basisklasse implementiert wird. Das typische Beispiel ist ein PF-Objekt, das eine Listbox kapselt und gleichzeitig von einem IAFActivator und einem IAFBrowser benutzt wird.

9.1.3 Diskussion der Architektur von IAF und PF

Durch die Aufteilung in Interaktions- und Präsentationformen kommt in der Anbindung des GUI eine weitere Indirektion zwischen Softwarewerkzeug und GUI-Toolkit hinzu, die im Vergleich mit anderen Architekturen erhebliche Änderungen nach sich zieht.
Die IAF-Klassen stellen eine Adapterfunktionalität (vgl. [GHJ+94]) zur Verfügung, mit der das Softwarewerkzeug über einen IAF-Objektadapter in der ersten Indirektion mit einem PF-Objekt verbunden wird. Dabei muß die Möglichkeit berücksichtigt werden, daß über diesen Adapter auch eine Verbindung zu mehreren PF-Objekten hergestellt werden kann (z.B. eine `IAFActivator`, die mit einem Button und einem Menüeintrag verbunden wird, die dieselbe Aktion auslösen).

Für die Realisierung der zweiten Indirektion zwischen Präsentationsform und Widget bestehen nun zwei Möglichkeiten: die Verbindung wird über einen weiteren Objektadapter hergestellt oder es wird ein Klassenadapter benutzt. In beiden Fällen muß für jedes Widget mindestens eine PF-Klasse geschrieben werden.

Erfolgt die Anbindung des Toolkits über einen Klassenadapter, werden die konkreten PF-Klassen mit Mehrfachvererbung von einer PF-Basisklasse und von der Widget-Klasse abgeleitet. Dabei müssen immer Methoden der PF-Basisklasse redefiniert werden, um die Funktionalität der IAF-Klientenklasse und der adaptierten Widget-Klasse zusammenzuführen. Häufig müssen zudem auch noch Methoden der Toolkit-Klasse überschrieben werden, z.B. um den Callback-Mechanismus des GUI-Toolkits einzubinden.

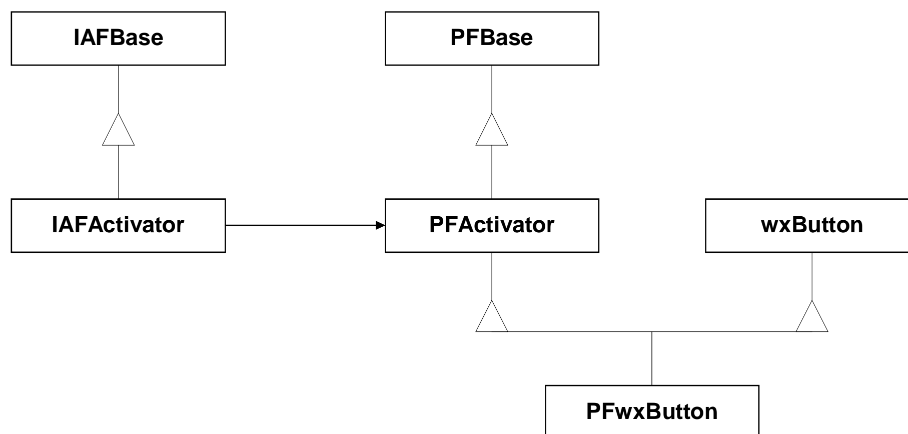

Abbildung 6: Benutzung eines PFKlassenadapters

In Abbildung 6 wird die Anbindung der Klasse IAFActivator an eine Widget-Klasse des Toolkits „WxWindows" (vgl. [Sma97]) dargestellt. IAFActivator ist in diesem Fall der Adapter für ein oder mehrere Objekte, deren Typ von PFActivator abgeleitet wurde.

Beim Einsatz von Objektadaptern muß für jede mögliche Verbindung zwischen den Widget-Klassen und den Interaktionsformen mindestens eine PF-Klasse geschrieben werden, mit der ein PF-Objektadapter erzeugt werden kann. Die Widget-Klassen dienen dabei nicht als Basisklassen, sondern werden lediglich benutzt. Die konkreten PF-Klassen werden nur von den PF-Basisklassen abgeleitet, so daß es zu einer einfachen Vererbungsbeziehung kommt.

Die Beziehung zwischen Interaktions- und Präsentationsformen ähnelt in beiden Fällen dem Proxy-Muster (vgl. [GHJ+94]), da sich weite Teile der Schnittstellen von Interaktions- und zugehörigen Präsentationsformen überschneiden. Die Interaktionsformen reichen im wesentlichen die Methodenaufrufe an die PF-Objekte nur durch, um ihrem Softwarewerkzeug keinen direkten Zugriff auf die Präsentationsform zu gewähren. Mit der Möglichkeit der

Aggregation von PF-Objekten bekommt die Architektur zusätzlich den Charakter eines Kompositionsmusters (vgl. ebenfalls [GHJ+94]). Dabei kann vor dem Softwarewerkzeug verborgen werden, wie groß die Anzahl der Präsentationsformen ist, die mit einer Interaktionsform verbunden sind. Sind es mehrere, muß die Interaktionsform intern für die Synchronisation ihrer Präsentationsformen sorgen.

Auffällig ist sicherlich der Einsatz der Mehrfachvererbung im PF-Klassenadapter, die natürlich begründet werden muß, da alternativ Objektadapters benutzt werden könnten. Tatsächlich wurden in einer ersten Implementation auch PF-Objektadapter benutzt, und es sprachen mit Blick auf die Klassenstruktur keine nennenswerten Argumente dagegen.

Man könnte die größere Zahl der Objekte zur Laufzeit bemängeln, da für die Verbindung von IAF- und Widget-Objekten immer noch ein PF-Adapter erzeugt werden müßte. Die Menge der konkreten PF-Klassen würde sich allerdings nicht wesentlich vergrößern. Nur wenn ein Widget mehr als eine Handhabungsmöglichkeit bietet (wie z.B. die Listbox), müßten für diese Toolkit-Klasse mehrere Objekt-Adapterklassen geschrieben werden. Mit dem Klassenadapter bleibt es hingegen bei einer Adapterklasse pro Widget. Das führt allerdings zu einer Mehrfachvererbung mit mindestens drei Basisklassen.

