1 Einleitung

„I have a dream“ heißt ein berühmter Kernsatz aus einer Rede von Martin Luther King. Würde jemals ein Informatiker eine flammende Rede halten (können), so wäre sicher einer seiner Träume der Traum nach Portabilität von Anwendungen. Tatsächlich ist es in der (kurzen) Geschichte der Softwaretechnik nie soweit gekommen, daß Anwendungen problemlos von einem Rechner auf einen anderen übertragen werden können und dort „laufen“.

Auch in jüngster Zeit lassen die Bemühungen nicht nach, hier den Schritt vom Traum zur Wirklichkeit zu gehen. Sowohl auf der technischen als auch auf der methodischen Ebene werden immer wieder neue Ansätze gefunden. Java mit seinem Anspruch der Plattformunabhängigkeit ist hier zu nennen und vor allem die am Arbeitsbereich Softwaretechnik entwickelte WAM-Architektur. Dieser Ansatz zeichnet sich vorrangig dadurch aus, daß der Begriff der „Anwendung“ ersetzt wird durch „Werkzeug“ und „Material“, so daß zunächst der Blick dafür geschärft wird, was überhaupt Gegenstand der Portierung und damit Gegenstand möglicher Änderungen im Code sein kann.

Diese Arbeit versucht, im Kontext des WAM-Ansatzes einen Beitrag dazu zu leisten.

1.1 Fachlicher Kontext

Relativ bald nach Einführung standardisierte Fenstersysteme wie z.B. MacOS, Windows oder Motif wurde versucht, die i.d.R. sehr unübersichtlichen prozeduralen APIs der Fenstersysteme hinter geeigneten Funktions- und Klassenbibliotheken (den sogenannten „GUI-Toolkits“ [1]) zu verbergen. Der nicht unerhebliche Aufwand, um Benutzungsoberflächen zu programmieren, mußte verringert werden.

Danach wurden im nächsten Schritt die mittlerweile äußerst zahlreichen GUI-Frameworks entwickelt. Sie heben z.T. die Grenzen zwischen den Fenstersystemen wieder auf, so daß eine Anwendung mit demselben Programmtext auf zahlreichen Betriebssystemen, für unterschiedliche Fenstersysteme und mit verschiedenen Entwicklungsumgebungen erstellt werden kann.

Die Bezeichnungen „Klassenbibliothek“ und „Framework“ lassen erkennen, daß parallel zu dieser Entwicklung objektorientierte Methoden eingesetzt wurden und zunehmend zum Einsatz kommen. Tatsächlich kann die Objektorientierung bei der Realisierung interaktiver Anwendungen eine maßgeblich Erleichterung mit sich bringen. Umgekehrt wurden auch zahlreiche objektorientierte Entwurfsmuster - man denke nur an das berühmte „Model-View-Controller“ - als Lösungen für Probleme im Bereich der Benutzungsschnittstellen entworfen.

---

[1] Viele Abkürzungen und Begriffe werden im laufenden Text immer wieder benutzt aber nicht erklärt. Statt dessen sind sie im Anhang in einem Glossar aufgeführt und werden dort beschrieben.

Einleitung

GUI-Rahmenwerke werden häufig auch als „Application Frameworks“ bezeichnet, obwohl sie den Software-Entwicklern hauptsächlich nur dazu dienen sollen, Benutzungsschnittstellen zu entwickeln. Zum einen läßt dies schon den Stellenwert dieser Rahmenwerke erkennen, auf der anderen Seite bedeutet dies aber auch, daß sie häufig nicht nur Funktionen für die Programmierung einer Oberfläche zur Verfügung stellen. Vielfach werden auch Klassen angeboten, um andere Teile einer Anwendung zu entwickeln.

Diese Tatsache an sich wäre nicht weiter erwähnenswert, wenn nicht allzuoft die GUI-Klassen mit den fachlichen Klassen vermischt und somit die anwendungsspezifischen Teile mit dem Toolkit an zahlreichen Punkten gekoppelt würden. Oft werden die Klassen des GUI-Frameworks als „Skelett“ der Anwendung benutzt und nur noch um die fachlichen Anteile ergänzt. Dies bringt erhebliche Probleme mit sich, da die Anwendung dadurch von einem einzigen Toolkit extrem abhängig wird, was sich in einem späteren Entwicklungszyklus als Sackgasse herausstellen kann.

Architektur nach WAM

Beim Entwurf einer Anwendung sollte also ein besonderer Wert auf die Trennung der Oberfläche („Interaktion“) und der fachlichen Anteile einer Anwendung („Funktionalität“) gelegt werden. Am Arbeitsbereich Softwaretechnik wird mit der „Werkzeug-Automat-Material-Metapher“ (kurz WAM) solch eine Architektur für die Entwicklung interaktiver (Software-) Werkzeuge propagiert. Dabei wird ein Werkzeug in eine Funktionskomponente (FK) und eine Interaktionskomponente (IAK) unterteilt. Nur die IAK steht mit dem GUI-Toolkit in Verbindung. Dadurch wird es möglich, durch den Austausch der IAK ein neues Toolkit einzusetzen.

In einem weiteren Schritt können sogenannte Interaktionstypen (IAT) benutzt werden, um die Beziehung zwischen der IAK und dem verwendeten Toolkit weiter zu lockern. Die konkreten Widgets - wie z.B. Texteingabefelder oder Buttons - werden dabei hinter einer geeigneten Typhierarchie „versteckt“. Die IAK benutzt deshalb Interaktionstypen, die aus Sicht des Werkzeuges lediglich das reine Verhalten der GUI-Objekte umsetzen sollen. Um eine Auswahl „1 aus n“ z.B. über eine Listbox zu ermöglichen, wird ein IAT „Scrolled List“ eingesetzt. Die eigentliche Listbox-Klasse des Toolkits wird nur intern bei der Implementation des IAT benutzt. Dies bedeutet, daß nach Art eines plattformübergreifenden Toolkits das Fenstersystem hinter den Interaktionstypen verborgen bleibt, und Werkzeuge nur durch die Einbindung einer neuen IAT-Bibliothek portiert werden können.

Im Laufe der Zeit hat sich allerdings gezeigt, daß diese Art der Abstraktion in vielen Fällen nicht konsequent genug ist. Tatsächlich ergibt sich oft nur durch die Benutzung der Interaktionstypen in einer IAK eine zu starke Festlegung, da implizit das verwendete Fenstersystem durchscheinen kann. Letztendlich bringt die Benutzung der Interaktionstypen keinen größeren Vorteil als der Einsatz eines guten Toolkits. Mir erschien es also als notwendig, daß das Konzept der Interaktionstypen weiterentwickelt werden sollte.

Interaktionsformen - Präsentationsformen

Das Hauptproblem ist, daß durch die existierenden Interaktionstypen keine Trennung zwischen abstrakter Interaktion und konkrekter Präsentation gelingt. Es genügt nicht, eine GUI-Klasse `Button` durch eine Klasse `IATButton` zu ersetzen. Tatsächlich ist es aus der Sicht eines Werkzeuges doch nur entscheidend, daß durch das Anklicken eines Buttons eine Aktion ausgelöst wird, die im fachlichen Kontext interpretiert werden kann. Dieses Verhalten des Buttons ist die einzige Eigenschaft, die für die Interaktionskomponente von Bedeutung ist.

Als Lösung für dieses Problem wird in dieser Arbeit eine Verfeinerung der Interaktionstypen vorgestellt, die ich zusammen mit einigen Mitarbeitern und Studenten am Arbeitsbereich Softwaretechnik entworfen habe. Dabei wird als neue Abstraktion das Konzept der Interaktionsform (IAF) und der Präsentationsform (PF) eingeführt. Eine IAK benutzt statt der Interaktionstypen nur noch Interaktionsformen, die tatsächlich nur das reine Interaktionsverhalten eines Werkzeuges berücksichtigen. Statt eines IAT „Button“ benutzt die Interaktionskomponente jetzt eine IAF „Activator“. Diese bietet lediglich die Möglichkeit, der IAK ein Ereignis zu melden, wenn der Benutzer etwas aktivieren möchte. Ob diese Interaktion nun tatsächlich durch einen Button oder durch einen Menüeintrag präsentiert wird, kann die IAK nicht mehr unterscheiden. Dies wird durch den Typ einer zugehörigen Präsentationsform entschieden, der außerhalb eines Werkzeuges definiert wird. Die PF kann mit einer Interaktionsform innerhalb der IAK verbunden werden. Erst die Präsentationsform stellt die Verbindung zum konkreten Widget her und erreicht dadurch, daß das Werkzeug vollständig vom Toolkit entkoppelt wird.

1.2 Übersicht

Im zweiten Kapitel folgt eine allgemeine Einführung in die Domäne der GUI-Toolkits. Dabei werden die verschiedenen Konzepte vorgestellt, die in diesem Bereich umgesetzt werden. U.a. wird auch auf die Konsequenzen hinsichtlich der Anwendungsarchitektur eingegangen, die sich durch die Benutzung der GUI-Frameworks ergeben.

Im dritten Kapitel wird die Architektur nach WAM beschrieben und insbesondere das Architekturmuster der Software-Werkzeuge vorgestellt. Als Beispiel einer konkreten Umsetzung dient das existierende WAM-Framework unseres Arbeitsbereiches, das in C++ entwickelt wurde. Das Hauptaugenmerk liegt dabei auf den Interaktionstypen.

