Studienarbeit

Konzeption und Realisierung eines GUI-Frameworks in Java nach der WAM-Metapher

Martin Lippert
Pagenfelder Straße 8
22111 Hamburg
4lippert@informatik.uni-hamburg.de

November 1997

Betreuung: Prof. Dr. Heinz Züllighoven

Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

„Die Begriffe der Menschen von den Dingen sind meistens nur ihre Urteile über die Dinge.“

Christian Friedrich Hebbel

„Die Neugier steht immer an erster Stelle eines Problems, das gelöst werden will.“

Galileo Galilei

Inhaltsverzeichnis

1 Einleitung

Im Rahmen der Entwicklung eines WAM-Framework in Java war die Umsetzung des GUI-Frameworks meine Aufgabe (zum Begriff „GUI“ siehe Abschnitt 1.1.1). Dabei standen zwei Aspekte im Mittelpunkt:

- Derzeit wird am Arbeitsbereich Softwaretechnik im Rahmen eines Workshops diskutiert, wie sich im WAM-Kontext eine Abstraktion von dem graphischen User-Interface erreichen läßt. Anlaß hierzu war die Feststellung, daß die bisher bekannten Konzepte Mängel aufweisen (siehe auch [Weiß97]), die die Erarbeitung neuer Konzepte nötig machen. Die Umsetzung dieser neuen Ansätze in das Java-GUI-Framework lag somit nahe, auch um Erfahrungen mit dieser neuen Konzeption zu gewinnen.
- Zum zweiten sollte die Verwendung von GUI-Buildern ermöglicht werden, ohne einen eigenen, speziellen GUI-Builder selbst zu entwickeln und ohne das Framework auf jeden vorhandenen Java-GUI-Builder abzustimmen. Hierzu war die Erarbeitung eines allgemeinen Resourcen-Konzeptes in Java nötig. Diese Konzeption wurde ebenfalls in das GUI-Framework integriert.

Die Werkzeug- und Material-Metapher setze ich hier als bekannt voraus und werde sie nicht erneut beschreiben. Wer damit nicht vertraut ist, sei an die entsprechende Literatur verwiesen, wie zum Beispiel [RZ95], [GKZ93] oder diverse Studien- und Diplomarbeiten des Arbeitsbereiches Softwaretechnik der Universität Hamburg.

1.1.1 Begriffe

Die Abkürzung *„GUI“* steht für „Graphical User Interface“. In dieser Arbeit verwende ich hauptsächlich diese englische Variante, auch in der Abkürzung „GUI“ oder nur „UI“. Die deutschen Varianten dieser Begriffe, wie „Benutzungsschnittstelle“ oder „Mensch-Maschine-Schnittstelle“, vermeide ich, da sie keine sehr gelungenen Übersetzungen darstellen. Aus technischer Sicht verwende ich ebenfalls für das User-Interface auch gelegentlich den Begriff der *„Oberfläche“*.
In den einzelnen Abschnitten verwendete Begriffe beschreibe ich am Anfang des entsprechenden Abschnittes und verweise dort auf Literatur zu den Begriffen.

1.1.2 Illustrationen

In dieser Arbeit finden sich verschiedene Abbildungen. Um nicht bei jeder Abbildung eine Legende anzuführen, findet sie sich im Anhang. Abweichungen von der Legende im Anhang werden in der Abbildungsbeschriftung erläutert.

1.1.3 Übersicht

Eine kurze Einführung in die Sprache Java ist an den Anfang dieser Studienarbeit gestellt. Da die Sprache Java bisher am Arbeitsbereich noch nicht Gegenstand zahlreicher Arbeiten war, stelle ich in diesem Abschnitt die Konzepte der Sprache kurz vor, ohne auf Details einzugehen. Die wesentlichen Merkmale der Objektorientierung in Java werden kurz vorgestellt und mit anderen objektorientierten Sprachen verglichen, sofern dies im jeweiligen Abschnitt sinnvoll ist.
Der zweite Teil widmet sich der Konzeption des GUI-Frameworks. Zunächst wird die bisheri-

ge Konzeption des GUI-Frameworks in der BibV30[1] vorgestellt. Hier wird kurz das Konzept der Interaktionstypen beschrieben, um einen Eindruck über die GUI-Abstraktion in der BibV30 in C++ zu bekommen. Anschließend wird das Konzept der Trennung in Interaktion und Präsentation beschrieben und wie man diese Trennung in ein Framework einbetten kann. Eine genauere Beschreibung findet sich in [Görtz97] und [Strunk97]. Der Abschluß dieses Abschnittes behandelt die Problematik, die durch die Verwendung von GUI-Buildern und die Einbindung der von diesen GUI-Buildern erzeugten Resourcen in das Framework entsteht: Da Resourcen von GUI-Buildern in der Regel nicht portabel sind, müssen bisher GUI-Frameworks an entsprechende Entwicklungstools angepaßt werden. Um von diesen GUI-Buildern unabhängig zu werden, muß von den Resourcen abstrahiert werden. Dazu werden einige generelle Lösungsansätze dargestellt.
Der Schwerpunkt dieser Arbeit liegt in der Realisierung eines Java-Frameworks zur Anbindung des graphischen User-Interface. Diese Realisierung in Java ist Gegenstand des vierten Abschnittes. Bei der Realisierung sollen die Konzepte der Trennung in Interaktion und Präsentation ebenso umgesetzt werden, wie die Abstraktion von den GUI-Buildern und den davon erzeugten Resourcen. Im Mittelpunkt stand der Anspruch, eine Lösung zu erarbeiten, die mit bereits existierenden GUI-Buildern handelsüblicher Java-Entwicklungsumgebungen zusammenarbeitet. Damit soll erreicht werden, eine praxisnahe Lösung zu finden, die nicht die Entwicklung eines eigenen GUI-Builders nötig macht.

1.1.4 Danksagung

Danken möchte ich an dieser Stelle Niels Fricke, Stefan Roock, Henning Wolf und Carola Lilienthal, die in regelmäßiger Diskussion Vorschläge und Kritik äußerten und mir dadurch eine große Hilfe waren. Ebenso gilt mein Dank den anderen Mitstudenten, darunter Frank Fröse und Thorsten Görtz, die oft meine Fragen und (Teil-)Ergebnisse erleiden mußten.

[1] Die BibV30 ist ein in C++ entwickeltes Framework, welches im Rahmen zahlreicher Arbeiten am Arbeitsbereich Softwaretechnik der Universität Hamburg entstand. Eine Beschreibung und Charakterisierung der BibV30 findet sich in [Weiß97].

2 Die Sprache Java

Die Sprache „Java“ wurde von der Firma Sun Microsystems im Jahre 1995 vorgestellt. Ursprünglich war das Ziel der Forschungsarbeit, die schon Jahre früher begonnen hatte, eine Sprache für eingebettete Systeme und das World-Wide-Web (WWW) zu entwickeln. Jedoch ist im Laufe der Entwicklung daraus eine Sprache entstanden, die universell einsetzbar ist. Mit Java lassen sich nicht nur kleine Web-„Applets“[2] bauen, sondern ganze Applikationen professionell entwickeln.
Java ist eine objektorientierte, stark getypte Sprache, die syntaktisch stark an C++ angelehnt ist. Allerdings gibt es zwischen Java und C++ einige entscheidene Unterschiede. Java kann als eine Sprache aufgefaßt werden, die Elemente der Sprachen C++, Smalltalk und Objektive-C beinhaltet als auch zusätzliche Eigenschaften aufweist.

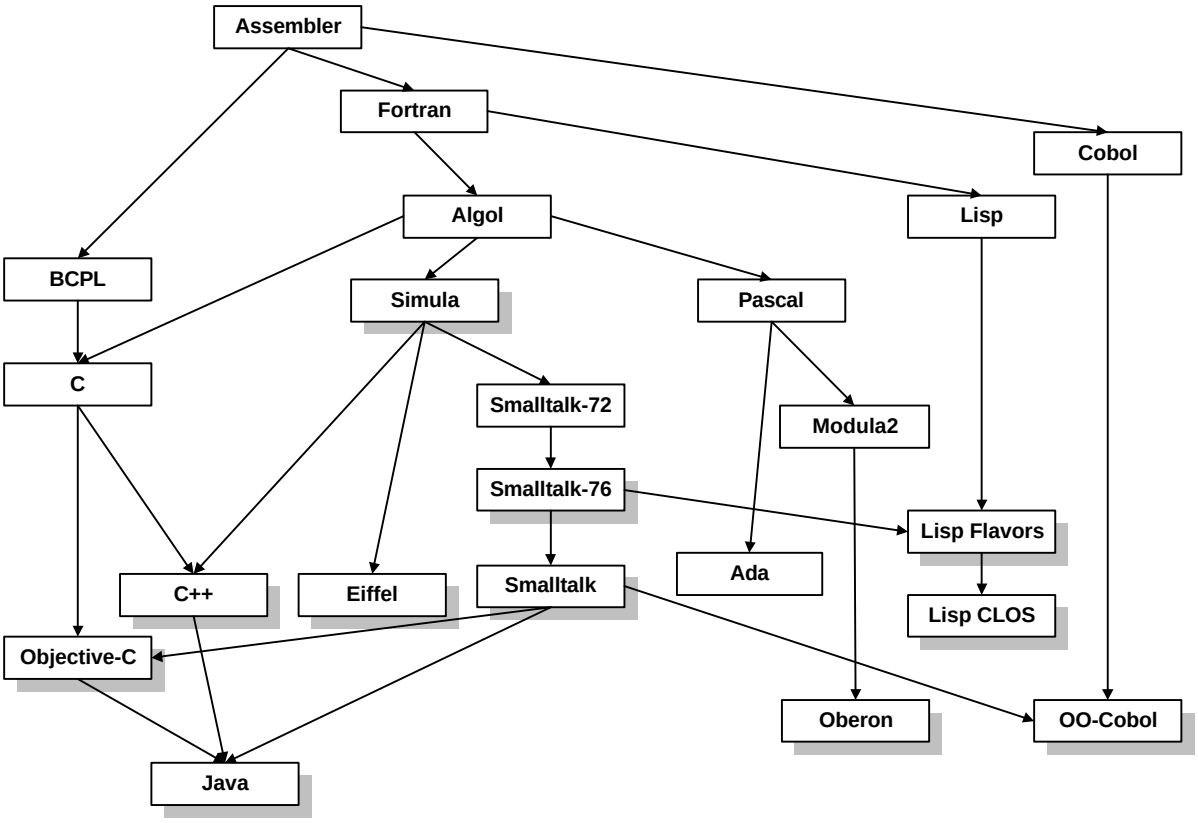


Abbildung 1: Der Weg zu objektorientierten Programmiersprachen[3]

Zu „Java“ gehört allerdings mehr, als nur die Sprachspezifikation. Java ist eine kompilierte *und* interpretierte Sprache. Der Compiler übersetzt den Source-Code in einen sogenannten „Byte-Code“, der plattformübergreifend ist und von jeder „Java Virtual Machine“ interpretiert werden kann. Dieses Konzept der Laufzeitumgebung durch die „Virtual Machine“ ist von Smalltalk her bekannt. Der „Byte-Code“ ist eine Mischung aus Source-Code und Maschinensprache. Dieser „Byte-Code“ ist plattfornübergreifend spezifiziert, so daß einmal kompilierte Source-Codes von jeder virtuellen Maschine ausgeführt werden können. Dabei spielt die ver-

[2] Ein „Applet“ ist ein, meist kleines, Java-Programm, welches in einem Web-Browser als Bestandteil der Web-Page abläuft.
[3] In Anlehnung an [LiWe96]

wendete Hardware und das Betriebssystem keine Rolle.
Das „Zusammenlinken“ des Byte-Codes zu einer Applikation ist in Java nicht nötig. Wird zur Laufzeit eine bestimmte Klasse benutzt, bemerkt dies die Virtuelle Maschine und lädt dynamisch den Byte-Code der entsprechenden Klasse.
Ich werde hier nur die sprachlichen Konzepte kurz vorstellen und nicht die komplette Java-Umgebung beschreiben. Wer mehr über Java und die Java-Umgebung wissen möchte, sei an die entsprechende Literatur verwiesen.[4]

2.1.1 Klassen und Interfaces

Java unterscheidet zwischen Klassen und Interfaces. Interfaces beschreiben nur eine Schnittstelle, die von einer Klasse implementiert werden muß. Deshalb können von Interfaces keine Objekte erzeugt werden. Das Konzept der Interfaces stammt aus Objective-C (dort Protocols genannt). In Klassen befindet sich sowohl die Deklaration der Schnittstelle, als auch die Implementation der Methoden und die Deklaration von Variablen (Attributen).
Implementiert eine Klasse ein Interface, so müssen sämtliche in dem Interface deklarierten Methoden von der Klasse implementiert werden, damit von der Klasse ein Objekt erzeugt werden kann. Ist dies nicht der Fall, muß die Klasse als „`abstract`“ deklariert werden, was wiederum bedeutet, daß auch von dieser Klasse kein Objekt erzeugt werden kann. Polymorph zuweisbar sind Bezeichner sowohl auf der Ebene der Klassenhierarchie als auch auf der Ebene der Interfacehierarchie.
Auf der Ebene des Typsystems definieren in Java sowohl Klassen als auch Interfaces Typen.

2.1.2 Vererbung

Vererbung wird einerseits im Sinne der Implementationsvererbung verwendet, andererseits um Begriffshierarchien abzubilden.
Auf der Klassenebene ist in Java nur die Einfachvererbung erlaubt. Unter Interfaces ist dagegen Mehrfachvererbung gestattet. Hinzu kommt, daß eine Klasse mehrere Interfaces implementieren kann (siehe Abbildung 2). Co- und Contra-Varianz für Operationsparameter und Rückgabetypen wird in Java nicht unterstützt.

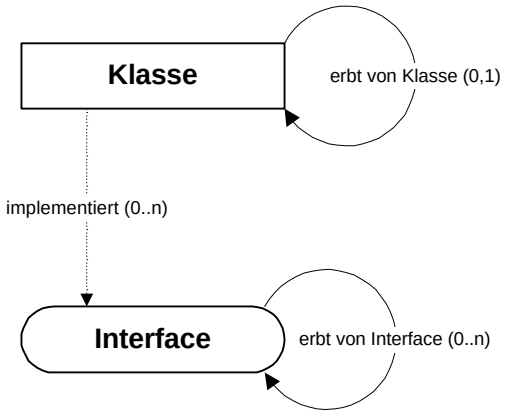


Abbildung 2: Vererbungs- und Implementationsbeziehungen zwischen Klassen und Interfaces

Die Einschränkung auf Einfachvererbung unter Klassen stellt eine deutliche Einschränkung bei der Konstruktion dar. Sinnvoll angewandt lassen sich mit Mehrfachvererbung beispielsweise

[4] [Flanagan96], [GJS97]

Aspektklassen mit Einschubmethoden sehr einfach implementieren. Oftmals ist man so mit Einfachvererbung dazu gezwungen, eine Mehrfachvererbung „nachzubauen“ oder künstlich zu umgehen, meist durch kompliziertere Konstruktionen.

2.1.3 Objektreferenzen und Garbage Collection

In Java gibt es keine Pointer, wie man sie aus C++ kennt. Objekte sind ausschließlich über ihre Objektreferenz zugänglich. Diese Objektreferenzen sind in Java in der Weise implementiert, daß sie entweder auf ein Objekt verweisen oder den Wert `null` haben. Hinzu kommt, daß in Java ein Garbage Collector integriert ist, der automatisch nicht mehr referenzierte Objekte aus dem Speicher entfernt. Damit genießt man den Vorteil, sich nicht um typische Pointer-Probleme kümmern zu müssen, wie „Lost Object“ oder „Dangling Pointer“:

- Verweisen zwei Referenzen auf dasselbe Objekt, kann es zu Problemen kommen, wenn das Objekt explizit gelöscht wird. Eine der zwei Referenzen referenziert dann ein nicht mehr existierendes Objekt (die Referenz wird dann „Dangling Pointer“ ganannt). Da unter Verwendung eines Garbage Collectors keine Objekte explizit gelöscht werden sondern vom Gargabe Collector erst gelöscht werden, wenn sie nicht mehr referenziert werden, kann es zu keinem „Dangling Pointer“ kommen.
- Ein „Lost Object“ ist ein Objekt, welches nicht explizit gelöscht wurde, aber gleichzeitig nicht mehr referenziert wird. Dies kann durch Setzen der Referenz auf ein anderes Objekt oder auf `null` leicht geschehen. Ist die Referenz auf dieses Objekt einmal verlorengegangen, kann das Objekt nicht mehr referenziert werden, somit auch nicht mehr gelöscht werden. Auch dieses Problem löst der Garbage Collector, indem er nicht mehr refenzierte Objekte selbständig aus dem Speicher entfernt.

Weitere Erläuterungen zu „Dangling Pointer“ und „Lost Object“ können [Sebesta97] entnommen werden.