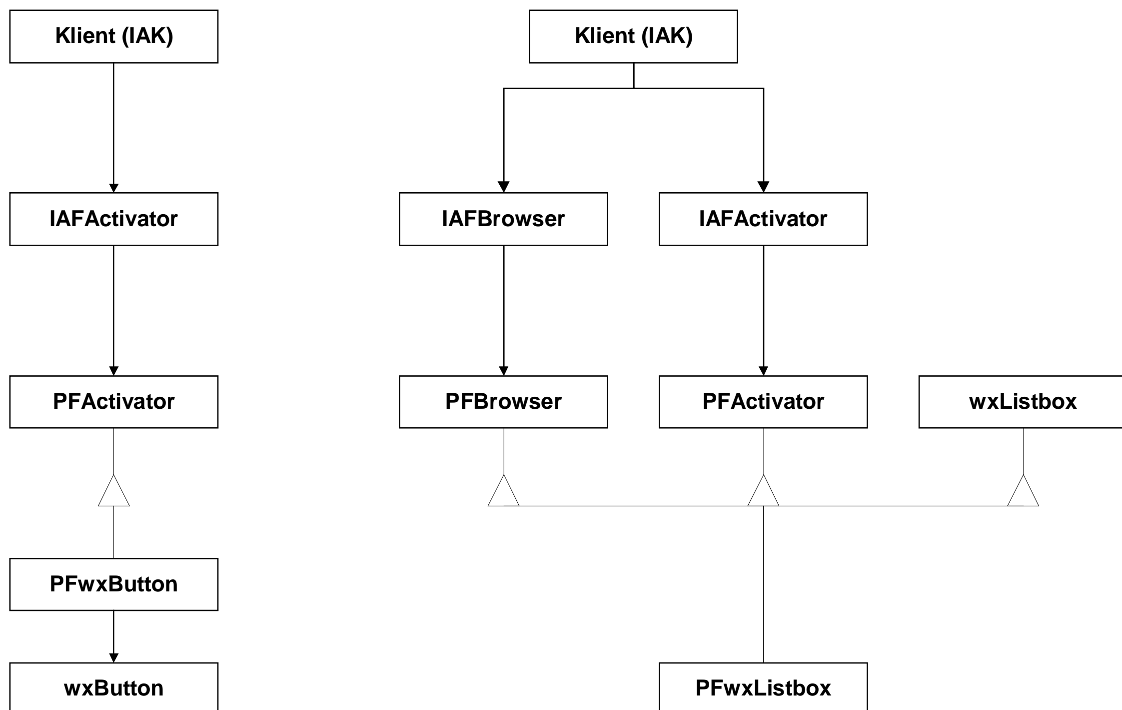

Abbildung 7: PF-Objektadapter und Klassenadapter mit drei Basisklassen

Die für einen Klassenadapter benötigte Mehrfachvererbung, die in Java (und anderen Sprachen ohne Mehrfachvererbung) im Prinzip nicht ohne weiteres realisiert werden kann, wurde durch die Benutzung von Interfaces erreicht. Die PF-Basisklassen wurden alle als Interfaces (rein virtuelle Klassen) entworfen, deren Protokolle in den konkreten PF-Klassen implementiert werden.

Die Widget-Klassen bilden jeweils die Oberklasse für eine konkrete PF-Klasse und werden so durch Implementationsvererbung eingebunden. Durch diese Verknüpfung mit der Widget-Klasse können die Schnittstellen der einzelnen PF-Klassen ziemlich umfangreich werden. Dies ist aber vertretbar, da die Präsentationsformen komplett im Rahmenwerk gekapselt werden und ein Entwickler nur die Interaktionsformen benutzt, mit den Schnittstellen der Präsentationsformen also nicht in Berührung kommt.

Mit der Implementationsvererbung wird letztlich die technische Anbindung der PF-Objekte an das GUI-Toolkit erreicht. I.d.R. müssen für das Protokoll der Widget-Klassen in den PF-Klassen Callback-Methoden definiert oder überschrieben werden, um die Anbindung an den Ereignismechanismus des GUI-Toolkits zu realisieren. Außerdem werden all die Methoden geerbt, mit denen die Attribute für das Aussehen der Präsentationsformen in der Benutzungsschnittstelle manipuliert werden können. Und erst hier wird die hierarchische Anordnung der Widgets umgesetzt, die bei der konventionellen GUI-Anbindung bereits im Softwarewerkzeug definiert werden mußte. Darauf kann i.d.R. nicht verzichtet werden, so daß die PF-Objekte dies - mit den Möglichkeiten der abgeleiteten Widget-Klassen - realisieren müssen.

9.1.4 Dynamik zur Laufzeit

Bis jetzt wurde das Konzept der Interaktions- und Präsentationsformen erläutert und eine mögliche Architektur dargestellt. Nun soll beschrieben werden, wie ein Softwarewerkzeug die Interaktionsformen benutzen kann und wie sich diese im Zusammenspiel mit den Präsentationsformen verhalten.

Interaktionsformen werden innerhalb eines Softwarewerkzeuges erzeugt und benutzt. Dies ist notwendig, damit das Softwarewerkzeug seinen Zustand und des des Materials weitergeben kann. Außerdem müssen Eingaben des Benutzers an das Softwarewerkzeug gemeldet werden. Anders sieht es mit den Präsentationsformen aus. Für diese wurde bisher immer nur die möglichst lose Kopplung mit dem Softwarewerkzeug betont. In den dargestellten Klassendiagrammen kann man erkennen, daß die IAF-Klassen die PF-Klassen benutzen. Gleichzeitig wurde aber nur erwähnt, daß die PF-Objekte außerhalb des Softwarewerkzeuges verwaltet werden. Wie also werden die Interaktionsformen eines Softwarewerkzeuges mit ihren Präsentationsformen verbunden?

Die Interaktionsformen können eindeutig identifiziert werden, da ihnen innerhalb eines Softwarewerkzeuges zusätzlich zur Objektidentität ein eindeutiger Bezeichner zugewiesen wird. Daneben besitzen auch die Präsentationsformen einen Bezeichner. Mit Hilfe dieser beiden Bezeichner wird die Zuordnung von Interaktions- und Präsentationsform hergestellt. Ein IAF-Objekt wird mit einem PF-Objekt genau dann verbunden, wenn ihre Typen im IAF-PF-Rahmenwerk als zusammengehörig definiert wurden und beiden Objekten der gleiche Bezeichner zugeordnet wurde. Wir nennen diesen Bezeichner hier „Interaktionsname". Die mögliche Zuordnung von IAF- und PF-Klassen wurde im Kapitel über die Architektur implizit dadurch gezeigt, daß der Teil des PF-Klassenbaumes, der die Basisklassen umfaßt, deckungsgleich mit dem IAF-Klassenbaum dargestellt wurde.

Beispielsweise kann ein `IAFActivator` mit einem `PFButton` verbunden werden, da die Klasse PFButton von der virtuellen Klasse PFActivator abgeleitet wird. Beide müssen aber den gleichen Bezeichner haben, der für das IAF-Objekt im Softwarewerkzeug gesetzt wird. Soll der IAF-Activator noch mit einem zweiten PF-Activator verbunden werden, beispielsweise mit einem Objekt vom Typ PFMenuItem, muß dieses nur den gleichen Bezeichner wie das erste PF-Objekt besitzen.

Für den Fall, daß eine Präsentationsform mehr als eine Handhabungsform umsetzt und mit mehreren Interaktionsformen verbunden werden kann, müssen für diese Präsentationsform auch mehrere Bezeichner verwaltet werden. Das Softwarewerkzeug kann bei der Benutzung nicht unterscheiden, ob mehrere IAF-Objekte auf ein und dasselbe PF-Objekt zugreifen. Als Beispiel kann die konkrete Adapterklasse für eine Listbox dienen, die sowohl von der Klasse PFActivator als auch von der Klasse PFBrowser erbt und jeweils einen Bezeichner für den Activator und für den Browser verwalten muß.