Im Hauptteil dieser Arbeit beschreibe ich im vierten Kapitel detailliert das Konzept der Interaktions- und Präsentationsformen. Im fünften Kapitel wird zusätzlich eine Architektur für die Einbindung graphischer Elemente vorgestellt.

Im sechsten Kapitel werden Teile der praktischen Umsetzung dieser neuen Idee präsentiert, die ich parallel zu dieser Diplomarbeit als Erweiterung des WAM-Frameworks implementiert habe. Zum Ende folgt eine abschließende Betrachtung meiner Arbeit.

2 GUI-Toolkits

2.1 Konzepte und Klassifizierung

In seinem Artikel „User Interface Software Tools“ [Mye95] stellt Brad A. Myers, Leiter des „Human Computer Interaction Institute“ an der Carnegie Mellon University, verschiedene Werkzeuge und Vorgehensweisen für die Entwicklung von Benutzungsschnittstellen vor. Gleich zu Beginn betont er dabei die Wichtigkeit der „User Interface Tools“ für die Entwicklung der Oberflächen. Alleine durch den Einsatz von GUI-Buildern, Klassenbibliotheken oder Rahmenwerken würde die Qualität der Benutzungsschnittstellen zunehmen.

Dieser Vorteil kann nach Myers schon in einem sehr frühen Stadium der Entwicklungsphase erkannt werden, wenn mit geeigneten Werkzeugen nur die Oberfläche einer Anwendung oder zumindest Teile davon als Prototyp präsentiert werden können. Bereits zu diesem Zeitpunkt besteht die Möglichkeit, daß die Entwickler den Anwendern mit dem Prototypen eine gemeinsame Grundlage für den Beginn einer partizipativen Entwicklung bieten. Die späteren Benutzer können so einen ersten Eindruck über die Funktionsweise der Anwendung gewinnen und durch Testen des Prototypen eine erste Rückkoppelung über die Benutzbarkeit geben.

Auf seiten der Entwickler ist es natürlich auch von Bedeutung, daß die Entwicklung der Oberflächen effizienter wird. Daneben ergeben sich aber noch weitere Auswirkungen, die ebenso dazu beitragen können, die Qualität zu steigern. Da wäre zum Beispiel die erhöhte Konsistenz der Benutzungsschnittstellen zu nennen, die mit denselben Werkzeugen erstellt werden. Außerdem entsteht so die Möglichkeit, daß Spezialisten - wie z.B. Graphiker, Industriedesigner oder Psychologen - direkt in den Entwicklungsprozeß eingreifen, ohne selber programmieren zu müssen.

Auf der programmiertechnischen Ebene ergeben sich noch zusätzliche Verbesserungen. Zum einen wird der Umfang des Programmtextes abnehmen, da bereits viele Funktionen in den angebotenen Bibliotheken implementiert werden. Der Programmierer kommt praktisch nicht mehr mit dem API des darunterliegenden Fenstersystems in Berührung, das i.d.R. äußerst umfangreich ist. Zahlreiche Interface Builder erzeugen sogar weite Teile des Quelltextes, der vom Entwickler nur noch angepaßt und mit den fachlichen Anteilen der Anwendung in Verbindung gebracht werden muß. Und schließlich kann es durch die Abstraktion des Fenstersystems auch erheblich einfacher werden, eine Anwendung auf eine andere Plattform zu portieren.

Bei der Entwicklung einer Anwendung für eine bestimmte Zielplattform sollte der Designer einer Oberfläche einige wesentlichen Punkte beachten. Für jedes Fenstersystem - wie z.B. Motif, MS Windows, OS/2 Presentation Manager oder MacOS - existieren sogenannte „Styleguides“. Darin werden Regeln aufgestellt, wie die Oberfläche einer Anwendung für diese Plattform entworfen werden soll. Dadurch soll für alle Anwendungen auf diesem System ein möglichst einheitliches Erscheinungsbild, das sogenannte „Look and Feel“, erzielt

werden. So wird eine möglichst konsistente Bedienung der verschiedenen Programme auf einer Plattform „erzwungen“. Diese Styleguides können z.T. sehr konsequente Forderungen aufstellen (z.B. [App87]). Dabei wird z.B. die Vorgehensweise für die Zusammensetzung von Menüpunkten, dem Layout von Fensterinhalten oder das Verhalten eines Programmes bei der Verarbeitung von Benutzerereignissen vorgeschrieben. Befolgen die Entwickler diese Styleguides, wird den Anwendern die Einarbeitung in ein neues Programm erheblich erleichtert. Die Kehrseite der Medaille kann dann allerdings zum Vorschein kommen, wenn eine Anwendung auf eine andere Plattform portiert werden soll, dessen „Look and Feel“ mit großem Aufwand neu implementiert werden muß.

Die Bedien-Metapher der direkten Manipulation kann als gemeinsame Grundlage für die Implementierung von Benutzungsschnittstellen dienen und wird durch praktisch alle GUI-Toolkits unterstützt. Myers definiert ein Toolkit ganz allgemein als eine Widget-Bibliothek. Dabei kann dem Entwickler auch nur eine prozedurale Programmierschnittstelle angeboten werden. Für bestimmte Anwendungsgebiete existieren spezielle Toolkits. Ganz ausgefeilte Toolkits, die auch die Widgets mehrerer Fenstersysteme abdecken, nennt Myers „Virtual Toolkits“. Dabei können zwei verschiedene Strategien verfolgt werden. Die eine Möglichkeit besteht darin, für jede Plattform das native Toolkit hinter den Schnittstellen des Virtual Toolkits zu kapseln. Für die zweite Alternative werden die Möglichkeiten jedes nativen Fenstersystems in einem großen Virtual Toolkit nachimplementiert. Im ersten Fall können die Entwickler das Look and Feel der jeweiligen Plattform einhalten, haben aber den Nachteil, daß sie im Prinzip nur die Widgets benutzen dürfen, die unter allen unterstützten Fenstersystemen zur Verfügung stehen. Bei der zweiten Strategie wird zwar meistens eine sehr komplette Funktionalität zur Verfügung gestellt, diese Toolkits neigen aber dazu, ihr eigenes Look and Feel zu definieren. Dabei werden auf einigen Plattformen Widgets implementiert, die dort eigentlich nicht dem Styleguide entsprechen. Außerdem werden sie meistens mit einem ziemlich umfangreichen Laufzeitsystem ausgeliefert und verbrauchen damit häufig ein erhebliches Maß an Ressourcen.

Abb. 1 zeigt, wie die Abhängigkeiten der einzelnen Komponenten bestehen, mit denen interaktive Anwendungen implementiert werden. Jede Schicht greift dabei auf Dienste ihrer darunterliegenden Schicht zu. Bei der Benutzung der Begriffe kann es dabei zu Verwirrungen kommen, da auch GUI-Frameworks häufig als „Toolkits“ bezeichnet werden. Außerdem können Bestandteile einer Schicht sich auch gegenseitig benutzen und typischerweise existieren bei der Programmierung von GUIs auch Mechanismen der Rückkoppelung zwischen den Schichten. Dabei werden Ereignisse, die in einer unteren Schicht erkannt werden, einer höheren Schicht gemeldet, ohne daß diese ständig nachfragen muß („Polling“).

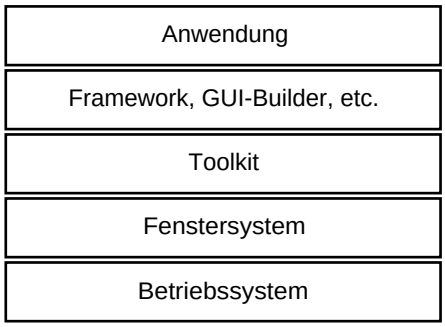


Abb. 1 - Ebenen bei der Benutzung von UI Tools [Mye95]

Nach Myers Definition werden alle Werkzeuge und Bibliotheken, die auf ein Toolkit aufsetzen, als „Higher Level Tools“ bezeichnet. Darunter fallen z.B. Systeme, mit denen man Front-Ends für Datenbankanwendungen entwickeln kann, aber auch Autorenwerkzeuge wie Apples „HyperCard“ (vgl. [Goo90]). Es wurden auch Versuche unternommen, Benutzungsschnittstellen mit eigenen Sprachen oder speziellen Notationen wie z.B. S/T-Netze zu beschreiben. Hier sollen aber nur objektorientierte Rahmenwerke und Interface Builder betrachtet werden.

GUI-Rahmenwerke für die Implementierung von Benutzungsschnittstellen werden von ihren Entwicklern oft auch als „Application Frameworks“ bezeichnet. Tatsächlich geben diese Frameworks die Architektur der Anwendungen vor, die mit ihnen entwickelt werden. Dieser Umstand wird meistens durch die innere Anbindung der Toolkits an die sogenannte „Main Event Loop“ bedingt, die durch das API eines Fenstersystems vorgegeben wird. Diese Schleife ist der zentrale Teil des Rahmenwerkes. Von hier aus werden die eingehenden Ereignisse des Fenstersystems wie z.B. Mausklicks oder Tastatureingaben über einen Dispatch-Mechanismus an die zuständigen Objekte der Anwendung verteilt, so daß die Main Event Loop häufig in den Mittelpunnkt der Architektur gestellt wird. Für einen Anwendungsentwickler kann dies Probleme aufwerfen, da mit Hilfe eines GUI-Toolkits ja nur die Oberfläche seiner Anwendung erstellt werden soll. Gleichzeitig kann aber der Wunsch bestehen, die Architektur der Anwendung selber bestimmen zu wollen. Dem Entwickler wird aber oft nur die Möglichkeit geboten, an speziellen Stellen die anwendungsspezifischen Teile seines Programmes in ein „Skelett“ einzufügen, das vom GUI-Rahmenwerk vorgegeben wird.