2.1.4 Wert- und Referenzsemantik

Objekte von Klassen sind in Java nur über die Objektreferenz zugänglich. Sie verhalten sich demzufolge nach Referenzsemantik. Diese Referenzsemantik kann nicht durch andere Konstrukte umgangen werden. Die Parameterübergabe an Methoden funktioniert daher ebenfalls nach Referenzsemantik. Ein „Call-by-Value“ oder „Call-by-Name“ gibt es für Objekte nicht.
Einzige Ausnahme sind die Basistypen, wie `int`, `char`, `boolean`, etc. Diese Typen verhalten sich nicht nach Referenzsemantik, sondern nach Wertsemantik und werden bei der Parameterübergabe an Methoden nach dem Schema „Call-by-Value“ übergeben.
Um Werte von Basistypen wie Objekte mit Referenzsemantik benutzen zu können, gibt es in der Klassenbibliothek von Java Wrapper-Klassen für die Basistypen. Von den Wrapper-Klassen können Objekte erzeugt werden. Damit handelt es sich um Exemplare einer Klasse, die sich, wie oben beschrieben, nach Referenzsemantik verhalten. Die Zustand dieser Exemplare lassen sich nach der Erzeugung nicht verändern. Der einmal bei der Konstruktion dem Konstruktor übergebene Wert des Basistyps kann nicht durch einen anderen Wert ersetzt werden (zu Wert- und Referenzsemantik siehe [Meyer90]).

2.1.5 Generizität und Templates

Die Sprache Java bietet derzeit keine Möglichkeit, Generizität auszudrücken. Einen Template-Mechanismus wie in C++ sucht man in Java vergeblich. Abhilfe schafft hier nur sehr bedingt die Tatsache, daß alle Klassen eine gemeinsame Oberklasse haben (`Object`). Diese mono-

lithische Klassenhierarchie stammt von Smalltalk. Allerdings läßt sich in Java sehr einfach zur Laufzeit prüfen, ob ein Objekt von einem bestimmten Typ ist (siehe Meta-Objekt-Informationen weiter unten).
Durch die monolithische Klassenhierarchie können in Java Container auf Basis der Klasse `Object` implementiert werden. Das bedeutet, daß die Schnittstelle des Containers auf `Object` beruht und bei der Verwendung des Containers ein Downcast erfolgen muß, der in Java sicher ist. Aus fachlicher Sicht stellt dieser Downcast jedoch keinen Unsicherheitsfaktor dar, da fachliche Behälter an ihrer Schnittstelle sicherstellen können, daß in sie nur Objekte eines bestimmten Typs eingefügt werden können. So kann der fachliche Behälter sicher sein, daß bei der Herausgabe eines Objektes aus dem Behälter der Downcast erfolgreich ist. Eine ausführlichere Diskussion zur Implementation von Behältern in Java findet sich in [Bohlmann98].

2.1.6 Das Vertragsmodell

Auch wenn man das Vertragsmodell nur in wenigen Sprachen findet (beispielsweise Eiffel, siehe [Meyer90]), so wäre es doch wünschenswert, es in einer modernen objektorientierten Sprache vorzufinden. In Java ist dies nicht der Fall. Java bietet keine direkte Unterstützung des Vertragsmodells. Ähnlich wie in C++ muß es auch hier zusätzlich implementiert werden (zur Java-Implementierung des Vertragsmodells siehe [Fricke97]).

2.1.7 Exception-Handling

Bei der Entwicklung von Applikationen kann nicht ausgeschlossen werden, daß in den Applikationen Fehler auftreten. Aus diesem Grund ist es notwendig, in Applikationen Fehlerbehandlungen vorzunehmen. Während des Einsatzes der Applikation auftretende Fehler sollten nach Möglichkeit innerhalb der Applikation aufgefangen werden, so daß der Benutzer der Applikation davon nichts bemerkt. Sollte dies nicht möglich sein, sollte dem Benutzer zumindest eine adäquate Meldung angezeigt werden.
Um dies zu ermöglichen, ist in Java ein Exception-Handling implementiert, welches dem von C++ recht ähnlich ist. Java unterscheidet Ausnahmen prinzipiell in drei Kategorien: Exceptions, Runtime-Exceptions und Errors. Diese drei Kategorien unterscheiden sich in der Art und Weise, wie und ob auf sie reagiert werden muß.
Um diese Kategorien kurz zu erläutern, sei folgendes Beispiel kurz beschrieben: Die Methode A ruft eine Methode B. Die Methode A wird hier mit „rufende Methode“ bezeichnet, die Methode B mit „gerufene Methode“. Die Methode B, die „gerufene“, löst eine Ausnahme aus.
Wirft die gerufene Methode (B) eine Exception aus, so *muß* die rufende Methode (A) entweder die Exception auffangen (in einem `try-catch`-Block) oder sie explizit weiterreichen. Das bedeutet nichts anderes, als daß die rufende Methode (A) dann an ihrer Schnittstelle ebenfalls deklarieren muß, daß sie diese Exception auswerfen kann. Beispiel: `IOException`, `AWTException`, `NoSuchMethodException`.
Im Unterschied dazu verlangen Runtime-Exceptions keine Behandlung. Runtime-Exceptions *können* aufgefangen werden, müssen es aber nicht. Wird die Runtime-Exception nicht aufgefangen, wird sie automatisch an die rufende Methode weitergereicht. Beispiel: `NullPointerException`, `IndexOutOfBoundsException`, `ClassCastException`.
Ein Error hingegen *sollte nicht* aufgefangen werden, da es sich bei Errors meist um extreme Systemfehler handelt. Beispiel: `OutOfMemoryError`, `InternalError`, `LinkageError`.
Werden die letztgenannten Ausnahmen (Runtime-Exceptions und Errors) in der kompletten Applikation nicht aufgefangen, gibt die Java Virtual Machine eine Fehlermeldung auf der Con-

sole aus. Daraufhin entscheidet die Java Virtual Machine über eine Fortsetzung des Programmablaufes. Bei Runtime-Exceptions wird sie den Programmlauf insofern fortsetzen, als daß der Kontrollfluß bei der Hauptereignisschleife (innerhalb der Virtual Machine) wieder aufgesetzt wird. Tritt ein schwerwiegender Error auf, werden auf der Console zusätzliche Informationen ausgegeben und die virtuelle Maschine beendet.

2.1.8 Meta-Objekt-Informationen

Java stellt Meta-Objekt-Informationen zur Laufzeit zur Verfügung. Neben den Informationen, ob eine Klasse von einem bestimmten Typ ist, lassen sich ab dem JDK1.1[5] wesentlich mehr Informationen von einer Klasse erfragen. Dazu gibt es eine Klasse `Class`, die nicht wie in Smalltalk eine wirkliche Meta-Ebene bereitstellt, sondern über die nur bestimmte Informationen über Klassen erfragt werden können.
An der Klasse `Class` extstieren Methoden, um die Methodensignaturen, die Attribute und die Konstruktoren einer Klasse abzufragen. Zu jeder dieser Meta-Information gibt es eine entsprechende Klasse (`Method`, `Field`, `Constructor`), die durch Methoden einen bestimmten Umgang mit diesen Informationen ermöglichen. So existieren an der Klasse `Method` eine Methode, um die durch das Objekt von Typ `Method` bezeichnete Methode zu rufen. Die Klasse `Field` bietet Methoden an, um das Attribut zu sondieren und verändern[6]. Diese Möglichkeiten der Veränderung beziehen sich allerdings nur auf Objekte einer Klasse, nicht auf die Klasse selbst. Eine Klasse kann mittels der Meta-Objekt-Informationen nicht verändert werden, wie dies in Smalltalk der Fall ist. Weitere Informationen zu Meta-Object-Informationen in Java finden sich in [JCR96].

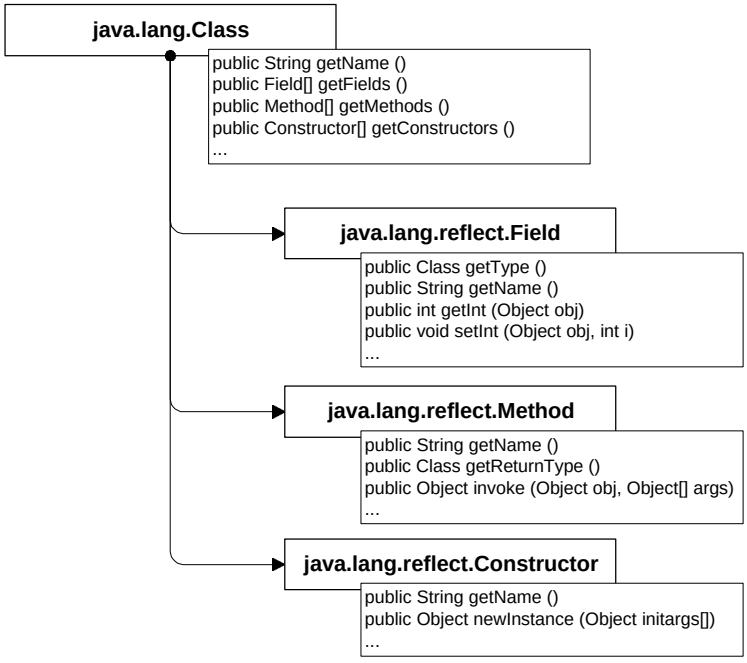


Abbildung 3: Überblick über die Meta-Objekt-Informations-Klassen mit einem Auszug der Methoden

[5] JDK steht für Java-Development-Kit, welches einfache Tools beinhaltet, wie einen Compiler, eine Java-Virtual-Machine, etc. Weiterhin ist die Versionsnummer Hinweis auf die Version der Sprache sowie den Umfang der Standard-Bibliotheken.
[6] Das Verändern eines Attributes ist nur möglich, wenn es als `public` deklariert ist.

2.1.9 Komponentenmodell

Komponententechnologie wurde zu dem Zweck entwickelt, Software aus vorgefertigten, kleinen, wiederverwendbaren Teilbausteinen zusammenzusetzen, die visuell verbunden werden können und deren Beziehungen zueinander dynamisch veränderbar sind. Diese dynamische Veränderbarkeit soll vor allem in Builder-Tools möglich sein. Solche Builder-Tools dienen dazu, Komponenten zusammenzufügen und über Einstellungen an die konkrete Anwendung anzupassen und zu einer konkreten Anwendung zusammenzufügen.
Seit dem JDK1.1 existiert für Java ein Komponentenmodell, das sogenannte JavaBeans-Modell. Zur Spezifikation dieses Komponentenmodells werden bestimmte Konventionen definiert. Diese Konventionen werden in der Spezifikation zwar „Design-Patterns“ genannt, jedoch handelt es sich dabei nicht um wirkliche Entwurfsmuster, sondern vielmehr um Konventionen zur Benennung von Methoden und Klassen.
So stellt jede Klasse, die nach diesen Konventionen implementiert wurde, eine Komponente dar, die in auf JavaBeans abgestimmten Entwicklungstools als solche verwendet werden kann und so visuell mit anderen Komponenten verbunden werden kann. Ein Hauptanwendungsgebiet stellt zur Zeit noch die Entwicklung graphischer Oberflächen dar, die aus Komponenten zusammengestellt werden. Weiterführende Informationen finden sich in [JBeans96].

2.1.10 Multi-Threading

Multi-Threading bezeichnet die Eigenschaft, innerhalb einer Applikation mehrere Prozesse (Threads) zu erzeugen und parallel ablaufen zu lassen. Dabei haben die verschiedenen Threads prinzipiell Zugriff auf alle Objekte der Applikation, sofern sie eine Referenz auf diese Objekte besitzen. Es existiert dann für jeden Thread ein eigener Kontrollfluß.
Die Fähigkeit zum Multithreading ist in Java auf Sprachebene eingebaut. Das bedeutet, daß jede Klasse, die in Java implementiert wird, automatisch multithreading-fähig ist. Dazu ist kein zusätzlicher Aufwand nötig. Das bedeutet allerdings nicht, daß sich diese Klasse auch sicher in einem Multithreading-Prozess verhält. Um dies sicherzustellen, ist zusätzlicher Aufwand erforderlich. Zur expliziten Behandlung der Threads sind in den Standard-Bibliotheken Klassen vorhanden, mit denen man Threads erzeugen, starten, anhalten, warten lassen kann, etc. Ist es in der Klassendefinition nicht explizit anders angegeben, kann auf ein Objekt von mehreren Threads gleichzeitig zugegriffen werden. Dabei ist nicht vordefiniert, welcher Thread wann auf das Objekt zugreift. So kann es vorkommen, daß zwei Threads ein Objekt mit derselben Methode rufen und diese Methode gleichzeitig[7] von zwei Threads ausgeführt wird. Um eine Synchronisation dieser Threads herzustellen, zum Beispiel um einen gegenseitigen Ausschluß zu implementieren, bietet Java ein spezielles Sschlüsselwort an (`synchronized`) sowie spezielle Methoden in der Klasse `Object` (`notify`). Einige detaillierte Ausführungen zum Thema „Java und Multithreading“ finden sich in [Lea97].

2.1.11 Zusammenfassung

Java ist eine Sprache, die die gängigen objektorientierten Technologien umsetzt. Java vereint sowohl Eigenschaften von C++ (z. B. starke Typisierung), Eigenschaften von Smalltalk (z. B. Einfachvererbung, monolithischer Klassenbaum, Garbage Collection) als auch zusätzliche Eigenschaften, wie Multithreading. Obwohl Java syntaktisch an C++ angelehnt ist, finden sich viele Eigenschaften von C++ in Java nicht wieder, was zum Teil positiv zu bewerten ist (keine Pointer), zum Teil allerdings auch negativ (keine Mehrfachvererbung). Andere Konzepte feh-

[7] Natürlich auf einem Einprozessorsystem nicht wirklich gleichzeitig, aber für den Entwickler erscheint es so, als ob mehrere Prozessoren vorhanden wären.

len ganz (Generizität).
Java ist nicht nur eine Sprache, die durch eine Grammatik festgelegt werden kann, sondern beinhaltet eine komplette Laufzeit-Umgebung, ähnlich wie Smalltalk. Zusätzlich bietet Java die Möglichkeit an, Klassen zu strukturieren. Dazu lassen sich in Java Klassen in „Packages“ einteilen. Diese Packages können hierarchisch gegliedert werden. An der Klasse wird gekennzeichnet, in welchem Package sie sich befindet. Dazu wird das Schlüsselwort `package` verwendet. Als Beispiel lassen sich die Klassen der Standard-Bibliotheken nehmen, die in Packages organisiert sind. In dem Package `java.util` befindet sich beispielsweise die Klasse `Vector`. Im Source-Code der Klasse `Vector` ist mit dem Schlüsselwort `package` diese Zugehörigkeit gekennzeichnet. Mit diesem Mechanismus läßt sich auch eine große Anzahl von Klassen strukturieren. Um eine Klasse zu verwenden, muß das entsprechende Package importiert werden. Dadurch läßt sich an den Import-Statements einer Klasse sehr leicht erkennen, welche anderen Klassen von dieser Klasse verwendet werden.
Obwohl es derzeit noch einige Performance-Probleme durch die Java Virtual Machine gibt, eignet sich Java keinesfalls nur für die Entwicklung bunter, aufgepeppter Web-Sites, sondern auch für die komplette und professionelle Applikationsentwicklung.
Aktuelle Berichte zeigen jedoch, daß Java in seiner Einsetzbarkeit umstritten ist, sowohl auf wissenschaftlicher (siehe [ACM97]), als auch auf wirtschaftlicher Seite. So hat beispielsweise die Firma „Corel“ die Entwicklung eines Projektes eingestellt, in dem ein Office-Produkt komplett in Java erstellt werden sollte. Über eine Beta-Version ist dieses Produkt nicht hinausgekommen.
Gründe für diese Schwierigkeiten mögen in der zur Zeit noch zu rasanten Entwicklung der Sprache sowie dem Fehlen professioneller Entwicklungstools liegen, die der Sprachentwicklung schon fast grundsätzlich hinterherhinken. Die mangelnde Performance mag ein zusätzlicher Grund sein.

3 Konzeption eines GUI-Frameworks

In interaktiven Systemen spielt das User-Interface eine wichtige Rolle. Die Schnittstelle zwischen dem Menschen und der Maschine wird in diesen Systemen meist durch eine graphische Darstellung am Bildschirm (sowie verschiedensten Eingabegeräten) realisiert. Zur Entwicklung dieser graphischen Oberfläche bieten moderne Systeme dem Entwickler sogenannte Toolkits zur Verwendung an. Mit Hilfe dieser Toolkits lassen sich die graphischen Elemente (Widgets), aus denen die graphische Oberfläche zusammengesetzt ist, durch den Entwickler handhaben. Diese Toolkits sind in der Regel systemspezifisch. Jedes System hat sein eigenes „Look & Feel“ und sein eigenes Design, wenn es um die Konstruktion dieser Toolkits geht. Um jedoch bei der Entwicklung von Anwendungen nicht von dem konkreten System (bzw. dem konkreten Toolkit) abhängig zu sein und Anwendungen leicht auf andere Systeme portieren zu können, hat es sich als sinnvoll erwiesen, diesen direkten Zugang zum System zu kapseln (siehe auch [Vlissides95]).

3.1 GUI-Abstraktion mittels Interaktionstypen

In der WAM-Konstruktion ist bereits die Trennung des Werkzeuges in Funktionskomponente und Interaktionskomponente ein Schritt in diese Richtung (vgl. [GKZ93]). Es ist jedoch wünschenswert, mit der Interaktionskomponente ebenfalls vom System unabhängig zu sein. Aus diesem Grund sind die sogenannten Interaktionstypen (auch IAT genannt) entwickelt worden. Diese Interaktionstypen kapseln die Widgets des Systems und werden von der IAK anstelle der Widgets des Systems benutzt.
Die IATs werden von der IAK erzeugt. Die IAK kann sich mittels Commands bei den entsprechenden IATs für bestimmte Aktionen anmelden. Dadurch ist auch der Reaktionsmechanismus für die IAK in einer WAM-konformen Art und Weise implementiert. Die IAK muß nicht mehr mit den entsprechenden Callback-Mechanismen des Systems umgehen. Dies erledigen die IATs.