9.1.5 Präsentationsformen im Kontext eines Werkzeuges

Fachlich gesehen schließen die Interaktionsformen ein Werkzeug ab. Die Interaktionskomponente kann komplett ohne die Kenntnis einer konkreten Benutzungsoberfläche entworfen werden. Sie definiert mit der Auswahl der Interaktionsformen, welche Möglichkeiten der Interaktion das Werkzeuges bietet. Nun muß noch eine Möglichkeit geschaffen werden, wie die Präsentationsformen außerhalb des Werkzeuges verwaltet werden.

Für diesen Zweck wurden die beiden Klassen IAFsContext und PFsContext eingeführt. Damit das IAF-PF-Rahmenwerk Zugriff auf die IAF-Objekte erhält, die in einem Softwarewerkzeug erzeugt werden, wird für jedes neue Softwarewerkzeug im Basiskonstruktor automatisch ein Objekt vom Typ IAFsContext erzeugt, das den Kontext für die IAF-Objekte verwaltet. Eigentlich würde die Benutztbeziehung zwischen Softwarewerkzeug und Interaktionsform genügen, um einen Kontext für die Interaktionsformen zu bilden. In der technischen Umsetzung wird dieses zusätzliche Kontextobjekt aber für die Verknüpfung der IAF-Objekte mit den PF-Objekten benötigt, die das Softwarewerkzeug nicht durchführen kann.

Im Rahmenwerk erzeugt der Basiskonstruktor eines Softwarewerkzeuges das IAF-Kontextobjekt, das intern zusätzlich ein Objekt vom Typ PFsContext erzeugt. Das Softwarewerkzeug muß im Konstruktor jeder Interaktionsform das IAF-Kontextobjekt als Parameter übergeben. Bei diesem registrieren sich die Interaktionsformen, so daß das IAF-Kontextobjekt Kenntnis über alle IAF-Objekte eines Softwarewerkzeuges bekommen kann.

Während der Registrierung der Interaktionsformen stellt das IAF-Kontextobjekt über das zugehörige PF-Kontextobjekt die Verbindung zwischen IAF- und PF-Objekten her. Wie beschrieben, müssen IAF- und PF-Objekte einen zusammengehörigen Basistyp und den gleichen Bezeichner besitzen. Diese komplizierte Initialisierung führt komplett das Rahmenwerk durch, wenn eine Interaktionskomponente erzeugt wird, so daß der Anwendungsentwickler sich darum nicht zu kümmern hat.

Nachdem die IAF- und PF-Objekte erzeugt und verknüpft wurden, muß das Softwarewerkzeug seine Interaktionsformen initialisieren. Die Materialien werden in Werte von Basisdatentypen konvertiert, die von den standardisierten Interaktionsformen bzw. Präsentationsformen verarbeitet werden können. Diese Werte werden dann den IAF-Objekten übergeben, die sie an die PF-Objekte durchreichen. Die Präsentationsformen stellen die Werte an der Oberfläche dar und bieten ggf. auch die Möglichkeit, sie zu verändern.

Für die Verarbeitung von Ereignissen werden Kommando-Objekte nach dem Befehlsmuster (vgl. [GHJ+94]) benutzt. In der Initialisierungsphase erzeugt das Softwarewerkzeug Kommando-Objekte, die den Interaktionsformen für die verschiedenen Ereignisse übergeben werden. Dieses Vorgehen kommt vor allem zum Einsatz, wenn IAF-Aktivator-Objekte benutzt werden. Zudem wurde vorgesehen, daß auch IAF-Objekte, die einen Wert manipulieren, ebenfalls Kommando-Objekte benutzen, um eine Änderung des Wertes signalisieren zu können.

9.2 Realisierung des GUI-Rahmenwerkes in Java

In diesem Abschnitt legen wir eine Java-Implementationsskizze der in dieser Arbeit vorgestellten Ideen und Konzepte dar. Die hier skizzierte Java-Implementation wurde für das JWAM-Rahmenwerk des Arbeitsbereiches Softwaretechnik erstellt und wird in diesem Rahmen seit 1,5 Jahren erfolgreich in der Lehre sowie für Studien- und Diplomarbeiten eingesetzt. Eine ausführliche Beschreibung der Implementation von IAF und PF findet sich in

[Lip97]. Wir gehen davon aus, daß der Leser mit grundsätzlichen Konzepten der Programmiersprache Java vertraut ist und verweisen für weitere Informationen auf [Fla97].

Wir beschreiben im folgenden den grundsätzlichen Aufbau der Implementation in Java und die Klassenstruktur, die dazu gewählt wurde. Dazu wird die Einbettung in die Java-Umgebung beschrieben sowie die Umsetzung der Interaktions- und Präsentationsformen in Java.

9.2.1 Ausgangspunkt Toolkit: Das AWT

Eine wesentliche Besonderheit ist, daß Java das AWT (Abstract-Window-Toolkit) mitbringt. Diese Standard-Bibliothek bildet die Grundlage, in Java plattformunabhängige Applikationen oder Applets zu entwickeln, die über eine graphische Benutzeroberfläche verfügen. Da das AWT zu den Standard-Packages in Java gehört, ist es auf allen Java-Plattformen verfügbar.

Allerdings hat die Entwicklung des AWT von JDK1.0 zu JDK1.1 gezeigt, daß auch das AWT Änderungen unterliegt, zum Teil sogar erheblichen[6]. Mit der Einführung des JDK 1.2 im Januar'99 gehören die Swing-Klassen zu den Standard-Packages. Diese Klassen bieten eine Alternative zu den AWT-Klassen. Es ist davon auszugehen, daß die Swing-Klassen die AWT auf lange Sicht ablösen. Der Wechsel von JDK 1.0 nach JDK 1.1 und der Wechsel von AWT nach Swing zeigen deutlich, daß auch die Standard-Packages von Java einer Weiterentwicklung unterliegen und man sich für künftige Entwicklungen offen halten muß.

Das macht es sinnvoll, bei der Realisierung eines GUI-Rahmenwerkes in Java zwar auf dem AWT aufzubauen, sich jedoch nicht von dem AWT abhängig zu machen. Eine Kapselung der AWT-Klassen ist sinnvoll, um bei späteren Änderungen der AWT-Klassen durch Sun oder bei einem Umstieg auf neuere GUI-Toolkits (wie Swing) flexibel und mit wenig Aufwand reagieren zu können.