Application Frameworks werden für die wichtigsten Plattformen oft schon vom Hersteller des Fenstersystems angeboten, um die Entwicklung von Anwendungen für diese Plattform zu unterstützen. Für Microsoft Windows sind dies z.B. die „Microsoft Foundation Classes“, kurz „MFC“ [Mic93]. Für die Macintosh-Plattform existieren mit „MacApp“ [Wil90] und „PowerPlant“ [Met96] gleich zwei konkurrierende Rahmenwerke. Daneben existieren zahlreiche weitere Rahmenwerke, die zum einen den Anspruch der „Virtual Toolkits“ gerecht werden wollen, zum anderen die Entwicklung von Software für spezielle Domänen wie z.B. graphische Editoren unterstützen sollen.

Nach Myers Definition stehen Interface Builder gleichberechtigt neben den GUI-Frameworks als „Higher Level Tools“, die auf ein Toolkit zurückgreifen. Tatsächlich ist es aber mittlerweile so, daß die Interface Builder praktisch immer in Verbindung mit einem Rahmenwerk betrachtet werden müssen, da sie i.d.R. auf ein bestimmtes Rahmenwerk ausgerichtet sind. Die Definition der Benutzungsschnittstelle eines Programmes wird dabei interaktiv in einem graphischen Editor durchgeführt. Dabei kann die spätere Oberfläche sofort betrachtet werden, z.T. können die Interface Builder mit gewissen Einschränkungen sogar das Verhalten der Oberfläche simulieren. Die Anbindung an die spätere Anwendung erfolgt meistens dadurch, daß der Interface Builder Quelltext erzeugt, mit dem die Oberfläche des Programmes so generiert werden kann, wie sie im Builder definiert wurde. In einem anderen Ansatz, der erstmals vom Interface Builder für das NeXT Betriebsystem eingeführt wurde [NeX91], werden die Objekte, die im Builder für die Oberfläche definiert wurden, abgespeichert und von der Anwendung zur Laufzeit wieder gelesen.

Auflistungen exisitierender GUI-Toolkits, Frameworks und Interface Builder sind im Internet verfügbar. Eine sehr ausführliche Aufstellung ist z.B. die Zusammenfassung von Myers [Mye96].

2.2 Konkrete GUI-Toolkits

Aus der fast unüberschaubaren Menge von GUI-Toolkits sollen hier exemplarisch zwei Rahmenwerke vorgestellt werden, die die typischen Merkmale der zwei wesentlichen Arten von Toolkits besitzen. Das erste - „MacApp“ von der Firma Apple - ist praktisch der Stammvater aller GUI-Frameworks. Es steht stellvertretend für die Rahmenwerke, die die komplette Anwendungsstruktur vorgeben und deren Klassen in der Art eines „White Box Frameworks“ mit Hilfe der Vererbungstechnik spezialisiert werden müssen[2] . MacApp steht nur für das Fenstersystem des Apple Macintosh zur Verfügung. Danach wird das Toolkit „Amulet“ vorgestellt. Dieses kann in drei wesentlichen Punkten konträr zu MacApp betrachtet werden. Es ist das Resultat eines akademischen Projektes und verfolgt deshalb einige ungewöhnliche Ansätze, die in kommerziellen Projekten aus Gründen der Marktakzeptanz wahrscheinlich eher selten oder gar nicht anzutreffen sind. Seine Architektur gibt nicht unbedingt die Struktur der Anwendung vor und verfolgt den Ansatz eines „Black Box Frameworks“. D.h. die angebotenen Klassen werden i.d.R. nur benutzt und nicht als Basisklassen für eine Vererbungsbeziehung gesehen. Der dritte Unterschied zu MacApp ist, daß Amulet für verschiedene Fenstersystem angeboten wird und damit den interessanten Aspekt der Plattformunabhängigkeit verfolgt.

[2] Detaillierte Ausführungen über Konzepte objektorientierter Rahmenwerke bietet z.B. [Wei97]

Kapitel 2

MacApp

„MacApp is perhaps the quintessential example of a quality object-oriented framework. First available in early 1985 in the language Object Pascal, MacApp represented one of the first commercially available object-oriented application frameworks. Its domain was quite general, intended to aid the creation of all applications that conformed to the Macintosh look and feel... A relatively stable version was finally release[d] in 1992. By this time, the library was much more approachable, understandable, and adaptable. No longer presented as just a sea of classes, the library's architecture was made explicit, and key collaborations were well-documented.“

Grady Booch

Das Application Framework „MacApp“ der Firma Apple ist das am längsten kommerziell verfügbare Rahmenwerk überhaupt. Es gilt für viele als das große Vorbild, das bei der Entwicklung zahlreicher anderer Rahmenwerke als Maßstab und Ausgangspunkt für eigene Konstruktionen diente. Häufig wurde in Seminaren die Benutzung MacApps im Zusammenhang mit einer grundlegenden Einführung in die objektorientierte Entwicklung gelehrt [Wilson90]. Mittlerweile wird es nur noch für die Programmiersprache C++ angeboten.

MacApp unterstützt mit dem MacOS nur eine GUI-Plattform und unterliegt damit nicht unbedingt den Anforderungen und Einschränkungen eines fenstersystemübergreifenden GUI-Toolkits. Im Gegenteil: da MacApp in seiner Evolution als Framework für die Programmierung des Apple Macintosh gewachsen ist, scheinen selbst in der neuesten Version an manchen Stellen noch Details des MacOS-API durch. Am auffälligsten ist dies bei der Speicherverwaltung, die sich eng an das Betriebssystem anlehnt. In ganz speziellen Fällen muß man sich sogar Gedanken über die Belegung des Speichers bei alten Macintosh-Rechnern machen, die noch über einen Motorola 680x0 Prozessor verfügen.

Eine typische MacApp-Anwendung folgt der Dokumenten-Metapher des Apple Macintosh. Die zwei Schlüssel-Abstraktionen des Rahmenwerkes sind *Applikation* und *Dokument*. Die Anwendung stellt dem Benutzer Funktionen zur Verfügung, um Daten, die in Form eines Dokumentes organisiert werden, zu editieren. Für ein Zeichenprogramm muß z.B. eine Klasse „Drawing Application“ von der Basisklasse `TApplication` abgeleitet werden. Die Dokumente dieses Zeichenprogrammes werden mit der Klasse „Drawing Document“ implementiert, die eine Subklasse von `TDocument` ist. Das Applikationsobjekt erzeugt die Objekte der spezialisierten Dokument-Klassen und verwaltet intern deren Zustand, um zum Beispiel beim Beenden der Applikation oder vor dem Schließen eines Dokumentes automatisch den Benutzer aufzufordern, das Dokument zu sichern.

Die Klassen MacApps lassen sich in vier große Bereiche einordnen: Entwicklung des GUIs, Umgang mit Dokumenten, Unterstützung des MacOS und technische Hilfsklassen. Damit stellt MacApp ein typisches Beispiel für ein Framework dar, das sowohl die Klassen für die Entwicklung der Benutzungsschnittstelle als auch für die Entwicklung fachlicher Klassen zur Verfügung stellt und in der Benutzung auch vermischt.

Die beiden anderen wichtigen Basisklassen neben `TApplication` und `TDocument` sind `TCommand` und `TView`, die um anwendungsspezifische Eigenschaften erweitert werden müssen. Dazu werden bestimmte Einschubmethoden redefiniert, die zur Laufzeit aus dem Rahmenwerk heraus aufgerufen werden[3].

Das Applikationsobjekt existiert in einer Anwendung nach dem Muster des „Singleton“ (vgl. [GHJ+94]) nur einmal. Es initialisiert die gesamte Anwendung und behält praktisch während der gesamten Laufzeit den Kontrollfluß. Die Hauptereignisschleife des GUI wird intern implementiert. Soll auf spezielle Ereignisse außerhalb des Singleton reagiert werden, so müssen dafür eigene Event-Handler definiert werden.

Damit wird die gesamte Anwendungsstruktur durch das Rahmenwerk vorgegeben, so daß die Routine „main“ praktisch nur aus der Erzeugung des Wurzelobjektes und der Übergabe des Kontrollflusses an MacApp besteht. Dieser wird vollständig durch das Applikationsobjekt gesteuert; in Einschubmethoden werden weitere Objekte erzeugt oder Entscheidungen getroffen, die MacApp nicht vorsehen kann.

```
void main( )
{
  [...]

  TApplication* anApplication = new TDrawingApplication( );

  anApplication -> Run( );

  delete anApplication;
}
```

Abb. 2 - Funktion „main“ einer MacApp-Applikation

Ein Ereignis, das durch den Benutzer ausgelöst wurde und Änderungen im Dokument bewirken könnte, wird nicht direkt in einen Methodenaufruf umgesetzt. Statt dessen wird über einen speziellen Dispatcher-Mechanismus ein Kommando-Objekt gesucht und dessen Methode `DoIt` aufgerufen. Der Typ des Kommando-Objektes ist eine von `TCommand` abgeleitete Klasse. `DoIt` ist eine Einschubmethode, die in der spezialisierten Kommando-Klasse die Aktion, die dem Ereignis zugeordnet ist, ausführt. Der große Vorteil dieser Indirektion ist, daß über zwei weitere Einschubmethoden `UndoIt` und `RedoIt` mit relativ einfachen Mitteln ein mächtiger Undo-Mechanismus implementiert wird.