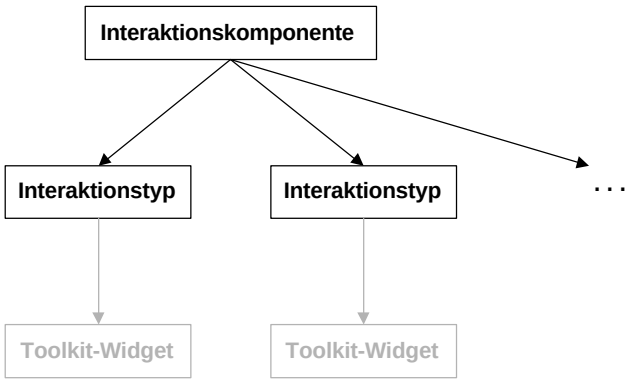


Abbildung 4: Zwischen der Interaktionskomponente und den Widgets des Toolkits liegen die Interaktionstypen.

Ziel dieser Entwicklung war es, die Schnittstelle zu dem Fenstersystem zu kapseln und die Anzahl der Klassen, die von dem konkret verwendeten Fenstersystem abhängen, möglichst klein zu halten. Dadurch hat man erreicht, daß bei einem Wechsel von einem Fenstersystem zu

einem anderen Fenstersystem nur eine kleine Anzahl von Klassen verändert werden muß. Das komplette Werkzeug kann unangetastet bleiben.

Im Laufe der letzten Jahre hat der Arbeitsbereich Erfahrungen mit dieser Art der Konstruktion gesammelt, unter anderem in diversen Projekten. Hierzu existiert am Arbeitsbereich innerhalb der BibV30 eine IATMotif-Bibliothek, sowie eine IATTcl-Bibliothek. Dabei stellte sich heraus, daß die Konstruktion dieser Bibliotheken sehr stark an dem jeweils zugrundeliegenden Toolkit orientiert ist. Das bedeutet, daß eine echte Abstraktion von dem Toolkit nicht stattgefunden hat. Die einzelnen Widgets sind zwar gekapselt worden, jedoch ist die Auswahl der IATs und deren Benutzung so stark an dem Toolkit orientiert, daß es nur sehr schwierig möglich ist, eine gleich zu benutzende IAT-Bibliothek für ein komplett anderes Toolkit zu implementieren.
An dieser Kritik zeigt sich, daß die ursprüngliche Abstraktion mittels IATs zuwenig an dem tatsächlichen Umgang orientiert war, sondern von der technischen Seite gesteuert wurde. Aus diesem Grund lag es nahe, eine den Umgang betonende Abstraktion zu konstruieren.

3.2 Die Trennung in Interaktion und Präsentation

Da die Abstraktion von dem Toolkit mittels Interaktionstypen zu wenig am Umgang orientiert war und sich somit als unpraktikabel herausgestellt hatte, entstand die Frage, wie eine am Umgang orientierte Abstraktion des graphischen User-Interfaces erreicht werden kann. Diese Frage zu lösen war die Aufgabe eines Workshops am Arbeitsbereich Softwaretechnik. Dieser hat eine Trennung in Interaktion und Präsentation vorgeschlagen und dazu detaillierte Konzepte erarbeitet.
Die Trennung in Interaktion und Präsentation beruht darauf, daß ein grundsätzlicher Unterschied besteht zwischen dem, *was* der Benutzer an der Oberfläche *tun* (wie er agieren) kann und dem, *wie* es ihm *dargeboten* wird.
Als Beispiel kann hier eine Auswahl herangezogen werden. Der Benutzer soll in die Lage versetzt werden, aus einer Menge von Möglichkeiten eine Auswahl zu treffen. Dies könnte beispielsweise bei der Bearbeitung eines Termins aus einer Menge von Terminen nötig sein. Der Benutzer muß den zu bearbeitenden Termin auswählen. Diese Auswahl ist die eigentliche Interaktion des Benutzers mit dem Werkzeug. Die Darstellung an der Benutzeroberfläche kann in Form einer Listbox oder einer Combobox[8] geschehen. Die Wahl dieser Präsentation kann unterschiedlichste Gründe haben und sollte keinen Einfluß auf das Werkzeug ausüben. Hier können noch weitere Beispiele gefunden werden.
Die Interaktion liegt auf der Ebene wie „Auswahl“, „Aktivierung“, „Ausfüllen“, etc. Diese Formen der Interaktion werden „Interaktionsformen“ genannt.
Die Präsentationsform bestimmt nun das Aussehen, die Darstellung einer Interaktion an der graphischen Oberfläche. Die Interaktion „Auswahl“ kann mittels verschiedener Arten an der graphischen Oberfläche „präsentiert“ werden. Diese Präsentation kann von unterschiedlichen Faktoren abhängig sein. Zum einen kann das „Look & Feel“ des Systems unterstützt werden. Zum anderen kann beispielsweise eine „Aktivierung“ an der Oberfläche als ein Knopf zum Drücken dargestellt werden, aber auch als ein Menüeintrag. Die Wahl dieser Form der Präsentation, der „Präsentationsform“, kann von verschiedensten Aspekten abhängen. Ohne die Art der Interaktion verändern zu wollen, können immer wieder Änderungswünsche auftreten, die ausschließlich die Präsentation betreffen.
Durch die Trennung von Interaktion und Präsentation sind die einzelnen Komponenten der

[8] auch bekannt als „Choice“ oder „Cascade-Button“.

Oberfläche eines Werkzeuges somit flexibel änderbar und austauschbar.

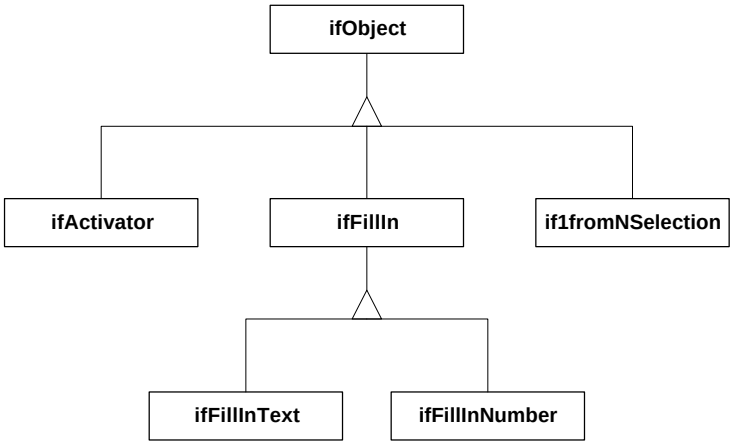


Abbildung 5: Die Klassenhierarchie der Interaktionsformen[9] (Ausschnitt)

Weiterführende und detailliertere, über diese kurze Einführung hinausgehende, Informationen zur Trennung von Interaktion und Präsentation finden sich in [Görtz97] und [Strunk97].

3.3 Die Umsetzung in ein Framework

Der Begriff des *Frameworks* hat in der objektorientierten Softwareentwicklung eine große Bedeutung gewonnen. Hier sollen nur einige Stellen zitiert werden, um den Begriff einzuführen. In [Weiß97] wird ausgeführt:
„Frameworks bestehen aus abstrakten und implementierten Klassen. Durch Vererbung werden Klassenhierarchien gebildet. Das wesentliche Merkmal objektorientierter Frameworks gegenüber aus prozeduralen Sprachen bekannten Funktions- und Bausteinbibliotheken ist ein bereits definiertes Modell der dynamischen Beziehungen verschiedener Klassen zur Laufzeit."
[Taligent95] führt zum Begriff des Frameworks aus:
„A framework is an extensible library of cooperating classes that make up a reusable design solution for a given problem domain."
Weitergehende Diskussionen zu Frameworks finden sich in [Weiß97], [Taligent95] oder [Lewis95].

3.3.1 Ausgangspunkt Toolkit

Der Ausgangspunkt für jedes Framework zur Behandlung des graphischen User-Interface ist das Toolkit, welches die elementaren Bestandteile eines User-Interface dem Entwickler zur Verfügung stellt. Dazu bieten Toolkits in der Regel sogenannte Widgets an. Die Widgets, die einzelnen Elemente eines User-Interface (z.B. ein Button, eine Listbox) kann der Entwickler benutzen. Diese Widgets sind von Toolkit zu Toolkit sehr unterschiedlich. Einige Toolkits basieren noch auf nicht objektorientierten Bibliotheken (z.B. die Macintosh Toolbox, siehe [Apple95]), andere stellen eine Hierarchie von Widget-Klassen zur Verfügung (z.B. StarView, siehe [Star92]).
Ein Ziel der Entwicklung von GUI-Frameworks im WAM-Kontext ist die Unabhängigkeit von

[9] Die Bezeichnung der Klassen mit dem Präfix „if" sowie die Benennung der Oberklasse als „ifObject" folgen den Styleguides zu Java (siehe [Style97]).

diesen Toolkits. Mit einem GUI-Framework soll eine objektorientierte Schnittstelle zur Verfügung gestellt werden, die an der WAM-Konstruktion orientiert ist. Dazu ist es nötig, die Klassen oder Bibliotheken der Toolkits in Klassen zu kapseln. Darüber hinaus kann mit Hilfe dieser Kapseln das verwendete Toolkit derart abstrahiert werden, daß eine Adaption auf ein anderes Toolkit zwar Aufwand bedeutet, jedoch dieser Aufwand auf die wenigen Klassen des Frameworks begrenzt ist, welche die Widgets kapseln. Die Applikation, die auf dem Framework aufbaut, braucht nicht angepaßt zu werden. Diese Abstraktion ist mit IATs bereits realisiert worden. Auch wenn sich die Konstruktion mittels IATs als nicht ausreichend am Umgang orientiert herausgestellt hat, muß ein neues Framework, welches die Trennung in Interaktion und Präsentation realisiert, ebenfalls diese Abstraktion von dem Toolkit leisten.

3.3.2 Die Struktur des GUI-Frameworks

Wenn man sich nun die Struktur ansieht, die aus der Trennung von Präsentation und Interaktion hervorgeht, entsteht folgendes, vierstufiges Design:

- Interaktionskomponente
- Interaktionsform
- Präsentationsform
- Toolkit (Widget)

Das GUI-Framework besteht aus den mittleren zwei Stufen. Das Toolkit steht außerhalb des Frameworks und kann nicht beeinflußt werden. Die Interaktionskomponente des Werkzeuges stellt den Benutzer des Frameworks dar (siehe Abbildung 6).
Die Interaktionskomponente benutzt nur noch Interaktionsformen. Die Interaktionsformen benutzen die Präsentationsformen. Diese Benutzt-Beziehung kann eine 1-zu-n-Beziehung sein, da zu einer Interaktionsform mehrere Präsentationsformen vorhanden sein können (mehr dazu in Abschnitt 3.3.3).
Die Präsentationsformen benutzen die Widgets des Toolkits. Hier muß die Anbindung über den Callback-Mechanismus des Toolkits realisiert werden, da man in der Regel die Widgets nicht verändern kann.

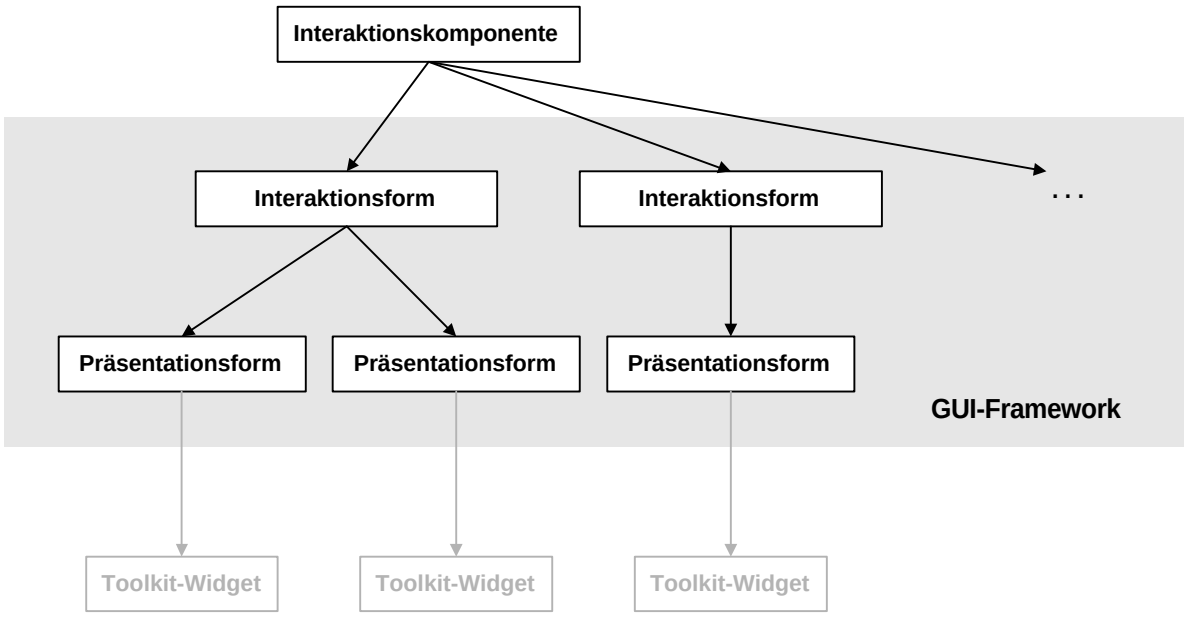


Abbildung 6: Die Struktur der Oberflächenanbindung an die Interaktionskomponente des Werkzeuges.

Zu bemerken ist jedoch, daß ein Widget mehrere Interaktionen (und vor allem Interaktionsformen) realisieren kann (mehr dazu im Abschnitt 3.3.3).

Hieraus ergibt sich, daß die Interaktionskomponente des Werkzeuges nur noch mit Interaktionsformen umgeht. Von deren Darstellung mittels Präsentationsformen muß sie keine Kenntnis besitzen. Dies hat den Vorteil, daß sich die Interaktionskomponente nur noch am Umgang orientiert, der eigentlichen Interaktion. Die Darstellung der Oberfläche wird von den Präsentationsformen realisiert. Dadurch kann die Darstellung variieren, ohne daß das Werkzeug verändert werden muß.

3.3.3 Beziehungen zwischen Interaktionsformen und Präsentationsformen

Bei User-Interfaces entsteht oft die Anforderung, eine mögliche Aktion durch den Anwender diesem auf verschiedene, alternative Arten zur Verfügung zu stellen. Ein häufiger Fall ist das Beenden eines Werkzeuges. Diese Aktion durch den Benutzer kann üblicherweise durch Anwählen eines Menüeintrages geschehen, durch „Beenden“ des Fensters durch den „Close“-Button des Fensters oder auch durch Anwahl eines Buttons auf der Oberfläche innerhalb des Fensters. Alle Aktionen drücken die gleiche Interaktion aus. Sie wird gleichzeitig auf unterschiedliche Art und Weise dargestellt. Diese Eigenschaft kann direkt auf das Framework übertragen werden. Eine Interaktionsform benutzt demnach mehrere Präsentationsformen. Für den Fall, daß eine solche Konstruktion auf eine Auswahl angewendet wird, stellt sich das Problem, daß der Auswahlzustand der Listen konsistent gehalten werden muß. Wird in einer Liste etwas ausgewählt, ist die andere Liste gezwungen, die Auswahl nachzuvollziehen. Diese Synchronisation der Präsentationsformen kann von der Interaktionsform realisiert werden, da sie von jeder Präsentationsform über eine eventuelle Auswahl benachrichtigt wird und dies an die anderen Präsentationsformen weiterleiten kann.
Darüber hinaus existiert auch der umgekehrte Fall. Stellt man sich ein Listbox vor, so könnte diese sowohl die Interaktionsform „1-aus-N-Auswahl“ erfüllen, als auch die Interaktionsform „Aktivator“ (ein Doppelklick auf ein selektiertes Element). Bei der Verwendung einer Auswahl ist diese oftmals gekoppelt mit einer nachfolgenden Aktion. Beispielhaft sei hier ein Werkzeug zur Kontenbearbeitung erwähnt, welches eine Liste von Konten bereitstellt und auf Knopfdruck das selektierte Konto in seinen Einzelheiten darstellt. Dieses Darstellen der Einzelheiten soll auch mit einem Doppelklick auf das Konto in der Liste ausgelöst werden können. Hier zeigt sich, daß das Aktivieren des Buttons „Konto anzeigen“ dieselbe Interaktion darstellt, wie der Doppelklick auf ein Element der Liste und die Listbox die Interaktion der „1-aus-n-Auswahl“ und die Interaktion „Aktivator“ darstellt (siehe Abbildung 7).