9.2.2 Interaktionsformen in Java

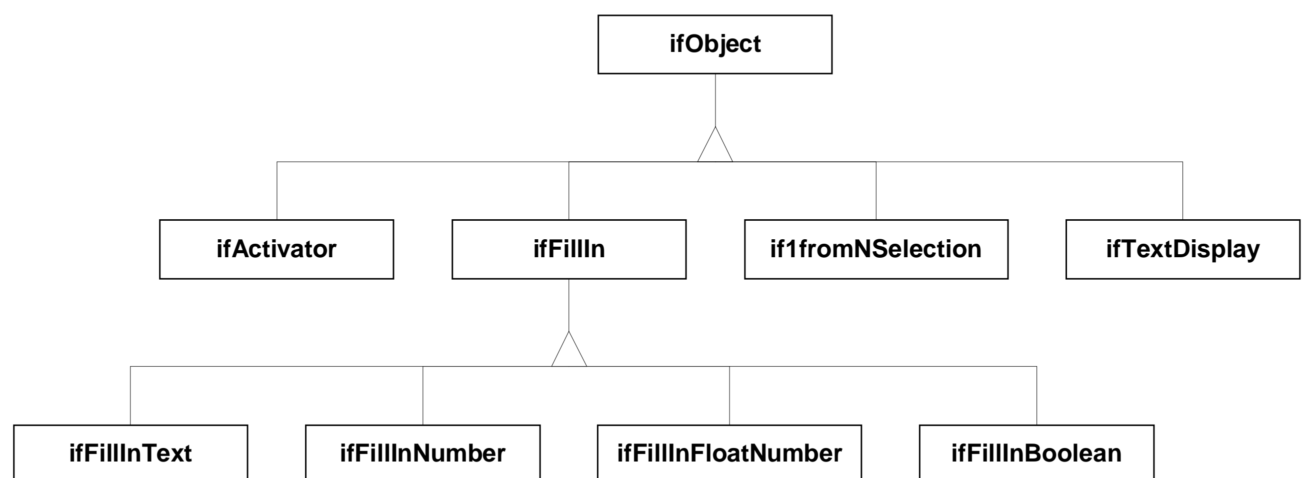

Abbildung 8 Die Java-Klassenhierarchie

Die Implementierung der Interaktionsformen in Java ist durch Java-Klassen umgesetzt. Wie Abbildung 8 darstellt, existiert eine Hierarchie von Interaktionsformen, ausgehend von einer Basisklasse `ifObject`. An ihrer Schnittstelle bieten sie die Möglichkeit, Kommando-Objekte anzumelden, um eine lose Kopplung an das Softwarewerkzeug zu realisieren. Die Benennung

[6] Der Compiler gibt bei der Compilierung alter Sourcen „deprecated"-Warnings aus, deren Anzahl sehr groß sein kann.

der Klasse weicht hier von der oben vorgestellten leicht ab. Die Benennung der Oberklasse als „Object“ orientiert sich an dem Java-Standard-Klassenbaum. Das Kürzel „if“ ist hier gewählt worden anstelle der oben verwendeten Abkürzung „IAF“, weil wir im JWAM-Rahmenwerk möglichst mit zwei Stellen für ein prefix auskommen wollen.

Diese Interaktionsform-Objekte verweisen auf möglicherweise mehrere Präsentationsformen. An diese Präsentationsformen sind die Interaktionsformen über einen Kommando-Mechanismus (siehe [GHJ+94], [RW96], [Zül98]) gekoppelt. Auf Kommandos von den Präsentationsformen reagieren die Interaktionsformen zunächst mit einer Synchronisation der Präsentationsformen, sofern nötig. Anschließend leiten sie die Aktion an das Softwarewerkzeug weiter, indem das vom Softwarewerkzeug angemeldete Kommando ausgeführt wird.

9.2.3 Präsentationsformen in Java

Aufgabe der Präsentationsformen ist zum einen, verschiedene Darstellungen zu einer Interaktionsform zu ermöglichen und zum anderen, die Widgets des GUI-Toolkits anzubinden, auf die gezwungenermaßen zurückgegriffen werden muß.

Da die Widgets der GUI-Toolkits in der Regel jedoch nicht am Umgang orientiert, sondern zum Teil in recht kuriosen Hierarchien angeordnet sind, müssen die Präsentationsformen hier eine Adaption vornehmen. Das bedeutet, eine Präsentationsform, die eine Listbox benutzt, muß andere Methoden an dem Widget rufen, als eine Präsentationsform, die eine Choice benutzt, auch wenn an beiden Widgets die gleiche Aktion durchgeführt werden soll.

Deshalb sind die Präsentationsformen in zwei Hierarchien unterteilt. Zu den Interaktionsformen existiert eine Hierarchie von Präsentationsform-Interfaces in der Form, daß zu jeder Interaktionsform ein Präsentationsform-Interface existiert. Diese Präsentationsform-Interfaces sind in ihrer Hierarchie genauso wie die Interaktionsformen angeordnet. Diese Präsentationsform-Interfaces (z.B. `pfActivator` oder `pf1fromNSelection`) stellen für die Interaktionsformen ein einheitliches Interface zur Verfügung. Die Interaktionsform `ifActivator` benutzt beispielsweise nur Präsentationsformen vom Typ `pfActivator` (siehe Abbildung 9).

Hinter diesen Interfaces können nun verschiedene Präsentationsform-Klassen stehen, die das Präsentationsform-Interface implementieren. Diese Präsentationsform-Klassen adaptieren die am Umgang orientierten Methoden auf die entsprechenden Methoden des Widgets und müssen so speziell für jedes Widget implementiert werden.

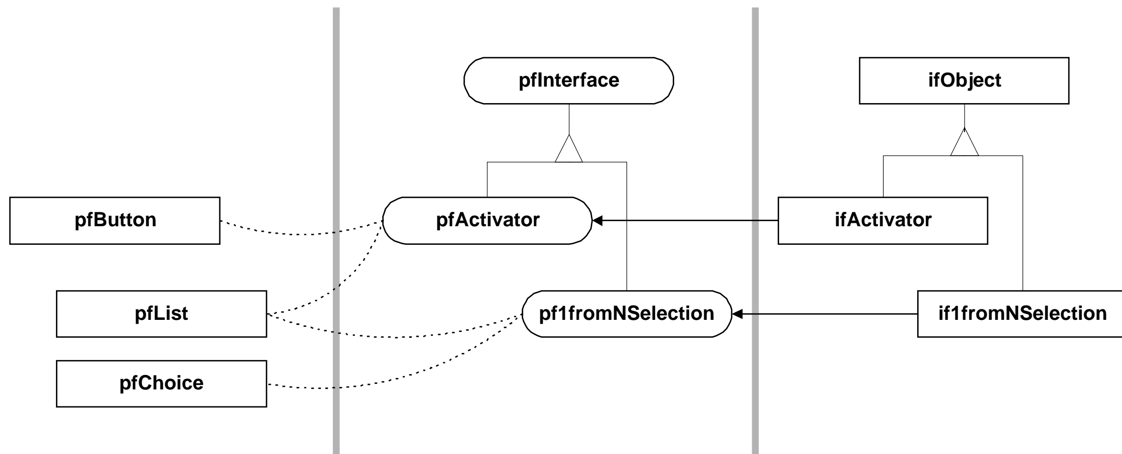

Abbildung 9: Eine Hierarchie von Interaktionsformen (rechts), die die entsprechenden Präsentationsform-Interfaces benutzen (mitte), die von den Präsentationsform-Klassen implementiert werden (links). Die grauen Linien trennen die Schichten des Frameworks.