[3] Sie verwirklichen damit das sog. „Hollywood-Prinzip“: „Don´t call us, we call you!“

Die vierte wichtige Basisklasse ist `TView`. Alle Objekte, die in einem Fenster dargestellt werden sollen, werden über ein zugehöriges View-Objekt visualisiert. View-Objekte können ineinander verschachtelt werden, wobei die Wurzel einer View-Hierarchie immer eine Fenster-View ist. Auch für Kontrollelemente (z.B. Button oder Texteingabefelder) werden Klassen von `TView` abgeleitet („TButtonView“ oder „TTexteditView“). Ereignisse, die über View-Objekte hereinkommen, werden in der Einschubmethode `DoEvent` verarbeitet. Alternativ kann dies auch an die Super-View delegiert werden.

Views, die den Inhalt eines Dokumentes darstellen sollen, werden vom Dokument-Objekt erzeugt. Hierfür muß die Einschubmethode `DoMakeViews` der Klasse `TDocument` redefiniert werden. In dieser Methode kann auf ein globales View-Server-Objekt zurückgegriffen werden, das View-Objekte mittels einer Ressourcenbeschreibung erzeugt, die vorher in MacApps Interface Builder definiert und abgespeichert wurden. Die graphische Darstellung einer View wird in der Einschubmethode `Draw` implementiert. Für jede View kann dabei ein Objekt vom Typ `TDrawingEnvironment` definiert werden. Über dieses Objekt können graphische Attribute - wie z.B. Farben, Linienstärke oder Textattribute - eingestellt werden. Die Methode `Draw` implementiert so nicht nur die Bildschirmausgabe, sondern dient gleichzeitig auch für die Ausgabe auf einem Drucker. Statt eine View zu spezialisieren, kann das Zeichnen auch über ein spezialisiertes Drawing-Environment-Objekt erfolgen. Dies hat den Vorteil, daß so nur noch die Standard-View-Klassen benutzt werden. Im Interface Builder kann dann angeben werden, welches Drawing-Environment-Objekt für eine View benutzt werden soll. Das wird dann zur Laufzeit per „Später Erzeugung“ für das View-Objekt angelegt und mit diesem verbunden.

Als Beispiel wird mit MacApp der Quelltext für einen einfachen Icon-Editor mitgeliefert. Für dessen Implementation werden eine Applikations-Klasse, eine Dokument-Klasse, eine Klasse für die Icon-Bitmap eines Dokumentes, die View-Klasse für das Dokument und sieben Kommando-Klassen benötigt. Es läßt sich gut erkennen, wie die Aufgaben zwischen Rahmenwerk und spezialisierten Klassen aufgeteilt werden und dabei die Vorgaben des Styleguides ([App87]) für das Look and Feel des Macintosh eingehalten werden.

Für die Menüzeile einer Mac-Applikation werden bestimmte standardisierte Menüeinträge gefordert (siehe Abb. 3). Es müssen zumindest die Menüpunkte „File“ und „Edit“ (bzw. „Ablage“ und „Bearbeiten“) angeboten werden. Die Menüpunkte unter „File“ werden komplett von MacApp abgehandelt. Durch die Einschubmethoden der Applikations- und der Dokumentenklasse können durch gezielte Methodenaufrufe aus MacApp heraus Dokument-Objekte erzeugt werden, die Daten eines Dokumentes aus einem Datenstrom gelesen bzw. in diesen geschrieben werden und durch den Aufruf der Methode `DoMakeViews` die View-Objekte für die Darstellung des Dokumentes am Bildschirm bzw. für das Ausdrucken angelegt werden.

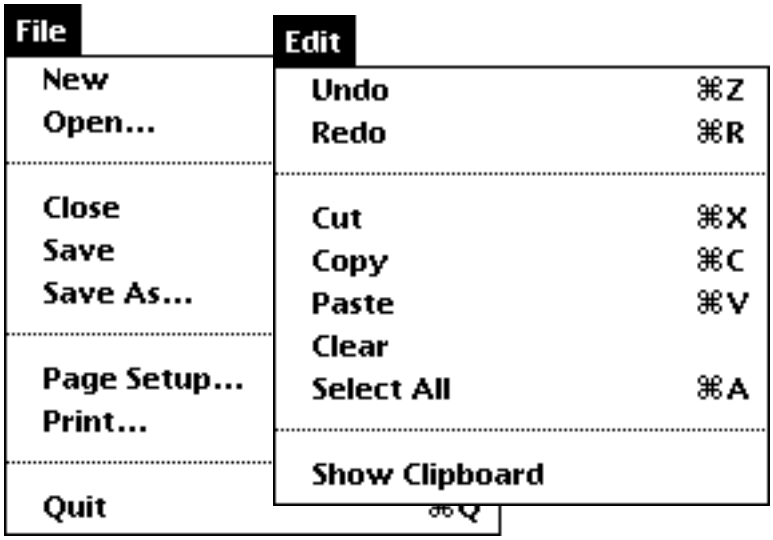


Abb. 3 - Standardeinträge einer Menüleiste unter MacOS

Unter dem Menüeintrag „Edit“ verwaltet MacApp die Menüpunkte „Undo“, „Redo“ und „Show Clipboard“. Nachdem MacApp die Kommando-Objekte übergeben worden sind, die die verschiedenen Funktionen der Anwendung implementieren, werden die Referenzen auf die Kommando-Objekte in einer verketteten Liste gespeichert. Dabei wird wieder die Methode `DoIt` aufgerufen. Die Auswahl des Menüpunktes „Undo“ bewirkt, daß die Methode `UndoIt` des letzten Objektes in der Liste aufgerufen wird. Danach wird ein Cursor auf das vorletzte Objekt gesetzt, so daß für das nächste „Undo“ die Methode `UndoIt` dieses Objektes aufgerufen wird. Für das „Redo“ verläuft dieser Prozeß umgekehrt, und der Cursor wandert wieder an das Ende der Liste. Die anderen Menüpunkte werden von den View-Objekten abgehandelt, die sich im aktiven Fenster befinden.

Die Entwicklung der View-Klassen wird durch einen Interface Builder - im Sprachgebrauch von MacApp „View Editor“ genannt - unterstützt. Dieser View-Editor bietet die Möglichkeit, die komplette View-Hierarchie einer Anwendung in der Art eines Zeichenprogrammes interaktiv zu erstellen. Dazu wird dem Entwickler in einer Palette eine Auswahl von Standard-Views angeboten, mit denen die Benutzungsoberfläche der Anwendung komplett erstellt werden kann.

Somit kann das Aussehen und das Verhalten der Standard-Views (z.B. die Views für Buttons, Textfelder, Scrollbars etc.) komplett in die eigene Anwendung übernommen werden, ohne eine einzige Zeile in C++ schreiben zu müssen. Der View-Editor sichert die Beschreibung der Views im Ressourcen-Format des MacOS. In der Entwicklungsumgebung wird diese Ressource-Datei später zur Anwendung dazugelinkt.

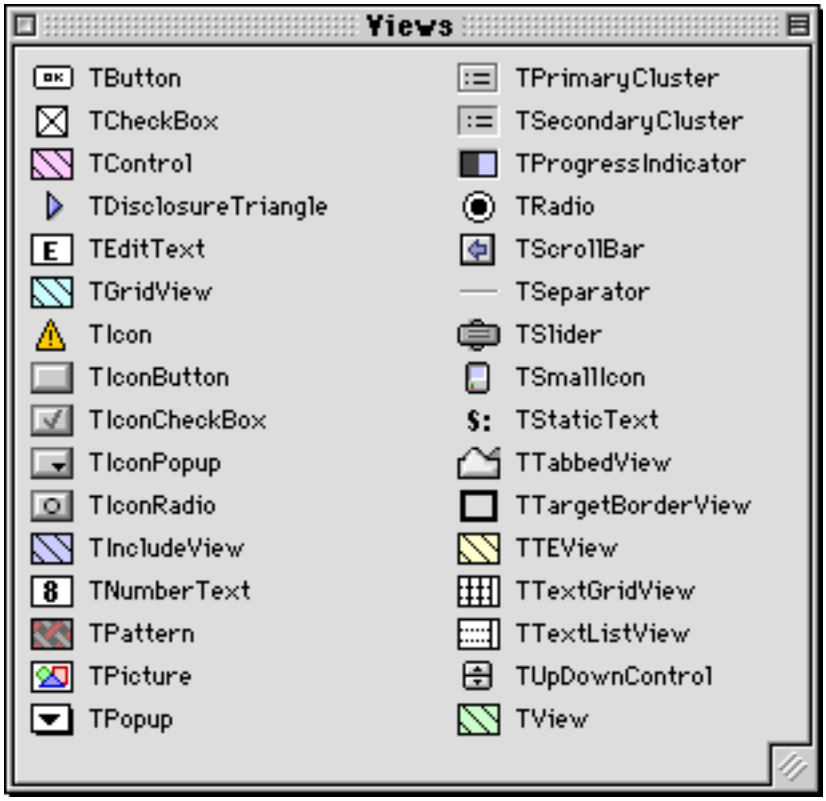


Abb. 4 - View-Palette des MacApp View Editor

Darüber hinaus ist es möglich, anwendungsspezifische View-Klassen schon im View-Editor zu definieren. Dabei wird ein sogenanntes „View-Template“ definiert, für das eine Standard-View als Basisklasse angegeben werden muß und dem man eine Liste mit Klassenattributen zuordnen kann. Jedem Attribut muß ein Typ, der sich aus der Menge der C++ Basistypen (z.B. „int“, „long“, „float“) ergibt, zugeordnet werden. Diese Attribute können dann im View-Editor genau wie die Attribute der View-Basisklasse verändert werden. Für sehr spezielle View-Klassen bleibt aber nur die Klasse `TView` als Basisklasse. Dabei kann es dann sehr aufwendig werden, die Liste der Attribute im View-Editor zu definieren.