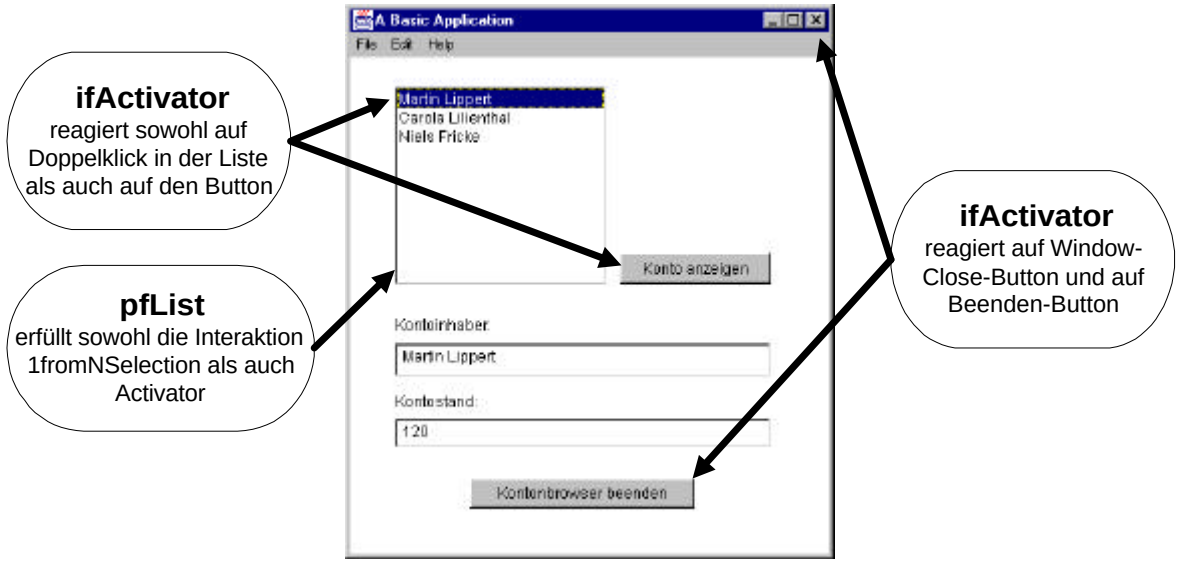


Abbildung 7: Ein Beispielwerkzeug zur Illustration von Beziehungen zwischen Interaktions- und Präsentationsformen. Die Listbox ist zur Laufzeit sowohl an einen IAF-Activator bebunden, als auch an einen IAF-1fromNSelection. Zusätzlich können Interaktionsformen an mehrere Präsentationsformen angebunden sein (hier ein Activator an den Beenden-Button und den Close-Button des Fensters, als auch ein Activator an die Listbox und einen Button).

Das bedeutet, daß zwei verschiedene Interaktionsformen durchaus ein und dieselbe Präsentationsform benutzen können.
Die Verbindung zwischen den Präsentationsformen und den Interaktionsformen wird von einem Kontext hergestellt, dem Interaktionsformen-Kontext bzw. dem Präsentationsformen-Kontext. An diesen Kontext können sich die Interaktionsformen wenden, wenn sie die ihnen zugeordneten Präsentationsformen erfragen möchten. Dieser Kontext wird von einem GUI-Manager geliefert, der die entsprechenden Kontext-Objekte für die einzelnen Werkzeuge liefert.

3.3.4 Der Command-Mechanismus zur losen Kopplung

Die Begrifflichkeiten dieses Abschnittes orientieren sich an der Taxonomie, wie sie in [RW96] beschrieben ist. Daraus stammt folgende Begriffsdefinition:
„Ein *Reaktionsmechanismus* stellt die Möglichkeit dar, Komponenten abstrakt über Änderungen an einer anderen Komponente zu informieren, so daß diese informierten Komponenten auf diese Änderung *reagieren* können."

Die zwei Schichten des Frameworks (Interaktionsform, Präsentationsform) stehen miteinander in Beziehung. Hinzu kommt die Beziehung zur Interaktionskomponente, die Schnittstelle zwischen dem Framework und der das Framework benutzenden Applikation. Jedoch sind diese Beziehungen nicht einseitig, so wie es in Abbildung 6 erscheint. Es reicht nicht aus, daß beispielsweise die Interaktionsformen die Präsentationsformen kennen. Um die Aktion an der Oberfläche von der Präsentationsform an die Interaktionsform weiterzuleiten und von der Interaktionsform an die Interaktionskomponente, ist eine Kopplung in dieser Richtung zusätzlich notwendig. In der objektorientierten Konstruktion hat sich hierfür die Verwendung von Reaktionsmechanismen zur losen Kopplung als sinnvoll erwiesen.
Die Interaktionsformen müssen von den Präsentationsformen benachrichtigt werden. Diese Benachrichtigung könnte als Event realisiert werden. Allerdings hat die Erfahrung im Arbeits-

bereich gezeigt, daß diese Kopplung nicht der eines Event-Mechanismus entspricht, sondern einen kommando-artigen Charakter besitzt. Grund hierfür ist die Tatsache, daß ein Event (vgl. [RW96]) einer Bekanntmachung eines Zustandswechsels dient, für die sich Beobachter interessieren können (oder auch nicht). In der Regel werden die Beobachter den neuen Zustand sondieren und entsprechend darauf reagieren. Bei der Kopplung zwischen Interaktionsform und Präsentationsform handelt es sich nicht um diese Form vom Verbindung. Eine Präsentationsform möchte ganz direkt ihrer Interaktionsform mitteilen, daß eine Aktion stattgefunden hat. Die Präsentationsform soll die Interaktionsform zwar nicht direkt kennen (feste Kopplung), aber sie besitzt dennoch das Wissen, daß sie möglicherweise durch die Benutzeraktion eine direkte Aktion an einer anderen Komponente auslösen soll. Es liegt also nahe, hier keinen Event-Mechanismus zu verwenden, sondern einen Command-Mechanismus, der diese lose Kopplung realisiert.
Die Kopplung der Interaktionsformen an die Interaktionskomponente ist ähnlich. Auch zwischen IAF und IAK soll kein Zustandswechsel bekanntgemacht werden, der sondiert werden muß, sondern eine Aktion soll ausgelöst werden. Daher wird auch für die Anbindung der IAFs an die benutzenden Komponenten der Applikation ein Command-Mechanismus verwendet.
Die Verbindung zwischen den Präsentationsformen und dem Toolkit bzw. den Widgets des Toolkits kann nicht durch einen eigenen Reaktionsmechanismus realisiert werden. Bei der Anbindung der PFs an die Widgets des Toolkits ist man zwangsläufig auf den Mechanismus der Widgets angewiesen.
Mit der losen Kopplung zwischen IAFs und PFs ist gewährleistet, daß die einzelnen Elemente der zwei Schichten des Frameworks leicht ausgetauscht und verändert werden können, ohne andere Teile des Frameworks zu beeinflussen. Mit der Kopplung mittels Command-Mechanismus zwischen den Interaktionsformen und der Interaktionskomponente ist sichergestellt, daß auch dort Teile einfach ausgetauscht und angepaßt werden können (siehe Abbildung 8). Abbildung 8 zeigt beispielhaft den Verlauf des Kontrollflusses bei der Aktivierung eines Button-Widgets.
Bei näherer Betrachtung der Rolle der Interaktionsformen bei dieser Benachrichtigung stellt man fest, daß die Commands im wesentlichen durchgereicht werden. Daraus könnte die Behauptung wachsen, daß dieses „Durchreichen“ unnötig ist und die Präsentationsformen das Command direkt an die Interaktionskomponente senden könnten. Das ließe sich so konstruieren, daß die Präsentationsform keine Kenntnis von der Interaktionskomponente bekommen müßte, ebenso wie die Interaktionskomponente keine Kenntnis von der Präsentationsform bräuchte. Die lose Kopplung wäre also weiterhin gewährleistet. Allerdings bekäme man dadurch einen Bruch in der Konstruktion. Die klare Trennung der Schichten Interaktionskomponente-Interaktionsform-Präsentationsform ginge verloren, ebenso wie die Möglichkeit, in der Interaktionsform eine Synchronisation der Präsentationsformen vorzunehmen (siehe Abschnitt 3.3.3).

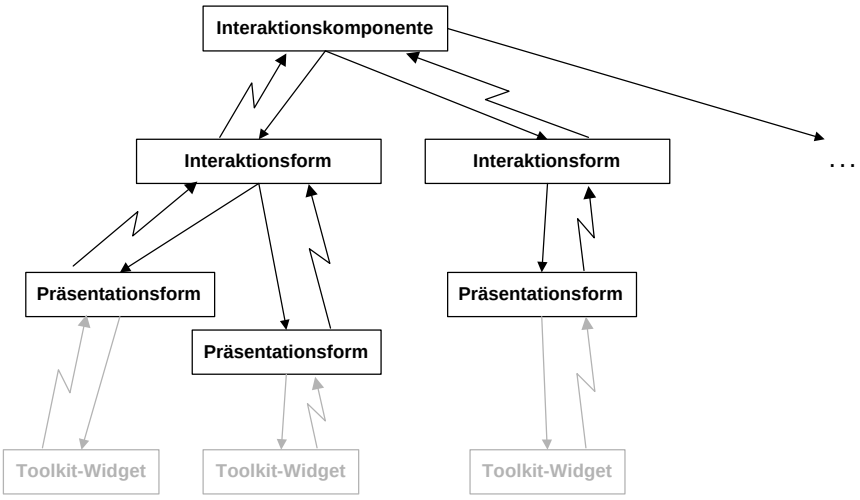


Abbildung 8: Die Kopplung zwischen den Schichten des GUI-Frameworks (Benutzung und Benachrichtigung).

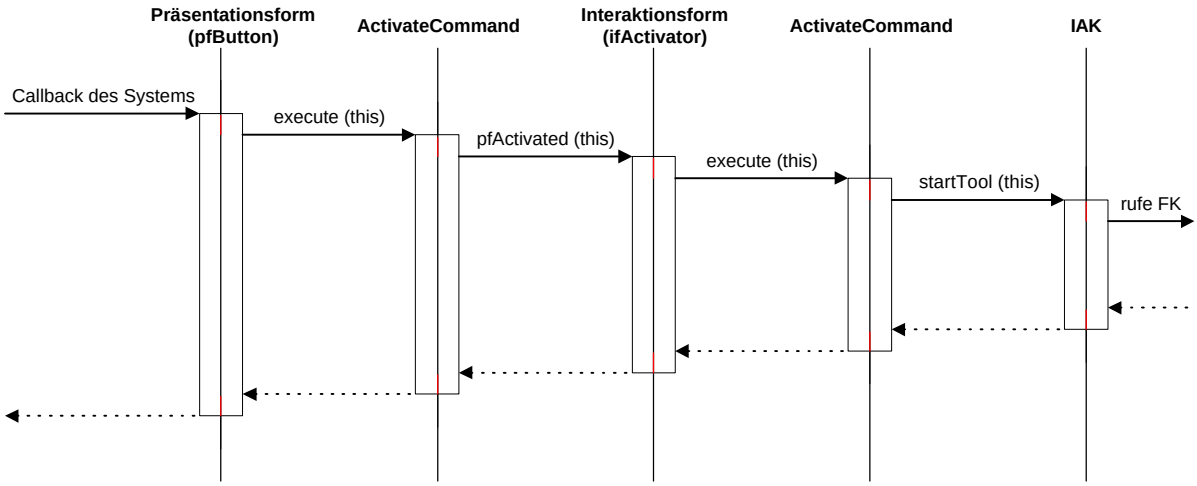


Abbildung 9: Interaktionsdiagramm für eine Aktion durch den Benutzer (hier Druck eines Buttons).

3.3.5 Einordnung des Frameworks

In dieser kurzen Anmerkung baue ich auf Begriffen auf, die unter anderem in [Weiß97] und in [Taligent95] beschrieben sind.

Ziel der Konstruktion eines Frameworks ist es, Konzepte und Design wiederzuverwenden. Nach der Art dieser Wiederverwendung werden Frameworks in Black-Box-Frameworks und White-Box-Frameworks unterschieden. Black-Box-Frameworks zeichnen sich dadurch aus, daß ihre Klassen durch Benutzung wiederverwendet werden und an die Bedürfnisse des Benutzers nur gegebenenfalls mittels Parameterklassen angepaßt werden. White-Box-Frameworks hingegen stellen Klassen zur Verfügung, die durch Erben und gezieltes Überschreiben von Methoden dieser Klassen wiederverwendet werden.[10]

Black-Box-Frameworks zeichnen sich dadurch aus, daß ihre Verwendung einfacher ist, als die der White-Box-Frameworks. Black-Box-Frameworks erfordern ein geringeres Maß an Verständnis darüber, wie das Framework im Innern konzipiert ist und funktioniert. Bei dem zu dieser Arbeit entwickelten GUI-Framework handelt es sich um ein Black-Box-Framework. Die im Framework vorhandenen Klassen werden in erster Linie nur benutzt. Es findet sich

[10] Dies umfaßt nicht das gesamte Spektrum der Framework-Charakterisierung. Genauere Erklärungen und Definitionen finden sich in [Weiß97] und [Taligent95].

keine typische White-Box-Verwendung, indem von den Framework-Klassen geerbt wird und Methoden überschrieben oder erst implementiert werden, um die benötigte Funktionalität zu erreichen. In dieser Hinsicht ähnelt es dem GUI-Framework in der BibV30, das mittels Interaktionstypen eine Kapselung der Benutzeroberfläche vornahm. Auch diese Kapselung gab Klassen vor, die in erster Linie benutzt und nicht beerbt wurden.
Das hat zur Folge, daß sich dieses GUI-Framework einfach benutzen läßt und keinen größeren Aufwand zur Einarbeitung nötig macht. Hinzu kommt, daß das Framework für den Applikationsentwickler eine Schnittstelle bietet, die relativ schmal ist. Der Entwickler von Interaktionskomponenten kann sich auf die Benutzung von Interaktionsformen konzentrieren. Er befaßt sich nicht mit den anderen Komponenten des Frameworks, den Präsentationsformen, oder gar den Widgets des Toolkits. Er benötigt kein Wissen darüber, wie die Interaktionsformen mit den darunterliegenden Schichten zusammenarbeiten. Er kennt nur die oberste Schicht des Frameworks, die Interaktionsformen. Dies macht die Abstraktion für die IAK auf die eigentliche Interaktion zusätzlich deutlich.

3.4 Einbeziehung von GUI-Buildern

Bei der Entwicklung großer, interaktiver Software reichen die einfachen Werkzeuge, wie Compiler und Debugger, für Entwickler oftmals nicht aus. Zur Gestaltung der Oberfläche ist ein spezielles Werkzeug notwendig. Diese Lücke schließen die GUI-Builder, die bei vielen Entwicklungsumgebungen mitgeliefert werden. Auch für Java gibt es inzwischen eine Vielzahl solcher graphischer User-Interface-Builder.

3.4.1 Die Abhängigkeit vom GUI-Builder

Da es jedoch, auch in Java, keine allgemeine, standardisierte Beschreibung eines graphischen User-Interfaces gibt, haben die verschiedenen Hersteller diese Lücke auf ihre eigene Art und Weise gelöst, jeder Hersteller unterschiedlich. Die meisten GUI-Builder für Java erzeugen Java-Source-Code, der die Oberfläche zur Laufzeit erzeugt. Zur Bearbeitung im GUI-Builder benutzen sie ein eigenes Format, welches es ermöglicht, die einmal gestaltete Oberfläche zu speichern und zu verändern. So brauchen die GUI-Builder nicht den einmal erzeugten Java-Source-Code zu parsen, um die nötigen Informationen zu bekommen, sondern können ihr spezielles Beschreibungsformat verwenden.
Man kann darüber spekulieren, wieso eine solche Konstruktion gewählt wurde. Fest steht, daß sich daraus eine Abhängigkeit von GUI-Builder ergibt. Hat man in einem Software-Projekt einmal angefangen, einen bestimmten GUI-Builder zu benutzen, kann man ihn nicht einfach gegen einen anderen auswechseln, ohne die gesamten Oberflächen unverändert zu lassen oder, im Falle einer Änderung, mit dem neuen GUI-Builder erneut zu erstellen. Der Grund hierfür ist die Tatsache, daß kein GUI-Builder „fremde“ (von einem anderen GUI-Builder erzeugte) Resourcen lesen kann.
Hinzu kommt, daß die Konstruktion eines GUI-Frameworks erschwert wird. Dieses Framework muß auf die erzeugten Source-Codes des GUI-Builders abgestimmt werden. Das resultiert daraus, daß die verschiedenen GUI-Builder äußert unterschiedliche Source-Codes für eine Oberfläche generieren. Einige GUI-Builder erzeugen die Source-Codes unter Benutzung bestimmter, vom Entwicklungstool implementierter Abstraktionen, wie Wrapper-Klassen. Zusätzlich haben viele GUI-Builder eine festgelegte Sichtweise von der Applikationsentwicklung, die in die Erzeugung der Source-Codes eingeht. Diese Sichtweise legt beispielsweise den grundlegenden Aufbau einer Applikation und den Kontrollfluß innerhalb der Applikation fest.
Als ein Beispiel kann hier der Java-Workshop von Sun aufgeführt werden. Der Java-

Workshop arbeitet nicht direkt mit den AWT-Klassen, sondern benutzt eine relativ große, eigene Klassenbibliothek, die die AWT-Klassen kapseln (mittels Wrappers). Zusätzlich fügen diese Workshop-spezifischen Klassen eigene Widgets hinzu, die in der Standard-AWT-Bibliothek nicht vorhanden sind. Hinzu kommt, daß der Java-Workshop eine bestimmte Philosophie bei der Generierung der Source-Codes, die die Oberfläche erzeugen, verfolgt. Er erzeugt sowohl ein Hauptprogramm, als auch einen Source-Code, in dem die Oberfläche konstruiert wird. Diese Vorgehensweise hat eine Reihe von Folgen: Durch diese Konstruktion ist es bedingt, daß die erzeugte Applikation immer ein Hauptfenster besitzen muß. Wird dieses Fenster geschlossen, wird die komplette Applikation beendet.[11] Hinzu kommen kleine Besonderheiten, die bei der Konstruktion mit dem Java-Workshop einzuhalten sind (eindeutige Namensgebung im ganzen Projekt, u.a.).
Daraus ist schon ersichtlich, daß die GUI-Builder ihre Philosophie der Applikationsentwicklung verfolgen und diese auch konsequent durchsetzen. Das spiegelt wieder, daß die Hersteller der Entwicklungstools von der Voraussetzung ausgehen, daß eine Anwendung mit *einem* Entwicklungstool entwickelt wird. Eine heterogene Zusammenstellung der Entwicklungstools innerhalb eines Software-Projektes ist quasi nicht möglich, ebensowenig wie der Wechsel des Entwicklungstools während eines Projektes.
Daraus resultiert, daß man für jeden GUI-Builder ein individuell angepaßtes GUI-Framework bauen müßte, welches die spezielle Philosophie des GUI-Builders berücksichtigt und auf die Art der erzeugten Source-Codes abgestimmt ist. Auch hier wäre eine heterogene Zusammenstellung in einer Applikation sicherlich nur durch erheblichen Zusatzaufwand zu lösen, wenn überhaupt möglich.
Daß dies durchaus wünschenswert sein kann zeigt die aktuelle Situation der Java-Entwicklungsumgebungen. Eine Vielzahl von Tools steht zur Verfügung und die Entwicklungszyklen dieser Tools sind extrem kurz. Die daraus resultierende Unsicherheit daürber, welches Tool in Zukunft noch weiterentwickelt wird oder ob ein zukünftiges Tool geeigneter sein könnte, ist sehr groß. Diese Tatsache wird durch die rasante Entwicklung der Sprache Java noch unterstützt. Kann sich der Entwickler von dieser Entwicklung unabhängig machen, indem er auf zukünftige Entwicklungen flexibel reagiert und nicht von dem verwendeten Tool abhängig ist, wäre diese Unsicherheit entscheident gemildert.