Die Präsentationsformen an die Widgets anzubinden, stellt mit dieser Lösung keine Schwierigkeit dar. Prinzipiell gibt es dafür zwei Möglichkeiten. Zum einen könnten die Präsentationsform-Klassen die Widgets benutzen. Dies wäre eine saubere Trennung. Allerdings führt diese Trennung dazu, daß eine zusätzliche Schicht in der Abstraktion eingeführt wird. Durch die Trennung in Präsentationsform-Interfaces und Präsentationsform-Klassen besteht das Rahmenwerk aus drei Schichten (Interaktionsform-Klassen, Präsentationsform-Interfaces, Präsentationsform-Klassen).

Zum zweiten bietet sich in Java die Möglichkeit an, die Präsentationsform-Klassen von den Widget-Klassen erben zu lassen (siehe Abbildung 10). Dies wird dadurch ermöglicht, daß die AWT-Klassen schon in einer objektorientierten Klassenbibliothek organisiert sind und im Prinzip schon die konkreten Widgets des Systems kapseln. Da eine solche Konstruktion innerhalb des Rahmenwerkes liegt und vor dem Benutzer des Rahmenwerkes komplett verborgen ist, halten wir eine solche Konstruktion für vertretbar. Bei der Konstruktion mit Java hat diese Konstruktionsweise eine Reihe von Vorteilen. Der wohl wichtigste ist die Tatsache, daß sich diese von den AWT-Klassen erbenden Präsentationsform-Klassen direkt in graphischen User-Interface-Buildern verwenden lassen. So ist der Applikationsentwickler in der Lage, im GUI-Builder direkt Präsentationsformen anzuordnen. Ebenfalls im GUI-Builder können dann nicht nur die Eigenschaften der Widgets (Farbe, Größe, etc.) eingestellt werden, sondern ebenso einfach Eigenschaften der Präsentationsform-Objekte. So kann im GUI-Builder direkt der „Interaktionsname" vergeben werden, unter dem die Präsentationsform dem Rahmenwerk bekanntgibt, für welche Interaktion diese Präsentation steht. Über diesen Namen wird die Präsentationsform vom Rahmenwerk an die Interaktionsform angebunden. Die AWT-Klassen besitzen zwar schon einen Namen, dieser ist aber nicht für den „Interaktionsnamen" zu verwenden. Die Namen der AWT-Objekte müssen eindeutig sein.

Das direkte Erben von den AWT-Klassen bedeutet natürlich, daß die Schnittstelle der PF-Klassen enorm aufgebläht ist, da die komplette Schnittstelle der AWT-Klassen hineingeerbt wird. Das sind in der JDK-Version 1.1 immerhin mindestens 130 Methoden (schon von der Basisklasse `java.awt.Component`). Jedoch muß die Schnittstelle der PF-Klassen dem Benutzer des Rahmenwerkes nicht bekannt sein. Bei der Werkzeugentwicklung werden ausschließlich Interaktionsformen verwendet. Diesen Interaktionsformen sind die Präsentationsformen nur durch die Präsentationsform-Interfaces bekannt, die eine sehr schmale Schnittstelle darstellen. Auch bei der Verwendung im GUI-Builder, indem die Präsentationsform-Klassen direkt benutzt werden, um die Oberfläche zusammenzusetzen, ist

die Kenntnis der Klassenschnittstelle nicht nötig. Das visuelle Zusammenstellen der Oberfläche im GUI-Builder geschieht mittels Drag & Drop.

Insofern ist diese aufgeblähte Schnittstelle der Präsentationsformen für den Benutzer des Rahmenwerkes nicht von Bedeutung. Sie ist nur innerhalb des Rahmenwerkes bekannt. Deshalb ist diese Konstruktion vertretbar.

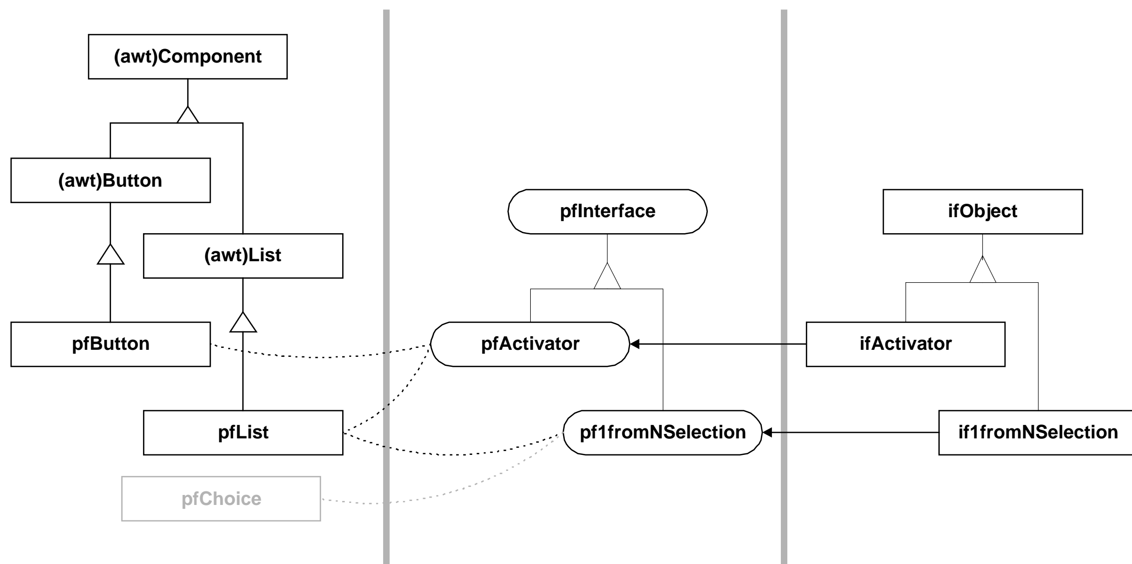

Abbildung 10: Die Präsentationsform-Klassen erben von den AWT-Klassen (links).

Da eine Präsentationsform mehrere Interaktionen darstellen kann, muß beispielsweise die Präsentationsform-Klasse `pfList` sowohl das PF-Interface `pfActivator` als auch das PF-Interface `pf1fromNSelection` implementieren (siehe Abbildung 9 und Abbildung 10). Dies ist deshalb kein Problem, da gleichbenannte Methoden in den beiden Interfaces nur dann vorkommen, wenn sie die gleiche Aktion darstellen. Dies ist beispielsweise für die Methoden der Fall, die beide Interfaces durch ihr Basisinterface `pfInterface` „geerbt" haben. Hier finden sich Methoden wie `hide()` und `show()`, welche die Präsentationsform an der Oberfläche sichtbar oder unsichtbar machen. Zwar werden die Methoden von beiden Interfaces deklariert, jedoch von der Klasse nur einmal implementiert.