Für jede im View-Editor definierte Klasse muß in der Anwendung eine entsprechende Klasse in C++ deklariert werden. Für die Standard-Views ist dies natürlich schon in MacApp geschehen. Für die anwendungsspezifischen Klassen hat der Entwickler die Klassen selber zu schreiben. Eine View-Klasse kann zur Laufzeit bei einem globalen View-Server-Objekt registriert werden. So kann ein Objekt dieser View-Klasse in der Art des Factory-Musters (siehe [GHJ+94]) erzeugt werden, ohne den Typ des konkreten View-Objektes zu kennen. Sind alle View-Klassen einer View-Hierarchie auf diesem Wege bekannt, kann der View-Server die View-Ressourcen, die im View-Editor gespeichert wurden, auswerten und die View-Hierarchie komplett erzeugen. Dabei werden die Werte der im View-Editor definierten Attribute berücksichtigt und bei der Erzeugung gesetzt.

Amulet

Das Amulet Toolkit ist ein portables GUI-Toolkit, mit denen Benutzungsoberflächen erstellt werden können, die der Metapher der direkten Manipulation entsprechen. Es wurde in der Sprache C++ entwickelt und unterstützt die Zielplattformen Unix (mit X Windows), Microsoft Windows und MacOS. Amulet („Automatic Manufacture of Usable and Learnable Editors and Toolkits“) ist der direkte Nachfolger von „Garnet“, einem im akademischen Umfeld vielbeachteten GUI-Toolkit, das in Common Lisp entwickelt wurde [Mye90]. Die Entwickler von „Garnet“ und „Amulet“ sind Mitarbeiter und Studenten am „Human Computer Interaction Institute“ von Brad A. Myers.

Aufgrund des akademischen Hintergrundes verfolgten Amulets Entwickler ein anderes Ziel als vergleichbare kommerzielle Anbieter. Amulet soll ein möglichst „komplettes“, aber flexibles und den Ergebnissen der neuesten Forschungserkenntnisse entsprechendes Toolkit darstellen. Trotzdem wirken die Programme, die mit dem Amulet entwickelt wurden, etwas hölzern und entsprechen nicht ganz dem Aussehen und Erscheinungsbild professioneller Programme.

Amulet besteht aus sieben Sub-Frameworks, die die folgenden Funktionalitäten umsetzen:

ORE - „Object Registering and Encoding“, Stellt in C++ ein neues, prototypenbasiertes Typkonzept zur Verfügung, in dem alle Typinformationen erst zur Laufzeit ausgewertet werden.

Constraints - Unterstützt die Definition von Objektabhängigkeiten, um eine Konsistenz der abhängigen Daten zu realisieren. Ähnelt dem Beobachter-Muster.

Commands - Callback Mechanismus, um Aktionen durchführen und wie in MacApp per Undo rückgängig machen zu können.

Interactors - Anbindung der GUI-Ereignisse an die anwendungsspezifischen Objekte.

Widgets - stellt standardisierte GUI-Objekte (z.B. Fenster, Buttons, Checkboxes oder Texteingabefelder) zur Verfügung.

GEM - Low Level Graphik-System, um plattformunabhängige Graphikausgaben zu realisieren.

Opal Graphics System - High Level Graphik-System, basierend auf dem GEM-Framework. Bietet Klassen, um anspruchsvolle graphische Editoren zu entwickeln.

In der neuesten Version 3.0, die im letzten Jahr veröffentlicht wurde, ist als neuestes Feature u.a. die Unterstützung von Animation hinzugekommen. Außerdem exisitiert mit „Gilt“ ein Interface Builder, der den Gebrauch des Sub-Frameworks „Widgets“ unterstützt. Für die Zukunft sind Erweiterungen in Richtung Multi-Media, Spracherkennung, Unterstützung von 3D-Graphik, Visualisierung, Unterstützung des WWW und Unterstützung von CSCW-Konzepten geplant (vgl. [Mye97]).

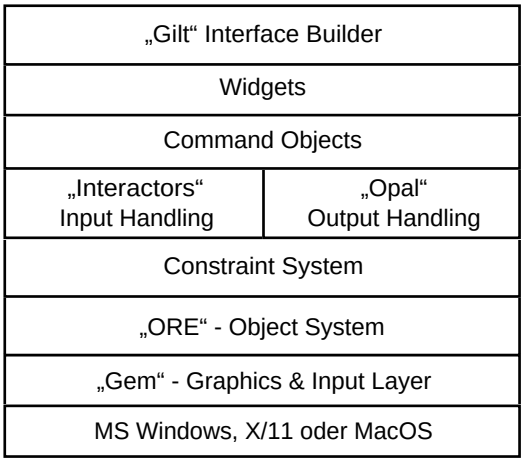


Abb. 5 - Die Schichten des Amulet Toolkits

Bei der Benutzung des Amulet Toolkits fällt einem sofort das eigene Typsystem des Toolkits auf, das auf C++ aufgesetzt wurde und nach dem Prinzip eines „Prototype-Instance Object System“ entworfen wurde. Sämtliche Klassen des Rahmenwerkes werden in diesem Typsystem definiert, da die Entwickler der Ansicht waren, daß das Laufzeitsystem von C++ nicht flexibel genug sei . Das ORE-Objektsystem unterscheidet sich in wesentlichen Punkten von C++ mit Merkmalen, wie z.B. eine automatische Speicherverwaltung, die automatische Benachrichtigung bei Wertänderungen oder die Verwaltung von „Attribute Value Pairs“.

Im Unterschied zu Rahmenwerken, die nach dem Vorbild MacApps als Application Framework die Architektur einer Anwendung vorgeben, kann ein Entwickler bei der Benutzung Amulets die Struktur seines Programmes weitestgehend selbst bestimmen. Amulet bietet im Prinzip nur die Möglichkeit, Objekte in der Art eines Black Box Frameworks zu benutzen. Dabei kann eine weitere Besonderheit des Amulet-Laufzeitsystems ausgenutzt werden, indem die Möglichkeit der automatischen Objekt-Aggregation eingesetzt wird. Im Manual wird diese Art der Programmierung als „Deklarative Programmierung“ bezeichnet:

Note that the Amulet code has a declarative programming feel to it, instead of standard imperative programming. Instead of giving Amulet step by step instructions on how to draw things, such as, „first draw a window, then draw a string,“ and so on, you tell Amulet, „Create a window. Create some text, and add it to the window. Go!“ The programmer never calls „draw“ or „erase“ methods on objects. Once you set up the graphical world how you want it, and „Go“ (start the main event loop), Opal automatically draws and erases the right things at the right time.

Zur Anschauung werden auch mit dem Amulet-Toolkit einige Beispielprogramme zur Verfügung gestellt. Dazu gehört ein kleines Simulationsprogramm für logische Schaltungen, das sehr schön die Möglichkeiten Amulets demonstriert. Z.B. können die Ein- und Ausgänge der Schaltelemente mit Leiterbahnen verbunden werden, die nach dem Verschieben der Elemente ihre Lage ebenfalls verändern. Dies wird mit dem Möglichkeiten des Sub-Frame-

works „Constraints“ realisiert. Die Architektur des Programmes vermischt die fachlichen Teile der Anwendung mit der Implementation der Benutzungsschnittstelle. Das Wurzelobjekt der Anwendung ist das Fensterobjekt, dem alle anderen Objekte hinzugefügt werden, u.a. eine Werkzeugpalette, die wieder aus graphischen Objekten besteht. Diese erfüllen ein Kommandoprotokoll und können dabei die fachlichen Objekte verändern.

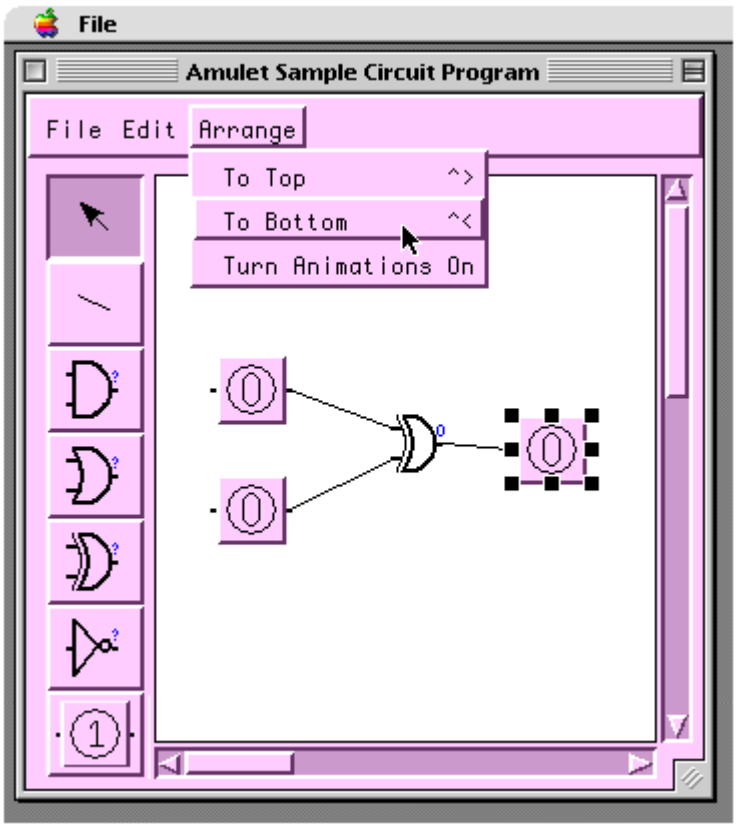


Abb. 6 - Ein Beispielprogramm des Amulet-Toolkits

Eine Besonderheit des Toolkits ist es, daß auf jeder der unterstützten Plattformen das Look and Feel einer anderen Plattform angeboten werden kann. In Abb. 6 sieht man das Simulatorprogramm, das auf einem Macintosh im Aussehen eines Motif-Programmes läuft.