3.4.2 Mögliche Lösungsansätze

Um vom dem GUI-Builder unabhängig zu werden, sind verschiedene generelle Lösungen denkbar, die jedoch alle an bestimmte Bedingungen gekoppelt sind:

- Eine mögliche Lösung liegt in einem einheitlichen Resourcen-Format, welches alle GUI-Builder verstehen und verarbeiten können. Dieses Resourcen-Format könnte von einem Framework verstanden werden. Dies wäre die Ideallösung.
- Da ein allgemeines Resourcen-Format im allgemeinen nicht spezifiziert ist und sich keine Entwicklung dahin abzeichnet, ist man gezwungen, andere Wege zu gehen. Liegt in der verwendeten Sprache und Umgebung ein Komponentenmodell vor, kann dies zur Konstruktion verwendet werden. Komponenten können in der Regel in GUI-Buildern verwendet werden. Über speziell entwickelte Komponenten wäre es so möglich, ein eigenes Resource-Format zu konstruieren, welches unabhängig von den verwendeten GUI-Builder ist. Eine spezielle Komponente könnte aus im GUI-Builder erzeugten Oberflächen das allgemeine Resourcen-Format erzeugen. Ebenso könnte eine spezielle Komponente das allgemeine Resourcen-Format wieder direkt in den GUI-Builder importieren, sollten die Re-

[11] Der Java-Workshop ist dabei so radikal, daß die JVM (Java-Virtual-Machine) beendet wird. So ist sichergestellt, daß man dieses „Feature“ auf keinen Fall umgehen kann.

sourcen verändert werden sollen. Das setzt allerdings voraus, daß die GUI-Builder komplett in das Komponentenmodell integriert sind. Das bedeutet, daß die Oberflächenkomponenten, die im GUI-Builder zusammengestellt werden können, schon zur Design-Zeit als echte Objekte der Komponenten vorliegen. Darüber hinaus muß der GUI-Builder in der Lage sein, von Komponenten importierte Objekte zu verarbeiten. Dies wäre eine praktikable Lösung, jedoch besitzt sie Voraussetzungen, die nur von wenigen Sprachen und Entwicklungsumgebungen erfüllt werden. Zudem macht man sich mit dieser Lösung stark von dem Komponentenmodell abhängig. Gegen Änderungen an dem Komponentenmodell ist diese Möglichkeit nicht abgesichert.

- Eine weitere Möglichkeit ist, Konverter zu bauen, die die verschiedenen Resource-Formate in ein eigenes Resource-Format konvertieren und umgekehrt. Dies ist allerdings ein hoher Aufwand, auch wenn er nur für verwendete GUI-Builder realisiert werden muß. Hierzu muß das Resourcen-Format der entsprechenden GUI-Builder bekannt sein. Das ist eine Lösung, die zwar aufwendig zu realisieren, dafür aber auf viele Sprachen und Entwicklungsumgebungen anwendbar ist.

4 Realisierung des GUI-Frameworks in Java

Der Schwerpunkt dieser Arbeit liegt auf der Beschreibung der Realisierung des GUI-Frameworks, so wie es im vorigen Abschnitt skizziert wurde. Als Sprache hierzu wurde Java gewählt, da derzeit am Arbeitsbereich ein WAM-Framework in Java entwickelt wird und dieses GUI-Framework Teil des WAM-Frameworks werden sollte. Die Beschreibung der Realisierung teilt sich in zwei Bereiche, die an der Struktur des vorangegangenen Abschnittes orientiert sind.

4.1 Der Aufbau des Frameworks

Der erste Teil beschreibt den grundsätzlichen Aufbau des Frameworks in Java und die Klassenstruktur, die dazu gewählt wurde. Dazu wird die Einbettung in die Java-Umgebung beschrieben, die Umsetzung der Interaktions- und Präsentationsformen in Java und deren Kontext und die Strukturierung des Frameworks in Java-Packages.

4.1.1 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[12]. Mit der Einführung des JDK1.2 voraussichtlich Ende ’97 werden die „Java Foundation Classes“ zu den Standard-Packages gehören. Diese Klassen bieten eine Alternative zu den AWT-Klassen. Die Frage, ob die „Java Foundation Classes“ die AWT auf lange Sicht ablösen, ist ungewiß. Aber schon die Möglichkeit zeigt, 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-Frameworks in Java zwar auf dem AWT aufzubauen (derzeit die einzige Möglichkeit), sich jedoch nicht zu stark 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 Toolkits (wie „Java Foundation Classes“) flexibel und mit wenig Aufwand reagieren zu können.

4.1.2 Interaktionsformen in Java

Die Implementierung der Interaktionsformen in Java ist durch Java-Klassen umgesetzt. Wie Abbildung 5 darstellt, existiert eine Hierarchie von Interaktionsformen, ausgehend von einer Basisklasse `ipObject`. An ihrer Schnittstelle bieten sie die Möglichkeit, Command-Objekte anzumelden, um eine lose Kopplung an die Interaktionskomponente zu realisieren.
Diese Interaktionsform-Klassen verweisen auf möglicherweise mehrere Präsentationsformen. An diese Präsentationsformen sind die Interaktionsformen über einen Command-Mechanismus gekoppelt. Auf Commands von den Präsentationsformen reagieren die Interaktionsformen

[12] Der Compiler gibt bei der Compilierung alter Sourcen „deprecated“-Warnings aus, deren Anzahl erheblich sein kann.

zunächst mit einer Synchronisation der Präsentationsformen, sofern nötig. Anschließend leiten sie die Aktion an die Interaktionskomponente weiter, indem das vom der Interaktionskomponente angemeldete Command ausgeführt wird.

4.1.3 Präsentationsformen in Java

Aufgabe der Präsentationsformen ist zum einen, verschiedene Darstellungen zu einer Interaktionsform zu ermöglichen und zum anderen, die Widgets des Toolkits anzubinden, auf die gezwungenermaßen zurückgegriffen werden muß.
Da die Widgets der Toolkits in der Regel jedoch nicht am Umgang strukturiert sind, 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 10).
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.

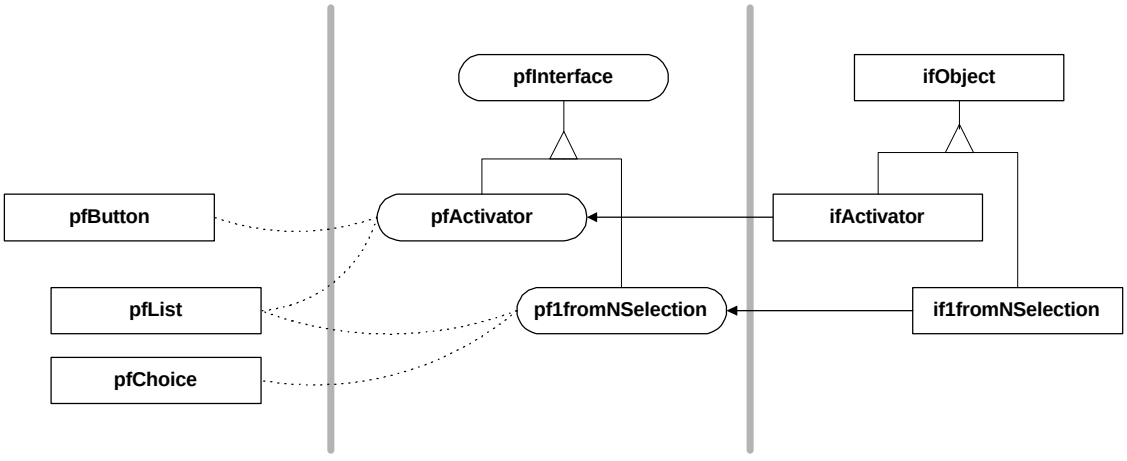


Abbildung 10: Eine Hierarchie von Interaktionsformen (rechts), die die entsprechenden Präsentationsform-Interfaces benutzen (mitte), die von den Präsentationsform-Klassen implementiert werden (links). Die grauen Linien trennen die Schichten des Frameworks.

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 Framework aus drei Schichten (Interaktionsform-Klassen, Präsentationsform-Interfaces, Präsentati-

onsform-Klassen).
Zum zweiten bietet sich in Java die Möglichkeit an, die Präsentationsform-Klassen von den Widget-Klassen erben zu lassen (siehe Abbildung 11). 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 Frameworks liegt und vor dem Benutzer des Frameworks komplett verborgen ist, halte ich 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-Klassen. So kann im GUI-Builder direkt der „Interaktionsname“ vergeben werden, unter dem die Präsentationsform dem Framework bekannt gibt, für welche Interaktion diese Präsentation steht. Über diesen Namen wird die Präsentationsform vom Framework 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. Wie in Abschnitt 3.3.3 erläutert ist es jedoch notwendig, daß eine Präsentationsform mehrere „Interaktionsnamen“ besitzt und das mehrere Präsentationsformen ein und denselben „Interaktionsnamen“ besitzen, da sie *eine* Interaktion darstellen.
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 Frameworks 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 Frameworks nicht von Bedeutung. Sie ist nur innerhalb des Frameworks bekannt. Deshalb ist diese Konstruktion vertretbar.

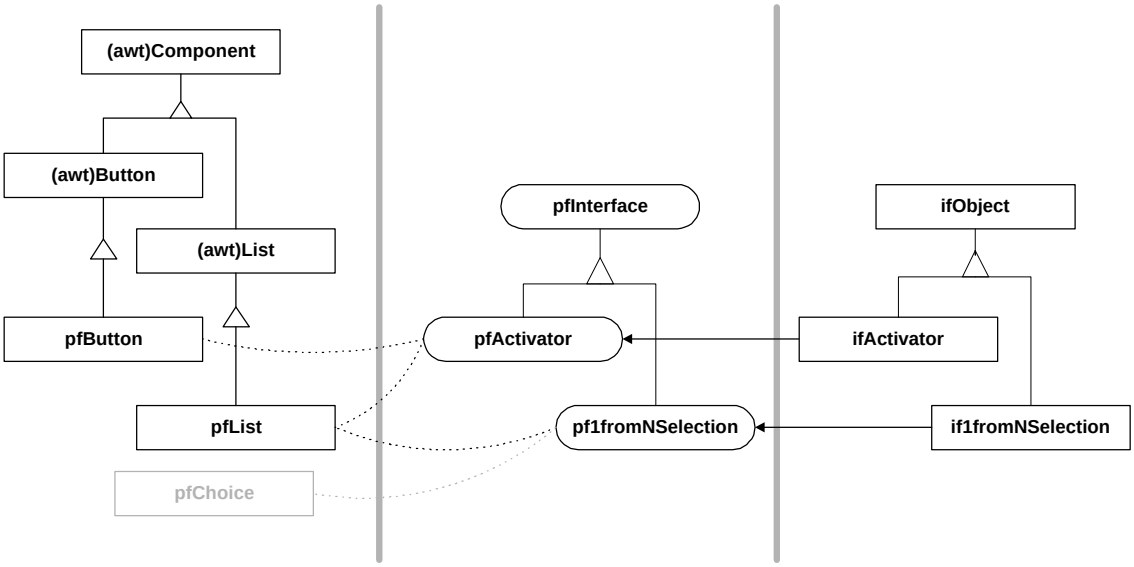


Abbildung 11: Die Präsentationsform-Klassen erben von den AWT-Klassen (links).

Da eine Präsentationsform mehrere Interaktionen darstellen kann (siehe Abschnitt 3.3.3), muß beispielsweise die Präsentationsform-Klasse `pfList` sowohl das PF-Interface `pfActivator` als auch das PF-Interface `pf1fromNSelection` implementieren (siehe Abbildung 10 und Abbildung 11). 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, da 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 „Methodenkollision“ 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 Framework 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 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.

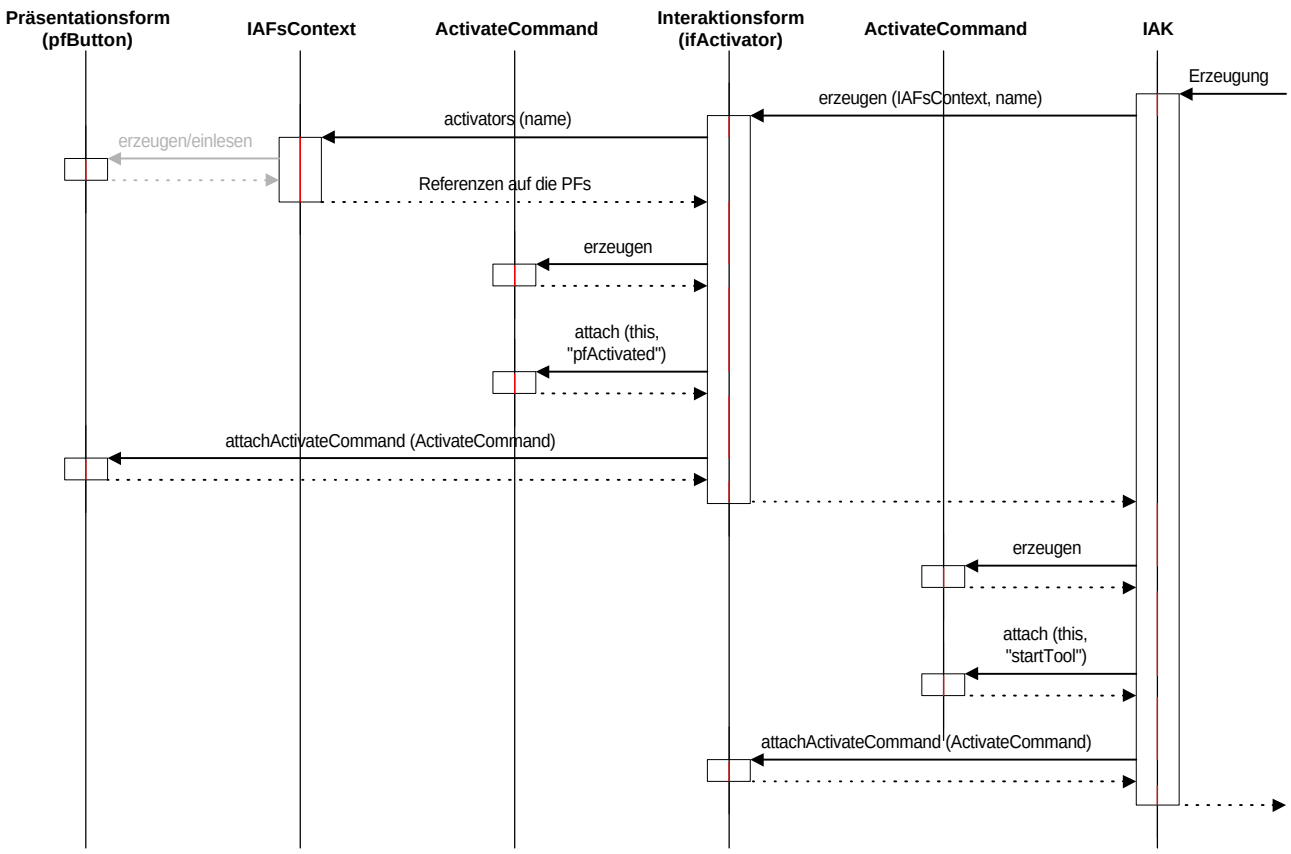


Abbildung 12: Interaktionsdiagramm für die Konstruktion der Interaktionsformen und deren Anbindung an die Präsentationsformen und die IAK.

4.1.4 Der Kontext der Interaktions- und Präsentationsformen

Wie im Abschnitt 3.3.3 beschrieben, benötigen die Interaktionsformen eine Möglichkeit, die ihnen zugeordneten Präsentationsformen zu referenzieren. Hierzu existiert im Framework 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 [Görtz97]).