Ein Problem dieser Konstruktion ist, daß bei der Realisierung der Präsentationsformen auf Namenskonflikte mit den AWT-Klassen geachtet werden muß. Deklariert ein PF-Interface eine Methode, die von der AWT-Klasse (von der die PF-Klasse erbt) ebenfalls implementiert wird, kann es zu unerwünschten Effekten kommen. Benennt man beispielsweise seine Methode zum Hinzufügen eines Elementes in eine Auswahl mit „addItem" (im pfInterface `pf1fromNSelection`) und implementiert man diese Methode in der Klasse `pfList` mit einem Aufruf von „add" an der Oberklasse (`java.awt.List`), gerät die Applikation beim Aufruf von „addItem" an der Präsentationsform in eine Endlosschleife. Dies liegt darin begründet, daß die Oberklasse von `pfList (java.awt.List)`, ebenfalls eine Methode „addItem" definiert und ihre Methode „add" in einen Aufruf von „addItem" delegiert. Durch das dynamische Binden der Methoden wird dabei die „addItem"-Methode gerufen, die in der Klasse `pfList` implementiert ist und somit wieder „add" ruft. Diese Art der „Methoden-kollision" muß bei der Realisierung umgangen werden.

Durch die Tatsache, daß die PF-Klassen von den AWT-Klassen erben, ist man abhängig von der AWT geworden. Änderungen an der AWT müssen im Rahmenwerk nachvollzogen werden. Dies wäre jedoch im Falle einer Benutzt-Beziehung zwischen Präsentationsform-Klassen und Widgets nicht anders gewesen. Jedoch wird bei dieser Art der Konstruktion der Austausch der Widgets durch zum Beispiel neuartige Widgets anderer GUI-Toolkits („Java Foundation Classes") etwas erschwert, da alle Präsentationsform-Klassen verändert werden müssen. Allerdings hält sich auch dieser Aufwand in Grenzen und übertrifft den Aufwand im Falle der Benutztbeziehung nicht allzu sehr. Es überwiegt also der Vorteil der Möglichkeit, die Oberflächen graphisch zu gestalten.

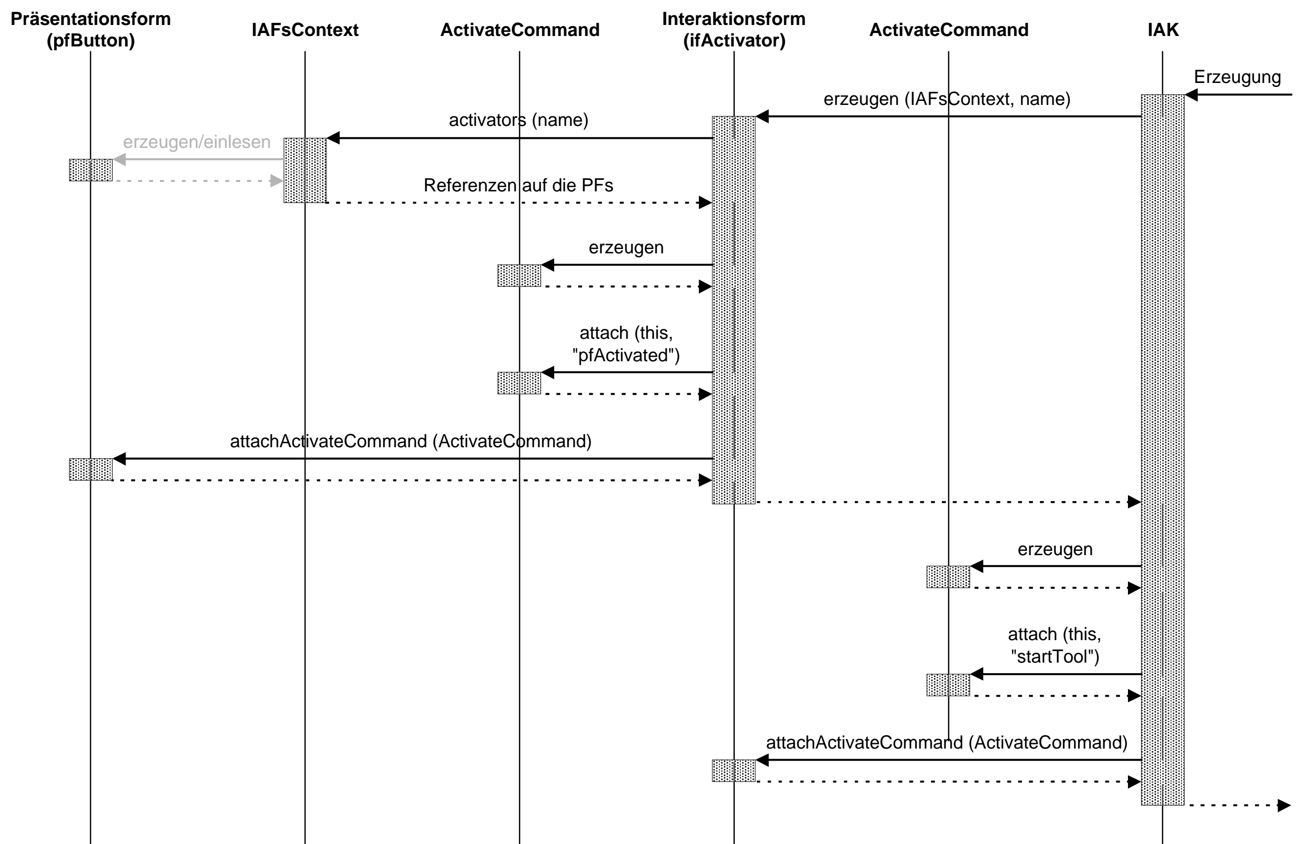

Abbildung 11: Interaktionsdiagramm für die Konstruktion der Interaktionsformen und deren Anbindung an die Präsentationsformen und die IAK.

9.2.4 Der Kontext der Interaktions- und Präsentationsformen

Die Interaktionsformen benötigen eine Möglichkeit, die ihnen zugeordneten Präsentationsformen zu referenzieren. Hierzu existiert im Rahmenwerk ein Interaktionsformen-Kontext (`IAFsContext`). Dieser Kontext kann von dem GUI-Manager erfragt werden und den Interaktionsformen übergeben werden. Die Interaktionsformen können von dem Kontext erfragen, welche Präsentationsformen zu ihnen gehören. Hierzu muß der Kontext die komplette Hierarchie von Präsentationsformen kennen. Somit kann er bei Bedarf durch diese Hierarchie iterieren, um die entsprechenden Präsentationsformen zu finden (siehe [Gör97]).

Da der Kontext im wesentlichen eine Hierarchie von Präsentationsformen beinhaltet, ist der Kontext in eine Klasse und ein Interface getrennt. Um auch in der Benennung eine deutliche

Trennung zwischen Interaktionsformen und Präsentationsformen beizubehalten, existiert ein Interface `IAFsContext`. Die Interaktionsformen bekommen ein Objekt vom Typ `IAFs-Context`. Hinter diesem Interface steht eine Klasse `PFsContext`, die das `IAFsContext`-Interface implementiert und die Hierarchie von Präsentationsformen beinhaltet. Dadurch ist es gelungen, die Interaktionsformen nur mit einem IAFsContext umgehen zu lassen, sie aber dennoch an die Präsentationsformen anbinden zu können.