Zuletzt soll noch einmal die Unterstützung durch den Interface Builder „Gilt“ erwähnt werden. Für die Anbindung an das Framework wurde dabei ein anderer Ansatz als bei MacApps View-Editor gewählt. Für die zusammengestellten Widgets generiert Gilt C++ Quelltext, mit dem die Oberfläche so implementiert wird, wie sie im Interface Builder zusammengestellt wurde. Bei jeder Änderung muß der Quelltext neu kompiliert und mit dem Rest des Programmes gebunden werden.

2.3 Kritische Betrachtung

Ausgefeilte GUI-Toolkits wie MacApp[4] bieten nicht nur Klassen für die Entwicklung von Benutzungsoberflächen, sondern geben praktisch die komplette Struktur einer Anwendung vor. Soll statt dessen eine eigene Architektur umgesetzt werden, arbeitet man praktisch gegen das Framework. Mit Maßgaben, z.B. als Rahmenwerke für dokumentenzentrierte Anwendungen zu dienen, werden die Einsatzmöglichkeiten zusätzlich begrenzt.

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

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

Amulet stellt den Gegensatz zu solchen Rahmenwerken dar, die der Architektur MacApps als Application-Framework nachempfunden sind. Neben vielen einzigartigen Eigenschaften wie z.B. dem Objekt-System lassen sich auch einige Merkmale erkennen, die nicht nur typisch für solch ein „Black Box“ Framework sind.

Da ist vor allem die Abstraktion des Fenstersystemes zu nennen, auch wenn sie in diesem Fall relativ unprofessionell implementiert wurde. Zusammen mit der Möglichkeit, die Struktur der Anwendungen relativ frei zu bestimmen, kann so eine relativ gute Entkoppelung vom Fenstersystem und vom Rahmenwerk erreicht werden.

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

---

[4] Vergleichbare kommerzielle Toolkits sind die „Microsoft Foundation Classes“ [Mic93] oder Metrowerks „PowerPlant“ [Met96]; Beispiele für freie Toolkits sind „ET++“ [WG95] oder „WxWindows“ [Sma97].

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

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

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

Die drei Forderungen lassen erkennen, daß es das „perfekte“ GUI-Framework wohl nicht geben kann. Wie Kilberth et al. in [KGZ94] schreiben, wird heute niemand mehr ohne größere Not ein eigenes Fenstersystem für die Implementation interaktiver Anwendungen programmieren. Das läßt sich sicherlich auch auf die GUI-Rahmenwerke übertragen. Es gilt also, einen „defensiven“ Ansatz zu finden, bei dem die Nachteile der Toolkits in einem gewissen Maße in Kauf genommen werden. Die fachlichen Anteile einer Anwendung sollten dabei vom GUI-Toolkit möglichst vollständig entkoppelt werden. So könnte das Toolkit eingesetzt werden, das den gegebenen (oder sich ändernden) Anforderungen am ehesten entspricht und vor allem mit geringem Aufwand auch gegen ein anderes ausgetauscht werden kann.

3 GUIs und WAM

3.1 Trennung von Funktion und Interaktion

Unter dem Stichwort „Von Softwareroutinen zu reaktiven Anwendungssystemen“ beschreiben die Autoren des Buches „Objektorientierte Anwendungsentwicklung“ [KGZ94], wie frühere Systeme oft mit einem hohen Maß an Ablaufsteuerung konstruiert wurden. Charakteristisch für diese Systeme waren Benutzungsschnittstellen, die durchgängig als Abfolge von Menüs und Bildschirmmasken aufgebaut worden sind und ausschließlich mit der Tastatur bedient werden konnten. Nach Kilberth et al. wurden in einigen Organisationen bewährte Abläufe qualifizierter Tätigkeiten nur deshalb geändert, weil diese Tätigkeiten durch Software unterstützt werden sollten und diese ausschließlich mit einer schematischen Ablaufsteuerung implementiert werden konnte. Eine aufgezwungene Reihenfolge bei der Erledigung komplexer, qualifizierter Aufgaben führt aber zwangsläufig zu Problemen, die im wesentlichen nur auf die Bedienung des Anwendungssystems zurückzuführen sind.

Eine Lösung für dieses Problem besteht darin, die Interaktion des Anwenders mit dem System weitestgehend reaktiv zu gestalten. Das bedeutet, daß ein Anwender die Ausführung seiner qualifizierten Tätigkeit in der Reihenfolge ausführt, die er für angemessen hält. Das System soll im Idealfall nur noch auf die Eingaben des Benutzers reagieren, in keinem Fall aber die Reihenfolge der Eingaben vorgeben. Dafür bietet sich bei der Implementation der Oberfläche die Metapher der direkten Manipulation an[5].

Anwendungen, die nach der „Werkzeug-Automat-Material-Metapher“ konstruiert werden, sind i.d.R. für interaktive Arbeitsplatzsysteme gedacht. Eine Anwendung unterteilt sich in verschiedene Bausteine, die nach WAM „Werkzeuge“ genannt werden. Dabei nimmt die Konstruktion des GUI einen wichtigen Stellenwert ein. Kilberth et al. betonen bereits im Kapitel „Grundlagen des objektorientierten Softwareentwurfs“ bei der Beschreibung der Architektur von Werkzeugen die Notwendigkeit der Trennung von Interaktion und Funktion innerhalb eines Werkzeuges:

Das Architekturmuster eines Werkzeuges besteht aus einer Funktions- und einer Interaktionskomponente. Durch dieses Prinzip wird die Unabhängigkeit eines Werkzeuges von einer Basissoftware, z.B. einem Fenstersystem, erreicht. Dieses Architekturmuster ist vergleichbar mit dem aus Smalltalk bekannten Model-View-Controller Paradigma zur Trennung von Funktionalität auf der einen Seite und Handhabung und Präsentation auf der anderen Seite.

[5] Fragen der Softwareergonomie sollen in dieser Arbeit aber nicht weiter behandelt werden. Dieses Thema wird z.B. in [Shn87] weiterführend behandelt.

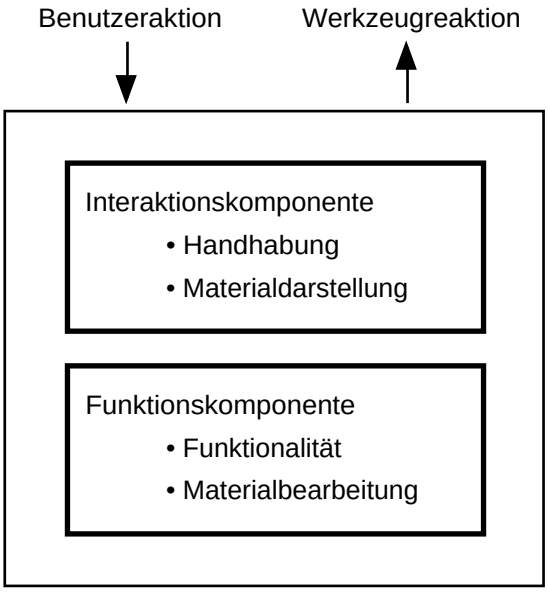


Abb. 7 - Architekturmuster eines Software-Werkzeuges

Die Interaktionskomponente (IAK) präsentiert das Werkzeug an der Benutzungsoberfläche und nimmt die vom Benutzer ausgelösten Ereignisse entgegen. Diese Ereignisse werden interpretiert und in Darstellungsänderungen oder Aufrufe der Funktionskomponente umgesetzt. Damit bestimmt sie die Handhabung des Werkzeuges.

Die aufgerufene Funktionskomponente (FK) ist der bewirkende Bestandteil eines Werkzeuges. Hier findet die Bearbeitung der fachlichen Werte der Anwendung, des sogenannten Materials statt. Dies geschieht in Abhängigkeit der vom Benutzer ausgelösten Ereignisse, ohne auf die Darstellung durch die Interaktionskomponente Bezug zu nehmen.

Die Trennung von Interaktion und Funktionalität innerhalb eines Software-Werkzeuges kann sowohl technisch als auch konzeptionell begründet werden. Konzeptionell charakterisiert die Funktionskomponente, *was* ein Werkzeug leistet; die Interaktionskomponente, *wie* die Leistung dargestellt wird. Hier soll aber im wesentlichen auf die technischen Grundlagen dieser Architektur eingegangen werden.