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, exitiert ein Interface `IAFsContext`. Die Interaktionsformen bekommen ein Objekt vom Typ `IAFsContext`. 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 Resourcen 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 Interaktion 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 Resourcen zu erstellen und statisch festzulegen. Zu den statisch durch einen GUI-Builder erzeugten Resourcen können, für die Interaktionskomponenten und die Interaktionsformen völlig transparent, dynamische und/oder nicht durch GUI-Builder erzeugbare Oberflächen integriert werden.

4.1.5 Strukturierung des Frameworks in Packages

Ein Framework muß klar strukturiert sein um verständlich und verwendbar zu sein. Diese Strukturierung sollte sich nicht nur in Klassendiagrammen wiederfinden, sondern auch bei der praktischen Verwendung des Frameworks sichtbar sein. In C++-Frameworks wird hierzu häufig die Einteilung in „Projekte“ genutzt. Diese Projekteinteilung ist allerdings häufig an das Entwicklungssystem gekoppelt. Dem Source-Code ist nicht anzusehen, in welchem Projekt er sich befindet. Wird das Framework mit einer anderen Entwicklungsumgebung verwendet, muß die Projektstruktur erneut nachempfunden werden.
In Java bietet sich für die Strukturierung des Frameworks der Package-Mechanismus an (siehe 2.1.11). Dieser Package-Mechanismus in Java gliedert Java-Code hierarchisch. Die einzelnen Stufen der Hierarchie werden durch Punkte getrennt. Der Vorteil dieser Methode ist, daß die Strukturierung unmittelbar im Source-Code sichtbar ist (durch das Schlüsselwort `package`).
Stellt sich die Frage, welche Klassen und Interfaces in welche Packages gepackt werden sollen. Hierfür bieten sich zwei Möglichkeiten an:

- Die Packages orientieren sich an der Vererbungshierarchie,
- Die Packages orientieren sich an den Dienstleistungen.

Für das GUI-Framework in Java wurde die zweite Alternative gewählt. Die Orientierung der Packages an den Dienstleistungen erleichert das Verständnis der Package-Hierarchie und vereinfacht so die Einarbeitung in das Framework und dessen Verwendung. Die Benennung der Packages erfolgt so nach den Dienstleistungen.
Zusätzlich ist das Framework grundsätzlich in drei Layer (aus Package-Hierarchie-Sicht) gegliedert:

- `wam`
- `gui`
- `interaction, presentation, management`

Das erste Layer `wam` dient der Kennzeichnung des Frameworks als zugehörig zum WAM-Kontext. Dieses Layer ist das erste des gesamten Java-Frameworks für WAM. Das zweite Layer `gui` kennzeichnet die Aufgabe dieses Teils des Frameworks. Es ist für das graphische User-Interface zuständig. Der dritte Layer unterteilt die verschiedenen Aufgaben innerhalb des GUI-Frameworks in `interaction`, `presentation` und `management`.

- Im Package `wam.gui.interaction` finden sich die Klassen der Interaktionsformen. In der Regel wird der Entwickler von Werkzeugen hauptsächlich dieses Package benutzen.
- Das Package `wam.gui.presentation` ist noch einmal gegliedert in drei Unterpackages. Diese sind:
  - `wam.gui.presentation.interface`
  - `wam.gui.presentation.implementation`
  - `wam.gui.presentation.util`

Diese vierte Stufe der Unterteilung ist nicht mehr zwingend nach den Dienstleistungen benannt. Hier werden die Präsentationsform-Interfaces von den Präsentationsform-Klassen getrennt. Im Package `util` finden sich Hilfsklassen (z.B. Hilfsklassen für das Tooltip[13]).

- Das Package `wam.gui.management` beinhaltet die verschiedenen GUI-Manager. Diese GUI-Manager dienen dem Werkzeug dazu, die Verbindung zu den Resourcen aufzubauen.

Der Benutzer des Frameworks kann sich mit diesen Packages leicht einen Überblick verschaffen. Die Benennung der Packages nach den Aufgaben erleichert dies zusätzlich. Die Tatsache, daß die Packages abgeschlossen sind, also vom Benutzer des Frameworks nicht erweitert werden, trennt das Framework zusätzlich von der Anwendung.
Die Vererbungshierarchie wird durch diese Art der Strukturierung in Packages nicht unmittelbar deutlich. Dies geschieht durch Kürzel vor den Klassen- und Interfacenamen. Diese Abkürzungen sind in den Styleguides zu dem Framework festgehalten (siehe [Style97]). Diese, zwischen zwei und drei Buchstaben langen Kürzel kennzeichnen die Vererbungshierarchie. Daher besitzen alle Interaktionsformen das Kürzel „if“ vor dem eigentlichen Namen. Präsentationsformen tragen das Kürzel „pf“.

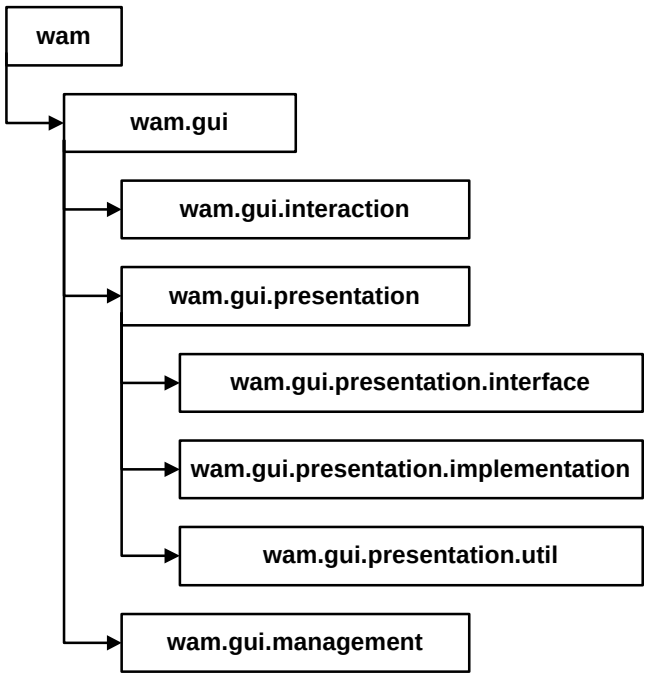


Abbildung 13: Die Package-Hierarchie im Überblick (Pfeile markieren die Verzweigung in Sub-Packages).

4.2 Unabhängigkeit vom GUI-Builder

Ziel sollte sein, ein GUI-Framework zu bauen, welches sich mit beliebigen Java-GUI-Buildern benutzen läßt. Außerdem sollte in einem Software-Projekt eine heterogene Zusammenstellung der Entwicklungstools möglich sein, sowie der Austausch der Resourcen problemlos funktionieren. Die Problematik ist im Abschnitt 3.4 beschrieben. Da in Java ein standardisiertes Komponentenmodells vorhanden ist, habe ich die komponentenbasierte Lösungsvariante aus Ab-

[13] Als „Tooltip“ wird hier die Hilfsfunktion genannt, die bei Berührung einer Komponente an der Oberfläche mit der Maus einen Hilfstext einblendet (auch als „Bubble-Help“ oder „Balloon-Help“ bekannt).

schnitt 3.4.2 für eine Umsetzung in Java gewählt. Hierzu habe ich ein Framework in Java implementiert[14], welches mittels JavaBeans nahezu die Unabhängigkeit von verwendeten GUI-Buildern realisiert. Inzwischen unterstützen die meisten am Markt verfügbaren Entwicklungstools die Java-Version 1.1 und somit auch JavaBeans[15]. Völlige Unabhängigkeit erreicht man mit dieser Lösung jedoch auch nicht. Die Abhängigkeit vom Komponentenmodell bleibt. Da JavaBeans jedoch von Sun als Standard etabliert ist, halte ich die Abhängigkeit von den JavaBeans für vertretbar.
Mit diesen JavaBeans ist es unter anderem möglich, visuell Komponenten zu manipulieren.[16] Primär ist diese visuelle Manipulation natürlich zur Konstruktion von graphischen Benutzeroberflächen geeignet. Daher sind die AWT-Klassen als JavaBeans implementiert und die neueren GUI-Builder unterstützen ebenfalls JavaBeans.
Da die AWT-Klassen schon als JavaBeans implementiert sind, werden von ihnen abgeleitete Klassen, wie die neuen PF-Klassen, ebenfalls JavaBeans. Um auch die Attribute der PF-Klassen in GUI-Buildern setzen zu können (z. B. bei einem `pfActivator` den „Activator-Name“), sind die in den PF-Klassen definierten Methoden nach den Konventionen der JavaBeans implementiert. Das bedeutet, daß diese PF-Klassen direkt in allen GUI-Buildern verwendet werden können, die JavaBeans unterstützen. Man kann so im GUI-Builder direkt die Präsentationsformen anordnen und entsprechend konfigurieren, d. h. zum Beispiel mit dem Interaktionsnamen versehen, der es dem Framework ermöglicht, die Präsentationsform wiederzufinden. Dies wird vom GUI-Manager benötigt, um für eine Interaktionsform auf Anfrage die passende Präsentationsform zu finden.

Dies allein ist jedoch nur ein Schritt auf dem Wege zur Unabhängigkeit vom GUI-Builder, da die Resourcen noch immer nicht austauschbar sind.
Um dies zu erreichen ist es notwendig, ein allgemeines Resourcen-Format zu definieren, welches von den GUI-Buildern erzeugt und nach Möglichkeit auch von den GUI-Buildern wieder eingelesen werden kann. Da die GUI-Builder jedoch in der Regel kein solches allgemeines Format kennen, muß ein solches Format erst einmal gefunden werden. Als ein solches Resourcen-Format bietet sich in Java die Objektserialisation an (siehe [JOS96]), die standardmäßig vorhanden ist. Um in Java Objekte zu serialisieren, ist nur sehr wenig Aufwand nötig, da die Grundfunktionalität in der Virtuellen Maschine bereits implementiert ist.
Die Möglichkeit der Serialisierung von zusammengestellten Komponenten macht die „Beanbox“[17] vor. Hier können Exemplare von JavaBeans erzeugt und zusammengestellt werden. Diese Zusammenstellung von Beans läßt sich abspeichern und zu einem späteren Zeitpunkt wieder einlesen. Dies wäre auch für GUI-Builder denkbar, die die zusammengesetzte Oberfläche als Exemplare von Beans serialisieren und später wieder einlesen könnten.
In den derzeitigen GUI-Buildern ist diese Möglichkeit allerdings nicht vorgesehen. Die Gründe hierfür mögen in der Entwicklung dieser GUI-Builder liegen, die größtenteils wohl aus den C++- bzw. Smalltalk-Varianten von GUI-Buildern hervorgegangen sind. Letztlich bleibt dies jedoch Spekulation. Fraglich ist, ob diese Funktionalität der Oberflächen-Objektserialisierung in zukünftigen Versionen von Java-Entwicklungsumgebungen jemals integriert wird. Ankündigungen hierzu gibt es derzeit nicht. Einzige Ausnahme hiervon ist „SodaPop“, eine Entwicklung von Andreas Dangberg von C-Lab. Dieser in Java implementierte GUI-Builder be-

[14] Im Zusammenarbeit mit Niels Fricke, Carola Lilienthal, Stefan Roock und Henning Wolf.
[15] Andere Hersteller haben zumindest schon Versionen ihrer Entwicklungsumgebungen angekündigt, die das JDK1.1 und JavaBeans unterstützen (Stand: September 1997).
[16] Siehe Abschitt 2.1.9. Näheres zu JavaBeans ist in der Spezifikation zu lesen [JBeans96] oder in [Englanger97].
[17] Die „Beanbox“ ist Bestandteil des Bean-Development-Kits von JavaSoft und bietet einen kleinen Testrahmen für JavaBeans, in dem Exemplare von Beans erzeugt und zusammengestellt werden können.

dient sich der Objektserialisierung zur persistenten Speicherung der erzeugten Oberflächen (siehe [Dangberg97]).
Um diese Unsicherheit der Entwicklung der Java-Entwicklungsumgebungen auszuschalten, ist es nötig, auf anderem Wege eine Serialisierung der Oberflächenobjekte aus gängigen GUI-Buildern zu ermöglichen.

4.2.1 Das Serializer-Bean

Dazu habe ich das sogenannte „Serializer-Bean“ entwickelt. Dieses JavaBean ist ein Button, der auf Knopfdruck sein übergeordnetes Objekt serialisiert und in eine Datei schreibt.
Dazu erbt das Serializer-Bean von der AWT-Klasse `Button`. Dadurch läßt sich das Serializer-Bean in GUI-Buildern wie ein normaler Button verwenden. Die Aktion des Klicks auf den Button interpretiert das Bean als Wunsch, die Oberfläche zu serialisieren. Um dies zu realisieren, greift das Serializer-Bean auf sein parent-Objekt in der Objekthierarchie der Oberfläche zurück und versucht, dieses Objekt zu serialisieren. Dies geschieht in Java durch einen einfachen ObjectOutputStream, der das Serialisieren von Objekten implementiert.
Da die Oberflächenkomponenten in Java hierarchisch angeordnet sind (nach dem Composite-Pattern, siehe [GHJ+96]), wird so die komplette Oberfläche (ausgehend von dem übergeordneten Objekt des Serializer-Beans) beispielsweise in ein File geschrieben. Das Serializer-Bean ist Bestandteil der Oberfläche und wird ebenfalls in der Objekthierarchie serialisiert. Da das Serializer-Bean von seinem parent-Objekt ausgeht und dieses serialisiert, werden in der Hierarchie oberhalb des parent-Objektes angeordnete Objekte nicht mitserialsiert. So kann durch die Plazierung des Serializer-Beans exakt gesteuert werden, welcher Teil der Oberfläche serialisert wird.
In Abbildung 14ff. ist ein Beispiel gezeigt. Die Oberfläche dieses Beispiels wurde in einem GUI-Builder zusammengestellt und um das Serializer-Bean ergänzt.

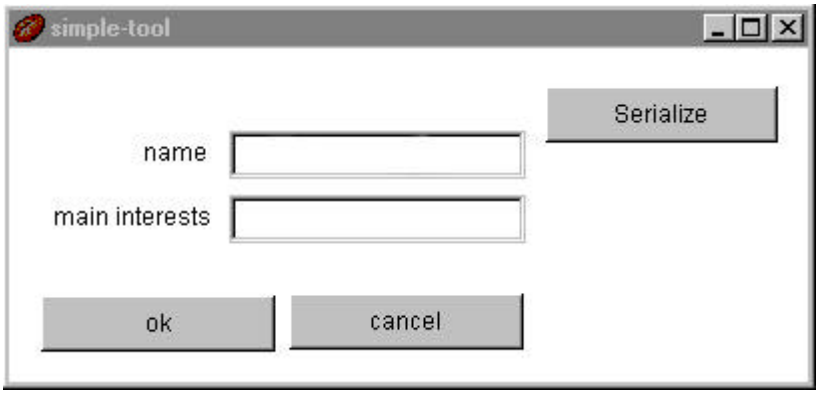


Abbildung 14: Beipsiel für eine im GUI-Builder zusammengestellte Oberfläche für ein Werkzeug. Das Serializer-Bean kann oben rechts als Button erkannt werden.

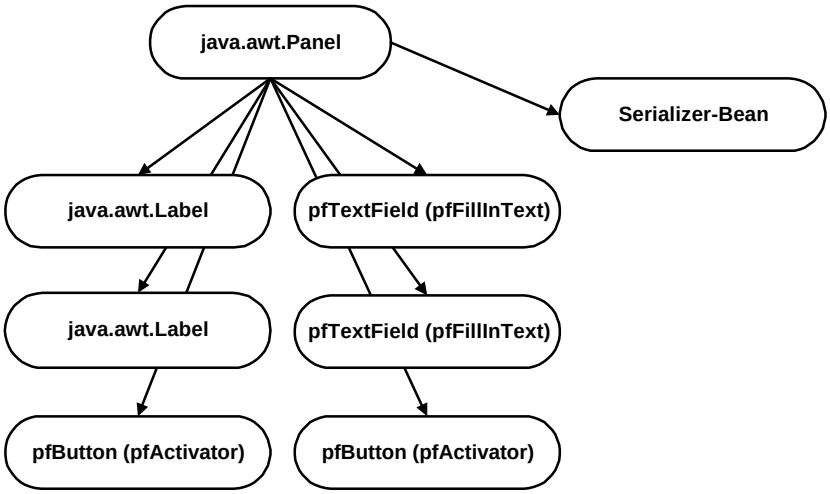


Abbildung 15: Die Objekthierarchie, wie sie vom Serializer-Bean serialisiert wurde.

Das GUI-Framework ist so konstruiert, daß es genau diese serialisierten Oberflächenobjekte einliest, das „Serializer-Bean“ versteckt (da es natürlich mitserialisiert wurde und als Button immer noch in der Oberfläche vorhanden ist) und so die Oberfläche zur Verfügung stellt. Diese Aufgabe wird vom GUI-Manager, einer Klasse des GUI-Frameworks, erledigt. Abbildung 16 zeigt, wie die Oberfläche in der Applikation erscheint.

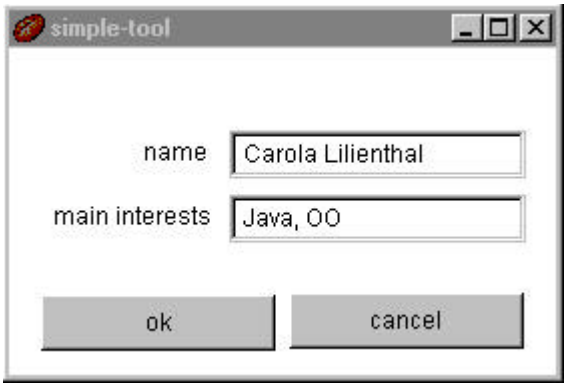


Abbildung 16: Das Werkzeug mit den eingelesenen Oberflächenkomponenten in der Verwendung.