Dieser Kontext bietet eine variable Möglichkeit, die Oberflächenkomponenten anzubinden. Sollte ein GUI-Builder verwendet werden, übernimmt ein GUI-Manager die Verarbeitung der Ressourcen und gibt dem Kontext die fertige Hierarchie von Oberflächenkomponenten. Sollte es für eine bestimmte Oberfläche nicht möglich sein, ein GUI-Builder zu verwenden (z.B. bei einer dynamischen Anzahl von Oberflächenkomponenten auf einem Desktop), kann in dem Kontext die Oberflächenhierarchie sofort erzeugt werden (im Konstruktor des Kontextes). Der Kontext ist ebenfalls in der Lage, dynamisch neue Oberflächenkomponenten hinzuzufügen, sollte eine Interaktionsform zum Beispiel nach einer noch nicht vorhandenen Präsentationsform fragen.

Durch diese Konstruktion ist der Entwickler nicht gezwungen, in einer Applikation alle Oberflächenkomponenten in Form von Ressourcen zu erstellen und statisch festzulegen. Zu den statisch durch einen GUI-Builder erzeugten Ressourcen können, für die Softwarewerkzeuge und die Interaktionsformen völlig transparent, dynamische und/oder nicht durch GUI-Builder erzeugbare Oberflächen integriert werden.

9.3 Übersicht über die Interaktionsformen und Präsentationsformen

Die folgenden Abbildungen zeigen den aktuellen Stand der Implementierung im JWAM-Rahmenwerk. Die Entwicklung ist keineswegs abgeschlossen. Geplante Änderungen und Erweiterungen sind gekennzeichnet.

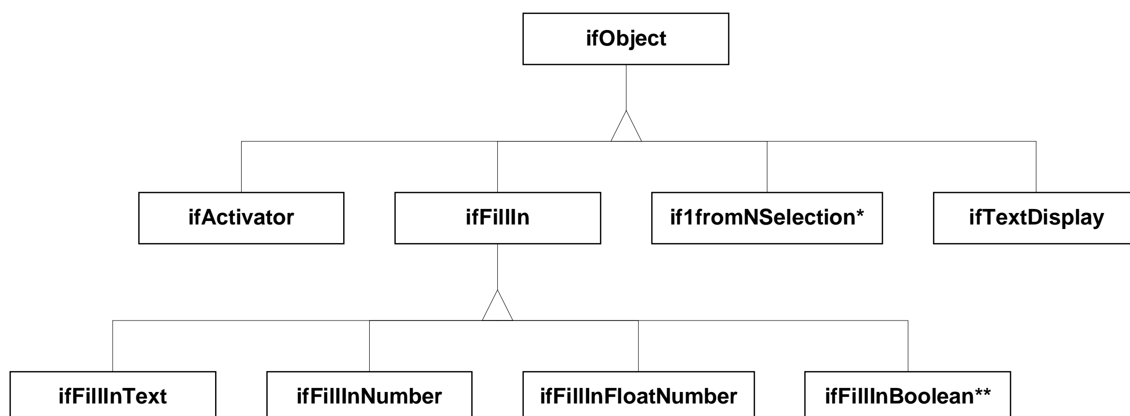

Abbildung 12: Aktueller Stand der Hierarchie der Interaktionsformen.
***) zu der 1fromNSelection werden noch Ergänzungen um Mehrfach-Selektion und statische und dynamische Selectionen hinzukommen.**
****) Der Name "ifFillInBoolean" ist nur eine vorübergehende Benennung.**

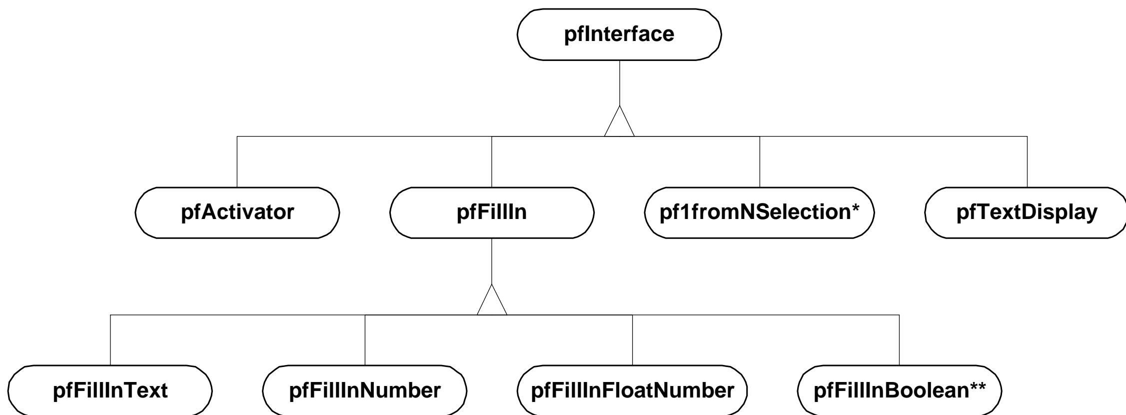

**Abbildung 13: Aktueller Stand der Hierarchie der Präsentationsformen.
*) zu der 1fromNSelection werden noch Ergänzungen um Mehrfach-Selektion und statische und dynamische Selectionen hinzukommen.
) Der Name "pfFillInBoolean" ist nur eine vorübergehende Benennung.

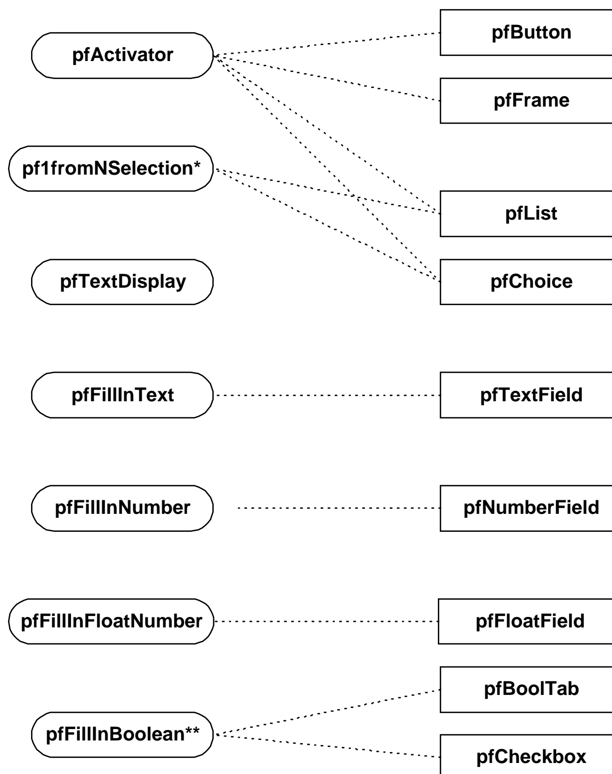

**Abbildung 14: Übersicht über die Präsentationsform-Klassen und welche PF-Interfaces sie implementieren.
(zu * und ** siehe oben)**

10 Zusammenfassung und Ausblick

In diesem Text haben wir das Konzept der Interaktionsformen und Präsentationsformen vorgestellt. Damit bieten wir ein neues Konzept für die Trennung von Handhabung und Darstellung an, das die Entwicklung von Softwarewerkzeugen für interaktive Anwendungssysteme unterstützt.