In der Interaktionskomponente wird die Anbindung des Werkzeuges an das GUI realisiert. Der reaktive Charakter eines Werkzeuges drückt sich dadurch aus, daß Aktivitäten ausschließlich vom Benutzer ausgehen und Zustandsänderungen im Werkzeug immer nur als Resultat dieser Aktivitäten vorkommen können. Die Aktivität im Werkzeug kommt also immer aus Richtung der Interaktionskomponente.

Die Funktions- und die Interaktionskomponente stehen in einer gegenseitigen Benutztbeziehung. Die IAK besitzt die volle Kenntnis über die Schnittstelle der FK. Sie ruft die Methoden des FK-Objektes direkt auf, darf aber keine Annahmen über die Auswirkungen dieser Methodenaufrufe treffen. In der Art des Beobachtermusters (vgl. [GHJ+94]) kennt die FK

ihre IAK lediglich unter der Schnittstelle eines Beobachters und ist so mit dieser nur lose gekoppelt. Der IAK werden über diesen Benachrichtigungsmechanismus eventuelle Zustandsänderungen in der FK gemeldet. Die FK darf dabei keine Annahmen über die Auswirkungen dieser Änderungen in der Präsentation machen. Die benachrichtigte IAK kann die Präsentation des Werkzeuges und des Materials aktualisieren, indem sie über sondierende Funktionen bei der FK die geänderten Daten abfragt. Im Kontext dieser Benachrichtigung darf keine Methode aufgerufen werden, die eine erneute Benachrichtigung nach sich ziehen könnte[6].

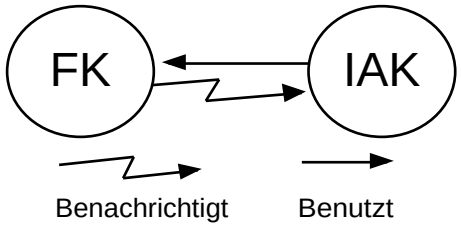


Abb. 8 - Wechselseitige Beziehung zwischen FK und IAK

3.2 Vom GUI-Toolkit zu Interaktionstypen

Um die Benutzungsschnittstelle eines Werkzeuges zu implementieren, bietet sich die Benutzung eines der GUI-Toolkits an, wie sie in Kap. 2 beschrieben wurden. Dadurch wird dem Entwickler schon ein erheblicher Teil der Komplexität bei der Programmierung mit dem API eines Fenstersystemes abgenommen. Die IAK kann die Möglichkeiten des Toolkits nutzen, um das GUI des Werkzeuges zu implementieren. Durch die Trennung des Werkzeuges in FK und IAK können Änderungen an der Oberfläche relativ einfach durch die Weiterentwicklung der IAK durchgeführt werden. Die FK kann unverändert bleiben, da ihr die IAK weiterhin nur unter der (unveränderten) Schnittstelle des Beobachters bekannt ist.

Mit der direkten Benutzung eines GUI-Toolkits durch die IAK kann man sich aber auch erhebliche Probleme einhandeln. Die Abhängigkeit von einem GUI-Toolkit kann gravierende Nachteile mit sich bringen, die z.T. schon einmal in Kap. 2 aufgeführt wurden:

- Bei der Portierung eines Werkzeuges auf ein anderes Fenstersystem muß das eingesetzte Toolkit das neue Fenstersystem unterstützen. Dies kann zwar durch den Einsatz eines Virtual Toolkits erreicht werden, tatsächlich existiert aber kein Toolkit, das sämtliche existierenden Fenstersysteme unterstützt.
- Die Verfügbarkeit eines eingesetzten Toolkits ist niemals vollständig gewährleistet, vor allem nach langen Zeiträumen, wenn vielleicht eine neue Version eines Werkzeuges erstellt werden soll. Häufig wurde die Produktpflege verschiedener Toolkits eingestellt, während sich die unterstützten Fenstersysteme stetig weiterentwickeln und auch neue Fenstersysteme hinzukommen.

[6] In [RW96] wird dies „reaktive Änderung“ genannt.

- Das benutzte GUI-Toolkit wird weiterentwickelt. Dabei kann es zu Änderungen in der Architektur des Toolkits und bei den Schnittstellen der Toolkit-Klassen kommen.
- Die Architektur von GUI-Toolkits gibt zumeist auch die Architektur der Anwendungen vor, die das Toolkit benutzen. Die Toolkits machen dabei enge Vorgaben für den möglichen Kontrollfluß, so daß dies beim Entwurf von Software-Werkzeugen unbedingt berücksichtigt werden muß.
- Die IAK muß sich auf den Ereignismechanismus des Toolkits einstellen. Z.B. müssen Callback Methoden eingeführt oder die IAK von einer bestimmten Klasse abgeleitet werden, um das Ereignisprotokoll der benutzten Widgetklassen zu erfüllen.

  All dies kann für die Weiterentwicklung eines Werkzeuges bedeuten, daß bei gravierenden Änderungen im Toolkit oder bei der Benutzung eines anderen Toolkits die IAK in weiten Teilen geändert oder sogar komplett neu entwickelt werden muß. Ein weiteres Problem kann sich ergeben, wenn ein Werkzeug gleichzeitig für mehrere Plattformen angeboten werden soll. Unter Umständen müssen dann für die verschiedenen Fenstersysteme bzw. GUI-Toolkits unterschiedliche Versionen der IAK verwaltet werden.

  Auf der einen Seite sollte man sich wünschen, daß ein einmal eingesetzes Toolkit sich nicht mehr ändert, um den Aufwand für die Pflege seiner fertigen Werkzeuge möglichst gering zu halten. Andererseits ist man aber darauf angewiesen, daß ein GUI-Toolkit sich den Neuerungen, die im Laufe der Zeit in den Fenstersystemen angeboten werden, anpaßt und somit weiterentwickelt wird. Idealerweise sollte also die Anwendung nicht zu eng mit dem Toolkit verflochten werden.

  Interaktionstypen

  Der Hauptgrund für die Trennung eines Werkzeuges in eine Funktions- und eine Interaktionskomponente war die Entkoppelung der Funktionalität des Werkzeuges von der Handhabung und Präsentation. Dies bietet zugleich die Möglichkeit, durch einen weiteren Schritt das Werkzeug auch vom eingesetzten GUI-Toolkit zu entkoppeln.

  Dabei werden die Widgets des Toolkits, mit denen die Oberfläche eines Werkzeuges zusammengestellt wird, durch sogenannte *Interaktionstypen* (IAT) abstrahiert. Statt die Widgets in der IAK direkt anzusprechen, werden nun IATs benutzt. Diese realisieren intern den Zugriff auf das Toolkit. So kann sich die IAK aus der Abhängigkeit eines bestimmten Toolkits lösen und trotzdem weiter die Handhabung und Präsentation des Werkzeuges bestimmen. Das Toolkit kann ausgetauscht werden, ohne Änderungen an der IAK vornehmen zu müssen, da die Schnittstellen der IAT-Klassen unverändert bleiben. Es muß dabei aber beachtet werden, daß die Menge der Interaktionstypen nur von den Widgets abstrahieren kann, die in möglichst vielen Toolkits vorkommen, da ja eigentlich nur die Widgets in der Schnittmenge der Klassen aus den verschiedenen Toolkits in Frage kommen. Sehr „spezielle“ IATs können also nicht angeboten werden, ohne sich nicht doch wieder in eine (indirekte) Abhängigkeit eines Toolkits zu begeben.

Kapitel 3

Das Prinzip der Trennung in Funktion und Interaktion wurde erstmals in [BKS+86] beschrieben und beruht auf der Architektur von „Model-View-Controller“, die in Smalltalk-Systemen verwendet wird (vgl. [Gol84]). Interessanterweise handelte es sich bei der verwendeten Entwicklungsumgebung um ein Prolog-System. In diesem Zusammenhang wurde auch die Idee der Interaktionstypen entwickelt. Dabei wird der Vorteil einer kleinen, überschaubaren Menge von Interaktionstypen betont. So kann eine Standardisierung bei der Bedienung der einzelnen Werkzeuge in komplexen Software-Systemen erreicht werden. Gleichzeitig sollte dieser Satz von IATs genügen, um einen weiten Bereich möglicher Anwendungen zu entwickeln.

Der von Budde et al. vorgeschlagene Satz von IATs umfaßt folgende Typen:

Button	Auslösen einer Aktion
Select 1	Auswahl 1 aus n; fixe Anzahl von Auswahlmöglichkeiten
Select n	Auswahl m aus n; fixe Anzahl von Auswahlmöglichkeiten
Switch	Ähnlich der Auswahl 1 aus n; die Auswahl wird bei Aktivierung immer um 1 hochgezählt
List Scroll	Auswahl m aus n; dynamische Anzahl von Auswahlmöglichkeiten
Fill In	Formatierte Texteingabe; das Format kann über einen regulären Ausdruck definiert werden

---

Interaktionstyp (IAT)

Ein Interaktionstyp realisiert die abstrakte Handhabung eines Werkzeuges. Eine IAK gebraucht Interaktionstypen, um die mögliche Interaktion mit dem Werkzeug umzusetzen.

Ein Interaktionstyp setzt eine fachliche Umgangsform mit einem Werkzeug um und hat keine Seiteneffekte, die sich auf das Werkzeug auswirken.

Interaktionstypen kapseln die Anbindung an ein GUI-Toolkit und machen ein Werkzeug damit unabhängig vom Toolkit; dieses kann beliebig ausgetauscht werden.