Die zusammengestellte Oberfläche besteht jedoch nicht nur aus Präsentationsform-Objekten. Um Elemente auf einer Oberfläche anzuordnen, wird auf andere Klassen zurückgegriffen. Dies ist zum Beispiel beim Festlegen des Layouts der Fall. Diese Layout-Klassen befinden sich ebenfalls in der AWT.
Es ist auch denkbar, daß einige GUI-Builder die AWT um eigene Klassen ergänzen, beispielsweise eigene Layout-Klassen oder eigene Widget-Klassen. In den derzeitig verfügbaren Entwicklungsumgebungen finden sich eine Vielzahl solcher speziellen Widgets. Das bedeutet, daß Objekte dieser Klassen mitserialisiert werden, sollten sie bei der Zusammensetzung der Oberfläche benutzt worden sein. Da Java Klassen dynamisch lädt (erst zum Zeitpunkt der ersten Verwendung während der Laufzeit, siehe Abschnitt 2), benötigt die virtuelle Maschine beim Einlesen von serialisierten Objekten die Klassen zu diesen Objekten. Dies stellt bei den AWT-Klassen kein Problem dar, da diese in den Standard-Packages enthalten sind. Wurden jedoch zur Oberflächengestaltung GUI-Builder-spezifische Klassen verwendet, müssen diese Klassen zur Laufzeit der virtuellen Maschine zugänglich sein. Das hat zur Folge, daß diese Klassen

mitgeliefert werden müssen, sollte die Applikation außerhalb der Entwicklungsumgebung eingesetzt werden. Diese Problematik ist jedoch kein Spezialfall, der nur bei der Verwendung des Serializer-Beans auftritt. Schon bei der normalen Verwendung eines GUI-Builders müssen in vielen Entwicklungsumgebungen nicht nur die erzeugten und selbstgeschriebenen Klassen als Bestandteil der Applikation mitgeliefert werden, sondern auch eine Sammlung von Klassen der Entwicklungsumgebung.

Als eine weitere Schwierigkeit kann die Weiterentwicklung des Frameworks betrachtet werden. Im Laufe der Entwicklung können Änderungen an den Präsentationsform-Klassen vorgenommen werden. Solche Änderungen können zur Folge haben, daß das serialisierte Objekt einer Oberfläche von einer älteren Version der Klasse stammt. Dies kann zu Problemen beim Einlesen des serialisierten Objektes führen, da die alte und die neue Klassenbeschreibung nicht mehr übereinstimmen. In der Spezifikation der Objektserialisation (siehe [JOS97]) findet sich eine genaue Auflistung sogenannter kompatibler und nicht kompatibler Änderungen an einer Klasse. Zu den kompatiblen Änderungen zählen beispielsweise das Hinzufügen und Herausnehmen von Attributen und eine Veränderung der Modifier von Attributen. Sollten sich also die Änderungen an den PF-Klassen im Rahmen der kompatiblen Änderungen bewegen, kann die Applikation mit dem neuen Framework problemlos die Objekte älterer PF-Klassen einlesen. Sollten sich nicht kompatible Änderungen an den PF-Klassen ereignet haben, ist der Entwickler nicht davor zu bewahren, die Oberfläche neu zu serialisieren. Es ist jedoch zu beachten, daß eine Neu-Serialisation der Resourcen zwar einen Aufwand darstellt, die Resourcen jedoch nicht neu zusammengestellt werden müssen. Im einfachsten Fall kann die Oberfläche im GUI-Builder eingelesen und dort neu serialisiert werden. Anderenfalls kann auf den vom GUI-Builder erzeugten Source-Code zurückgegriffen werden, der aufgeführt werden muß, um das Serializer-Bean betätigen zu können.

Es stellt sich die Frage, wann man die Komponenten serialisieren kann. Praktisch für die Entwicklung wäre es, wenn es schon zur Design-Zeit direkt im GUI-Builder möglich wäre, das Serializer-Bean durch einen Knopfdruck auszulösen und somit die erstellte Oberfläche zu serialisieren. Das setzt jedoch voraus, daß der GUI-Builder zur Design-Zeit mit echten Exemplaren der Oberflächen-Beans umgeht. Dies macht zum Beispiel der Java-Workshop vorbildlich. Leider ist dies nicht die Regel. Andere GUI-Builder erzeugen keine Exemplare der Komponenten, sondern scheinen sich lediglich zu „merken“, welche Klassen angelegt werden müssen, um aus diesen Informationen den Source-Code zu generieren. Bei Verwendung der letztgenannten GUI-Builder gibt es leider nur die Möglichkeit, den vom GUI-Builder erzeugten Source-Code einmal auszuführen und dann den Serializer-Bean-Knopf zu betätigen. Ich denke jedoch, daß auch dieser Aufwand vertretbar ist, zumal dies in der Regel direkt aus den GUI-Buildern geschehen kann (viele besitzen eine „Test“-Funktion).
Damit ist eine Lösung konstruiert, die es ermöglicht, ein Framework zu bauen, welches mit nahezu jedem GUI-Builder zusammenarbeitet, ohne daß es modifiziert werden muß. Man ist jetzt in der Lage, in einem Software-Projekt mit *einem* Framework zu arbeiten, aber durchaus *unterschiedliche* GUI-Builder zu benutzen. Die Oberflächen, die mit den verschiedenen GUI-Buildern erzeugt wurden, können problemlos in einer Applikation verwendet werden, ohne daß zusätzlicher Aufwand nötig wäre.
Allerdings ist die Frage noch ungeklärt, wie man die einmal mit einem GUI-Builder erzeugten Oberflächenkomponenten mit einem anderem GUI-Builder weiterbearbeiten kann. Auch hierzu gibt es eine konzeptionelle Lösung, die im nächsten Abschnitt beschrieben wird.

4.2.2 Das Importer-Bean

Zur Kooperation mit dem Serializer-Bean wurde ein Importer-Bean entwickelt, welches es ermöglicht, einmal mit dem Serializer-Bean serialisierte Oberflächenkomponenten wieder einzulesen. Damit kann man in dem GUI-Builder diese Komponenten wieder einlesen und in die aktuelle Hierarchie eingliedern.
Die Idee dabei ist, im GUI-Builder einen Container anzuordnen, der die vom Serializer-Bean serialiserte Oberfläche aufnimmt. Dazu könnte auf den Container geklickt werden. Um dies zu realisieren erbt das Importer-Bean von der AWT-Klasse `Panel`. Ein Klick aktiviert das Importer-Bean und eine Datei kann auszuwählt werden. Aus dieser Datei ließt das Importer-Bean die Objekte ein (mittels Java-Object-Serialisation), prüft, ob es sich um GUI-Elemente handelt (erben von der AWT-Klasse `Component`) und fügt die Objekthierarchie der Oberfläche im GUI-Builder hinzu. Dazu ruft das Importer-Bean an seiner Oberklasse die Methode „add“, die als Parameter die neuen „child“-Objekte bekommt. Diese Vorgehensweise ist in Abbildung 17ff veranschaulicht.
Konkret bedeutet dies allerdings, daß es bei dem Importer-Bean zwingend erforderlich ist, daß der GUI-Builder mit Exemplaren der JavaBeans schon zur Design-Zeit arbeitet. Dies ist deshalb zwingend erforderlich, damit die eingelesenen Oberflächenkomponenten im GUI-Builder bearbeitet werden können. Arbeitet der GUI-Builder nicht mit Exemplaren der Oberflächenklassen zur Design-Zeit, kann das Importer-Bean nicht aktiviert werden (es existiert noch nicht als Objekt).
Da nicht alle GUI-Builder mit Exemplaren von Oberflächenelementen arbeiten, stellt das Importer-Bean nur eine Teillösung dar. Denkbar ist jedoch, daß zukünftige GUI-Builder dies beherrschen.

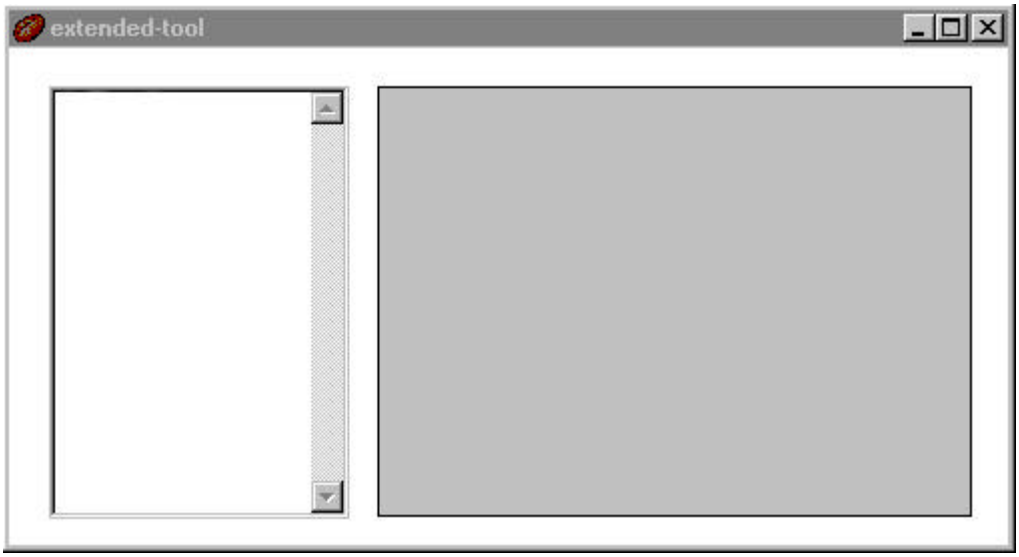


Abbildung 17: Im GUI-Builder gestaltete Oberfläche mit dem Importer-Bean, welches als graue Fläche zu sehen ist.

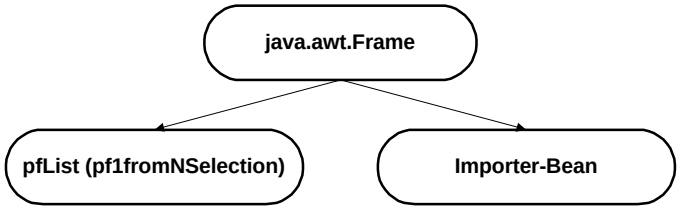


Abbildung 18: Die Objekte, so wie sie vor dem Einlesen im GUI-Builder zusammengestellt wurden.

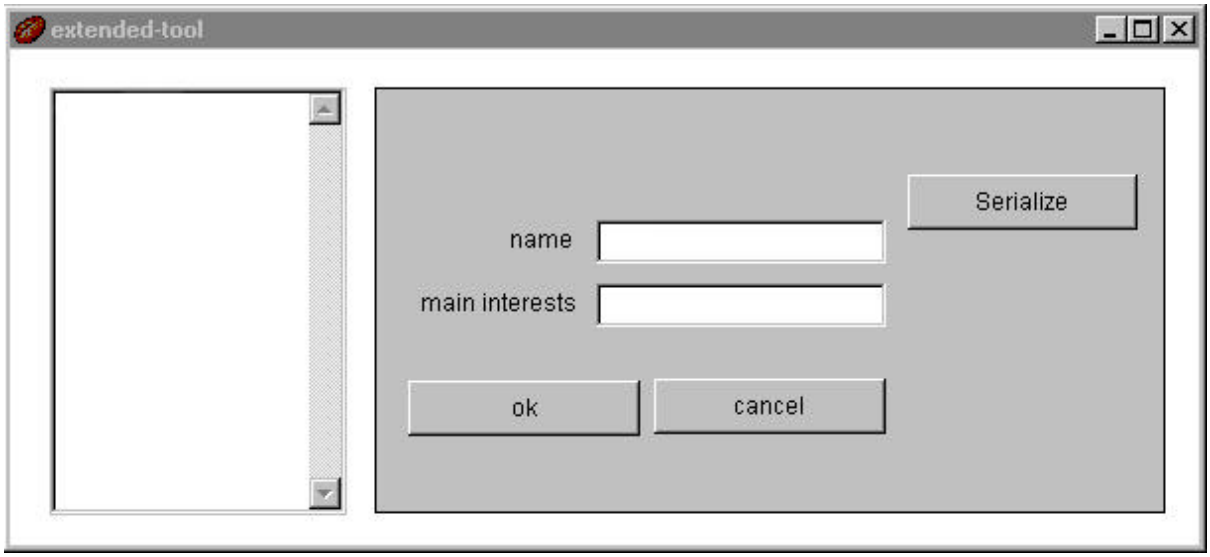


Abbildung 19: Die Oberfläche im GUI-Builder nach dem Einlesen des einfachen Beispiels aus dem vorangegangenen Abschnitt durch das Importer-Bean.

Leider stellt sich selbst bei den GUI-Buildern, die mit Exemplaren der Beans arbeiten, heraus, daß sie ein Einlesen von Komponenten mittels dieses Importer-Beans nicht beherrschen. Das liegt höchstwahrscheinlich darin begründet, daß die GUI-Builder neben der Hierarchie der Oberflächenobjekte eine zweite Struktur verwalten, um aus dieser Struktur die Oberflächenkomponenten zu erzeugen und ggf. den Source-Code zu generieren, sowie die Resource-Files zu schreiben. Spekulativ läßt sich über diese zweite Struktur allerdings wenig sagen.
Daraus resultiert, daß mittels des Importer-Beans eingelesene Komponenten zwar angezeigt werden, sie vom GUI-Builder aber nicht registiert werden (bzw. nicht in die zweite Struktur übertragen werden). Das bedeutet, daß der GUI-Builder daraus keinen Source-Code erzeugen kann, was für unsere Zwecke noch nicht allzu schlimm wäre. Jedoch scheinen die GUI-Builder diese interne Struktur zu benutzen, um die Handhabung mit den zusammengestellten Komponenten zu realsieren. Das bedeutet, daß man diese eingelesenen Objekte nicht mit dem GUI-Builder verändern kann, da sie für ihn praktisch unsichtbar sind.
Insofern sind die Abbildungen 18 und 19 noch illusorisch und sollen lediglich zeigen, wie das Prinzip funktioniert.
Vorstellbar ist, hier eine vorübergehende Lösung zu implementieren. Durch die Benutzung des Serializer-Beans wird sichergestellt, daß auf spätere Entwicklungen reagiert werden kann. Sollte es also möglich werden, das Importer-Bean mit diversen GUI-Buildern einzusetzen, können die erstellten Resourcen wiederverwendet werden.
Sollte schon vor diesem ungewissen Zeitpunkt der Wechsel des GUI-Builders nötig werden[18],

[18] Gerade im sich rasant entwickelnden Bereich Java ist es derzeit nicht vorherzusehen, welche Entwicklungstools in naher Zukunft noch weiterentwickelt und gepflegt werden. Es ist also durchaus denkbar, den GUI-

wäre die Entwicklung eines speziellen Importer-Beans denkbar, welches die vom Serializer-Bean serialisierten Objekte einliest und in das Resource-Format des neuen GUI-Builders transformiert. Dies ist zwar ein relativ hoher Aufwand, da das Resource-Format des neuen GUI-Builders bekannt sein muß und eventuell sehr komplex sein kann, allerdings relativiert sich dieser Aufwand, wenn man die Alternative betrachtet: Alle Resourcen im neuen GUI-Builder neu zusammenstellen.
Der Vorteil dieser Methode ist, daß nicht für jeden GUI-Builder so ein spezielles Importer-Bean geschrieben werden muß, sondern nur im Falle des Wechsels lediglich *ein* spezielles Importer-Bean.

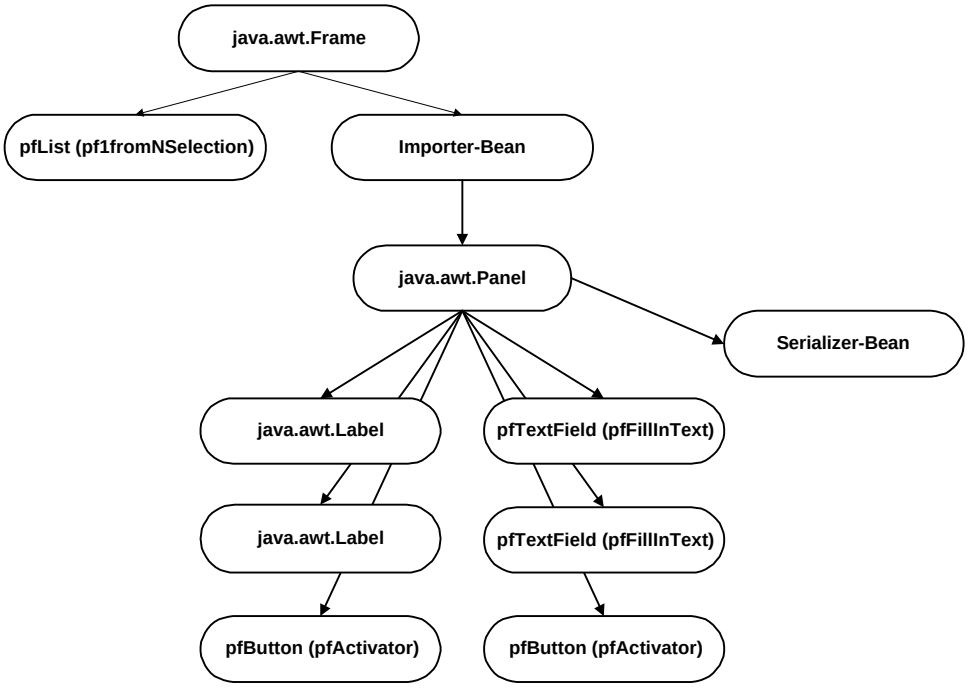


Abbildung 20: Objekthierarchie im GUI-Builder nach dem Einlesen der Komponenten durch das Importer-Bean.