Mit Hilfe von Interaktionsformen können Softwarewerkzeuge elegant an grafische Benutzungsoberflächen angebunden werden. Interaktionsformen abstrahieren von der konkreten Präsentation auf dem Bildschirm und fokussieren auf den Umgang, den der Benutzer mit dem Softwaresystem hat. Seine Interaktionen mit dem System sind dafür der Ausgangspunkt.

Wir haben vor diesem Hintergrund eine Menge von Interaktionsformen herausgearbeitet und beschrieben.

Interaktionsformen werden durch Präsentationsformen auf dem Bildschirm dargestellt. Wir haben beschrieben, wie Interaktions- und Präsentationsformen mit Java und C++ konstruiert werden können.

Das Konzept der Interaktionsformen hat sich in den letzten 2 Jahren in mehreren universitären und industriellen Projekten bewährt. In diesen Kontexten sind jedoch auch neue Anforderungen entstanden, die noch in das Konzept bzw. die vorliegenden Implementationen integriert werden müssen.

In wenigen Fällen reichen die Manipulationsmöglichkeiten über die Interaktionsformen nicht aus. In diesen muß das Softwarewerkzeug die Möglichkeit haben, die Präsentation vollständig zu kontrollieren. Ein uneingeschränkter Zugriff auf die Präsentationsformen an den Interaktionsformen vorbei ist eine mögliche Lösung.

Die Interaktionsformen für Tabellen und Tree-Views haben nicht den gleichen Stabilitätsgrad erreicht, wie er bereits bei den "einfachen" Interaktionsformen vorherrscht. In diesem Bereich müssen wir weitere Erfahrungen mit dem Konzept der Interaktionsformen sammeln und genau betrachten, in welchen Zusammenhängen - also mit welchen Interaktionen - Tabellen und Tree-Views verwendet werden.

11 Anhang

11.1 Legende der Abbildungen

11.1.1 Klassen- und Objektdiagramme

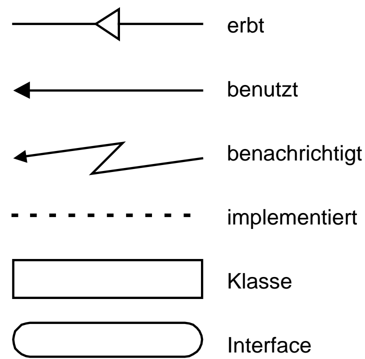

Abbildung 15 Klassen- und Objektdiagramme

11.1.2 Interaktionsdiagramme

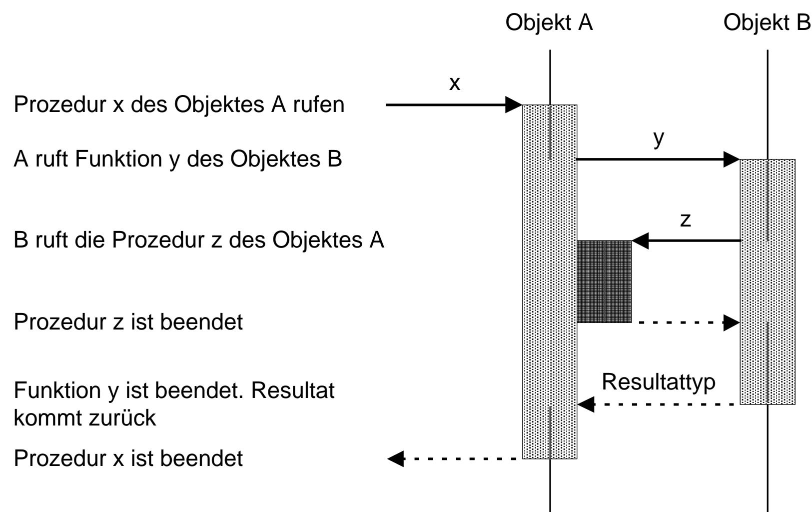

Abbildung 16 Interaktionsdiagramme

12 Literatur

[Bäu98] Dirk Bäumer: *Softwarearchitekturen für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme*. Dissertationsschrift am Fachbereich Informatik der Universität Hamburg, Januar 1998.

[BZ90] Reinhard Budde, Heinz Züllighoven. *Software-Werkzeuge in einer Programmierwerkstatt*. Oldenbourg. München. 1990.

[Fla97] David Flanagan. *Java in a Nutshell*. Sebastopol, Kalifornien: O'Reilly & Associates. 1997.

[GHJ+94] Erich Gamma, Reinhard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1994.

[Gör98] Thorsten Görtz. *Abstraktion der GUI-Komponente in einem Rahmenwerk. Diplomarbeit*. Arbeitsbereich Softwaretechnik. Fachbereich Informatik. Universität Hamburg. 1998.

[Gra95] Sven Grand: *Trennung von Interaktion und Funktion in der Werkzeug und Material-Methapher*. Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1995.

[KGZ94] Klaus Kilberth, Guido Gryczan, Henz Züllighoven. *Objektorientierte Anwendungsentwicklung. Konzepte. Strategien. Erfahrungen*. 2. verbesserte Auflage. Braunschweig/Wiesbaden. Vieweg. 1994.

[Lew95] Ted Lewis (Editor). *Object-Oriented Application Frameworks.* Greenwich, MA: Manning Publications Co., 1995.

[Lip97] Martin Lippert. *Konzeption und Realisierung eines GUI-Frameworks in Java nach der WAM-Metapher.* Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1997.

[Mey88] Betrand Meyer. *Objektorientierte Softwareentwicklung*. Carl Hanser Verlag und Prentice-Hall. München. 1988.

[Ros95] Larry Rosenstein. *MacApp: First Commercially Successful Framework.* in [Lew95]

[RW96] Stefan Roock, Henning Wolf. *Konzeption und Implementierung eines Reaktionsmusters für objektorientierte Softwaresysteme*. Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1996.

[Sma97] Julian Smart. *WxWindows - Documentation.* http://web.ukonline.co.uk/ julian.smart/wxwin. 1997

[StrFrö96] Wolfgang Strunk, Frank Fröse (1996). *Using Design Patterns to Restructure the User Interface Part of an Application Framework*. In: *Theory and Practice of Object Systems 2*, 1 (1996), pp. 53-60.

[Zül98] Heinz Züllighoven. *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz*. Heidlberg: d.punkt Verlag. 1998.