Builder wechseln zu müssen. Auch die enorm schnelle Entwicklung neuer Tools macht einen leichten Umstieg von einem Tool auf ein anderes wünschenswert (siehe Abschnitt 3.4.1).

5 Abschluß

Mit dem in dieser Arbeit vorgestellten GUI-Framework ist die nötige Unabhängigkeit von Entwicklungstool und dem verwendeten GUI-Builder erreicht worden. Damit ist die Möglichkeit geschaffen worden, auch in einem großen Software-Projekt unabhängig vom Entwicklungstool zu sein und sich die Möglichkeit offenzuhalten, einfach und problemlos auf zukünftige Entwicklungen im Bereich der Entwicklungsumgebungen zu reagieren. Zusätzlich zu der Möglichkeit, in einem Projekt eine heterogene Systemlandschaft zu nutzen (durch die Plattformunabhängigkeit von Java), ist es nun möglich, zusätzlich eine heterogene Entwicklungstool-Landschaft zu nutzen.
Die Verwendung der Objekt-Serialisation als Schnittstelle zwischen GUI-Builder und Framework scheint derzeit eine zukunftsweisende Konzeption zu sein. Die Beanbox von SUN zeigt diese Möglichkeit der Objektserialisation von Komponenten und soll laut SUN Techniken für zukünftige Entwicklungen vorschlagen. Es besteht also die Möglichkeit, daß zukünftige Entwicklungsumgebungen sich ebenfalls dieser Technik bedienen. Dies bleibt allerdings abzuwarten. Auch Görtz greift in seiner Implementation (siehe [Görtz97]) auf die Technik der Objektserialisation zurück.
Solange die Objektserialisation noch nicht das Standard-Format für GUI-Builder ist, scheint die Arbeit mit dem Serializer-Bean eine gute Lösung zu sein. Sollte in Zukunft das Serialisieren direkt aus dem GUI.-Builder möglich werden, ist eine Veränderung des Frameworks nicht notwendig. Es setzt das Vorhandensein eines Serializer-Beans nicht zwingend voraus. Das Serializer-Bean ist nur eine Brücke, die eine Verbindung zwischen GUI-Builder und Framework schafft. Sollte diese Lücke von den GUI-Buildern in Zukunft selbst geschlossen werden, braucht das Serializer-Bean nicht mehr verwendet zu werden. Weder eine Veränderung des Frameworks noch eine Anpassung von in der Entwicklung befindlicher Applikationen muß vorgenommen werden.
Für das Importer-Bean ist es sinnvoll, eine breitere Palette von Entwicklungstools zu untersuchen. Da die bis jetzt bekannten Entwicklunsgumgebungen eine Zusammenarbeit mit dem Importer-Bean strikt ablehnen, bleibt nur zu hoffen, daß zukünftige GUI-Builder entweder direkt das Serialisieren und Einlesen serialierter Oberflächenkomponenten beherrschen, oder zumindest dem Importer-Bean es ermöglichen, diese Lücke zu schließen.
Wird diese Möglichkeit auch von zukünftigen GUI-Buildern nicht geschaffen, bleibt immer noch die Lösung, spezielle Importer-Beans zu entwicklen, die aus den serialiserten Objekten das GUI-Builder-spezifische Resourcen-Format generieren und damit die Weiterverwendung bereits erstellter Resourcen mit neuen GUI-Buildern ermöglichen.

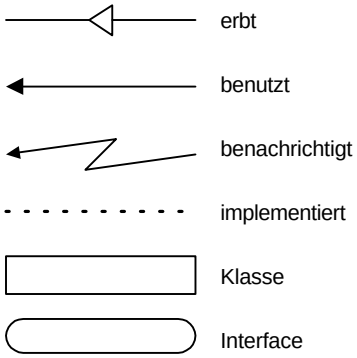
Das GUI-Framework in Java ist in das WAM-Framework in Java eingebettet. Mit diesem WAM-Framework werden in Zukunft am Arbeitsbereich Softwaretechnik erste Erfahrungen gesammelt. Wenn diese Erfahrungen vorliegen, wird das Framework sicherlich verändert und eventuell erweitert werden. Der Praxiseinsatz wird zeigen, wie das Framework zu beurteilen ist. So wird sich zeigen, ob das Zusammenspiel mit den GUI-Buildern wirklich so verwendbar ist, wie es nach ersten Tests erscheint. Ebenso wird sich die Oberflächenanbindung mit Interaktions- und Präsentationsformen einem Praxistest unterziehen müssen. Hier wird die Arbeit von Thorsten Görtz (siehe [Görtz97]) sicherlich weiterführende Fragen dieser Art beantworten. Darüber hinaus bleibt zu beantworten, wie sich Drag & Drop mit Interaktions- und Präsentationsformen realisieren läßt, sowie weitere Fragen nach speziellen Interaktionsformen und der Behandlung von Fachwerten durch solche.
Durch die flexible Struktur des Frameworks sollte es möglich sein, zukünftige Erkenntnisse

aus diesen Bereichen einfach zu integrieren.

6 Anhang

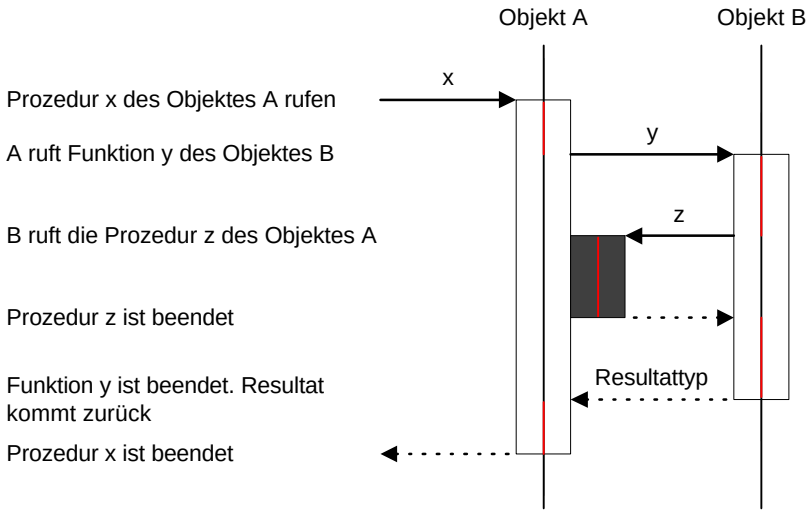
6.1 Legende der Abbildungen

6.1.1 Klassen und Objektdiagramme



Grau dargestellte Klassen und Objekte befinden sich außerhalb der WAM-Konstruktion. Das ist beispielsweise bei Klassen des Toolkits der Fall.

6.1.2 Interaktionsdiagramme



6.2 Übersicht über die Interaktionsformen und Präsentationsformen

Die folgenden Abbildungen zeigen den aktuellen Stand der Implementierung in dem Framework. Die Entwicklung ist keineswegs abgeschlossen. Geplante Änderungen und Erweiterungen sind gekennzeichnet.

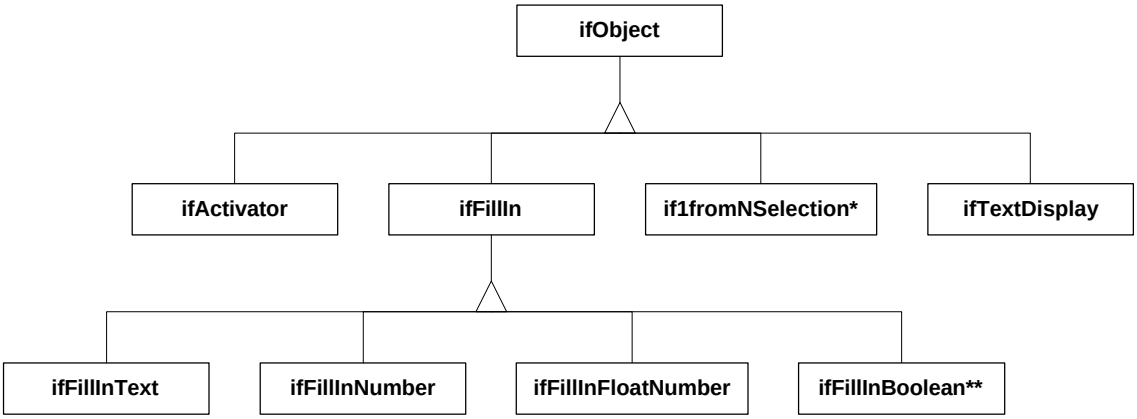


Abbildung 21: Aktueller Stand der Hierarchie der Interaktionsformen.
***) zu der 1fromNSelection werden noch Ergänzungen um Mehrfach-Selektion und statische und dynamische Selectionen hinzukommen.**
****) Der Name "ifFillInBoolean" ist nur eine vorübergehende Benennung.**

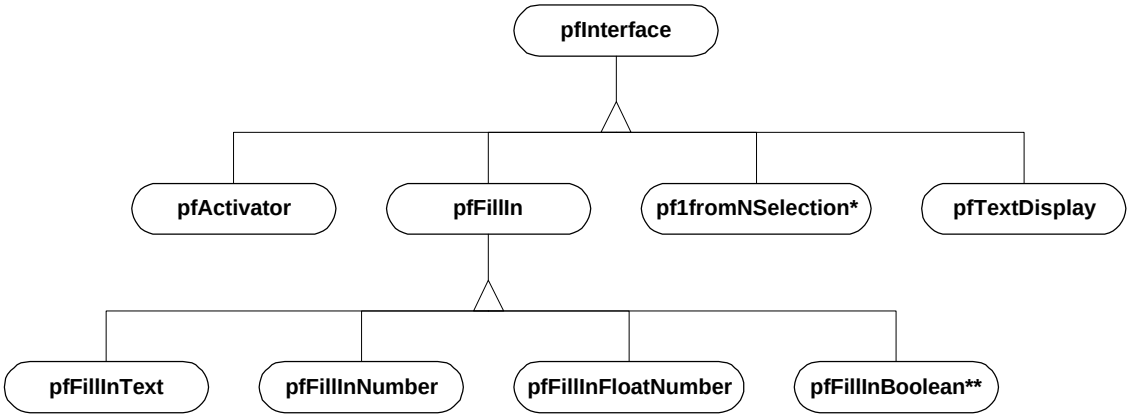


Abbildung 22: Aktueller Stand der Hierarchie der Präsentationsformen.
***) zu der 1fromNSelection werden noch Ergänzungen um Mehrfach-Selektion und statische und dynamische Selectionen hinzukommen.**
****) Der Name "pfFillInBoolean" ist nur eine vorübergehende Benennung.**

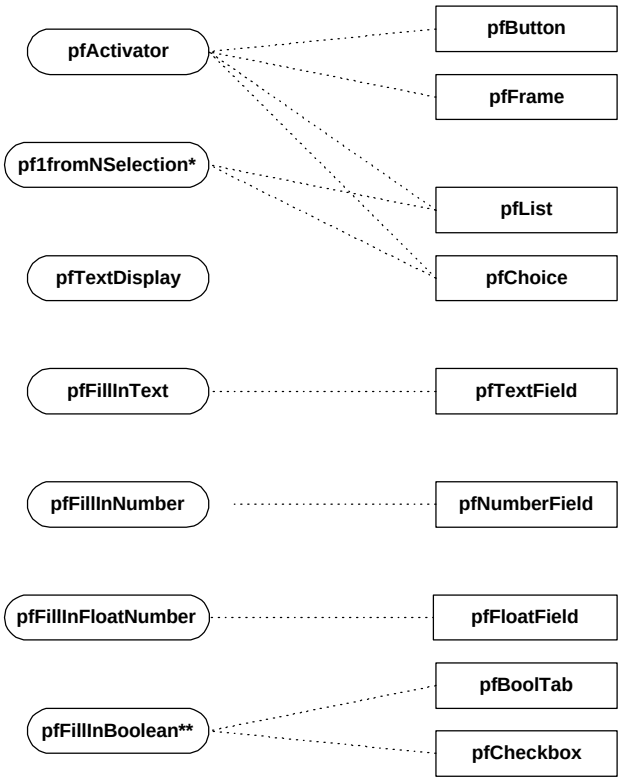


Abbildung 23: Übersicht über die Präsentationsform-Klassen und welche PF-Interfaces sie implementieren. (zu * und ** siehe Abbildung 21)

6.3 Literaturverzeichnis

[ACM97]
„Software Engineering Notes“, Special Interest Group on Software Engineering, Volume 22, Number 4, ACM Press, July 1997

[Apple95]
Apple Computer: „Inside Macintosh“, Apple Computer Inc., Cupertino, 1995 (CD-ROM, published by Addison-Wesley)

[Bohlmann98]
Holger Bohlmann: „*Eine Behälterbibliothek in Java“ (vorläufiger Titel)*, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, *voraussichtlich 1998*

[Dangberg97]
Andreas Dangberg: „Ein interaktives Werkzeug zur Konstruktion von portablen graphischen Benutzungsschnittstellen mit Unterstützung variabler Layout-Algorithmen“, Notizen zu Interaktiven Systemen, Tagungsband zu PB’97: „Prototypen für Benutzungsschnittstellen“, Universität Paderborn, Heft 19, November 1997

[Englanger97]
Robert Englander: „Developing Java Beans“, O’Reilly & Associates, Inc., First Edition, Ju-

ne 1997

[Flanagan96]
David Flanagan: „Java in a Nutshell“, O’Reilly & Associates, Inc., First Edition, 1996 (inzwischen in einer zweiten, überarbeiteten Auflage erschienen)

[Fricke97]
Niels Fricke: *„Der Umgang mit einem Server für WAM-Entwicklungsdokumente“ (vorläufiger Titel)*, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, *voraussichtlich 1998*

[GHJ+96]
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: „Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software“, Addison-Wesley, Bonn 1996

[GJS97]
James Gosling, Bill Joy, Guy Steele: „Java, die Sprachspezifikation“, Addison-Wesley, 1997 (deutsche Übersetzung, Originaltitel: „the Java language spezification“)

[GKZ93]
Guido Gryczan, Klaus Kilberth, Heinz Züllighoven: „Objektorientierte Anwendungsentwicklung“, Vieweg, Braunschweig/Wiesbaden, 1993

[Görtz97]
Thorsten Görtz: „Abstraktion der GUI-Komponente in einem Rahmenwerk“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1997

[JBeans96]
Javasoft: „Java-Beans 1.0 API specification“, Sun Microsystems, 1996

[JCR96]
Javasoft: „Java Core Reflection specification“, Sun Microsystems, 1996

[JOS96]
JavaSoft: „Java Object Serialization specification“, Sun Microsystems, 1996

[Lea97]
Doug Lea: „Concurrent Programming in Java - Design Principles and Patterns“, JavaSoft, Reading, Massachusetts, Addison-Wesley, 1997

[Lewis95]
Ted Lewis (Editor): „Object-oriented application frameworks“, Manning Publications Co., Greenwich, 1995

[Lilienthal95]
Carola Lilienthal: „Konzeption und Realisierung eines an der Anwendungssprache orientierten Hilfesystems nach der Werkzeug-Material Metapher“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1995

[LiWe96]

Carola Lilienthal, Ingrid Wetzel: „Einführung in C++“, Folienunterlagen zum Einführungskurs in die Sprache C++ am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1996

[Meyer90]
Bertrand Meyer: „Objektorientierte Softwareentwicklung“, Prentice-Hall, London, 1990

[PLoP95]
John M. Vlissides, N. Korth, James O. Coplien (Eds.): „Pattern Languages of Program Design“, Addison-Wesley, 1996 (a book publishing the reviewed Proceedings of the Second International Conference on Pattern Languages of Programming, Monticello, Illinois, 1995)

[RW96]
Stefan Roock, Henning Wolf: „Konzeption und Implementierung eines ‘Reaktionsmusters’ für objektorientierte Softwaresysteme“, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1996

[RZ95]
Dirk Riehle, Heinz Züllighoven: „A Pattern Language for Tool Construction and Integration Based on the Tools & Materials Metaphor“ in [PLoP95]

[Sebesta97]
Robert W. Sebesta: „Concepts of Programming Languages“, Third Edition, Addison-Wesley, Reading, Massachusetts, 1997

[Star92]
Star Division: „StarView C++ class library“, StarDivision, 1992

[Strunk97]
Wolfgang Strunk: „Entwicklung einer flexiblen Software-Architektur für interaktive Anwendungssysteme“, Dissertation, Universität Hamburg, 1997 (voraussichtlich)

[Style97]
Niels Fricke, Martin Lippert, Stefan Roock, Henning Wolf: „Java-Styleguides“, http://swt-www.informatik.uni-hamburg.de/~1wolf/java/Styleguides.html

[Taligent95]
Sean Cotter: „Inside Taligent Technology“, with Mike Potel, Addison-Wesley Inc., 1995

[Vlissides95]
John M. Vlissides: „Unidraw: A Framework for Building Domain-Specific Graphical Editor“, in [Lewis95].

[Weiß97]
Ulfert Weiß: „Konzeption und technische Weiterentwicklung eines objektorientierten Frameworks nach der Werkzeug-Material-Metapher“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1997