Die IAK bestimmt die Handhabung und Präsentation des Werkzeuges. Durch die Auswahl der Interaktionstypen wird bestimmt, wie der Zustand des Werkzeuges und das Material an der Oberfläche dargestellt und geändert werden kann und damit die Funktionalität umgesetzt wird. Dabei dürfen die einzelnen Interaktionstypen keine Seiteneffekte bewirken, die den Zustand des Werkzeuges verändern könnten.

Realisierung der Interaktionstypen

In der konkreten Umsetzung kapselt jedes IAT-Objekt genau ein Widget. Um den Austausch des Toolkits zu ermöglichen, sollten die Interaktionstypen nur die Schnittmenge der Widgets unterstützen, die von allen Toolkits angeboten werden. Typischerweise sind dies Menüs, Knöpfe, Texteingabefelder oder Listboxen, die praktisch in allen Fenstersystem vorkommen. Das Problem besteht nun genau darin, sich auf einen Satz von Interaktionstypen festzulegen, da sich die Fenstersysteme und damit auch die Toolkits weiterentwickeln. Auf der einen Seite darf man sich also nicht zu sehr festlegen, um nicht doch die angestrebte lose Koppelung zu verlieren. Auf der anderen Seite sollte die Abstraktion bei der Kapselung der Widgets nicht zu weit führen, damit die Anbindung einer konkreten Oberfläche, um die es ja immerhin geht, nicht zu sehr erschwert wird.

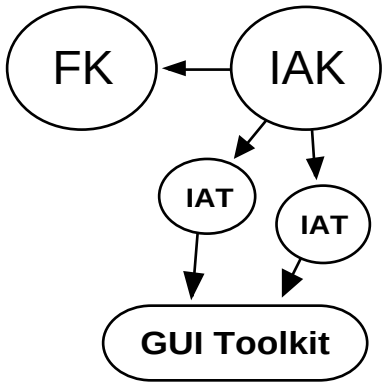


Abb. 9 - Kapselung des GUI-Toolkits

Die Kapselung des Toolkits durch die Interaktionstypen kann erreicht werden, indem Objekt-Adapter[7] eingesetzt werden. In [Frö96] wird dies ausführlich beschrieben und mit anderen, geschickteren Möglichkeiten verglichen. Das Beispiel in Abb. 10 zeigt eine IAT-Klasse, die ein Listbox-Widget kapselt. Mit `IATListBox` wird ein Adapter-Objekt erzeugt. Dieses adaptiert das Objekt einer konkreten IAT-Klasse, die für ein spezielles Toolkit (in diesem Beispiel Motif) entwickelt werden muß.

[7] Zur Diskussion des Adaptermusters siehe [GHJ+94]

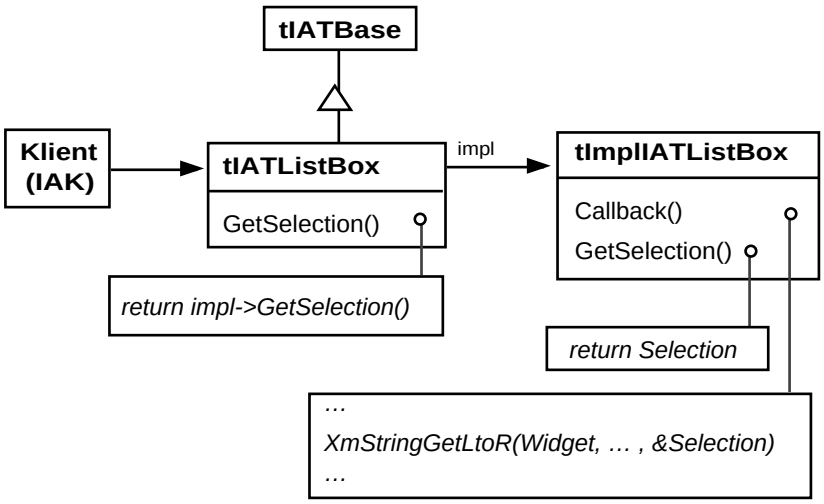


Abb. 10 - IAT mit einfacher Motif-Kapselung (aus [Frö96][8])

Damit die Interaktionstypen verschiedene Toolkits auf verschiedenen Fenstersystemen kapseln können, sollten zudem so viele Informationen über die konkrete Präsentation an der Oberfläche wie möglich außerhalb der IAK verwaltet werden. Einige Toolkits, die einen ähnlichen Weg wie die Interaktionstypen einschlagen, verwalten z.B. die Layout-Informationen getrennt von den Widgets. So könnte für jedes unterstützte Fenstersystem eine sogenannte „Ressourcen-Datei“ angelegt werden, in der Informationen über das Aussehen der einzelnen Widgets gespeichert wird. Damit kann in gewissen Grenzen das „Look and Feel“ der verschiedenen Fenstersysteme eingehalten werden, ohne dies bei der Benutzung der Interaktionstypen in der IAK berücksichtigen zu müssen.

Koppelung von IAT und IAK

Bisher wurde beschrieben, wie mit Hilfe der Interaktionstypen die Benutzung eines konkreten GUI-Toolkits gekapselt werden kann. Ein Problem bei der direkten Benutzung eines Toolkits war, daß die Behandlung des Ereignisflusses vom GUI durch das Toolkit vorgegeben wird. Es gilt also, auch die Rückkoppelung zwischen der IAK und seiner IAT-Objekte zu standardisieren. Dies soll an einem Beispiel erläutert werden:

In Abb. 11 wird die Oberfläche eines Werkzeuges dargestellt, mit dem aus einer Liste von Dateien verschiedene Dateien ausgewählt werden können. Jeder Eintrag, der ausgewählt wurde, wird aus der oberen Liste in die untere verschoben. Das Verschieben von ausgewählten Einträgen zurück in die obere Liste ist ebenfalls möglich.

[8] Es wurde die Notation aus [GHJ+94] übernommen, die auch in dieser Arbeit für alle weiteren Klassendiagramme benutzt wird.

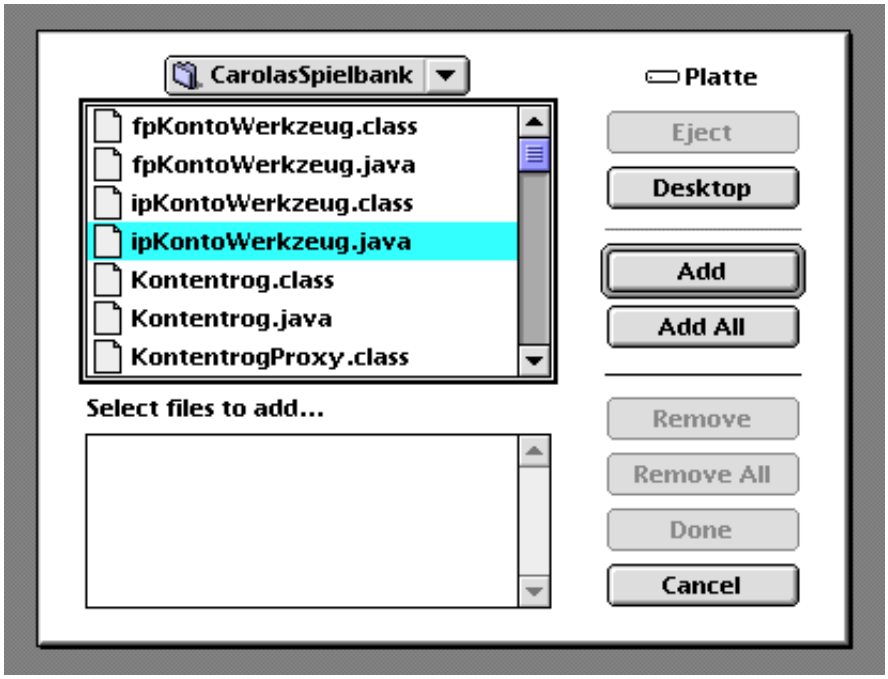


Abb. 11 - Werkzeug, um Dateien auszuwählen

Die Knöpfe und die beiden Listboxen sind durch IAT-Objekte in der IAK erzeugt worden. Der Inhalt der beiden Listen wird durch die FK verwaltet, die auch die Operationen für das Verschieben der Einträge zur Verfügung stellt. Die IAK hat während der Initialisierung die Liste des oberen List-IAT gefüllt. Als Beispiel soll der Kontrollfluß in der Anwendung beschrieben werden, der durch das Klicken auf den Button „Add“ ausgelöst wird.

Der Kontrollfluß kommt mit einem GUI-Event vom Fenstersystem in das Toolkit. Das Event wird dem Knopf „Add“ zugeordnet und über einen Callback an den kapselnden Button-IAT weitergereicht. Dieser interpretiert das Event als Aktivierung und meldet es als „höheres“ Ereignis der einbettenden Interaktionskomponente. Die IAK setzt das Ereignis in einen Operationsaufruf `Verschiebe_Datei(index)` ihrer zugeordneten Funktionskomponente um. Die FK führt die Operation aus und meldet bei Erfolg, daß sich der Zustand des Materials (der Datei- und der Auswahlliste) geändert hat. Die IAK holt sich mit den sondierenden Operationen `Gib_Dateien` und `Gib_Auswahl` das geänderte Material als zwei Listen von Strings und übergibt diese den beiden Listen-IATs.

Dieser Vorgang wird in der folgenden Abbildung noch einmal graphisch in einem Objektinteraktionsdiagramm veranschaulicht.

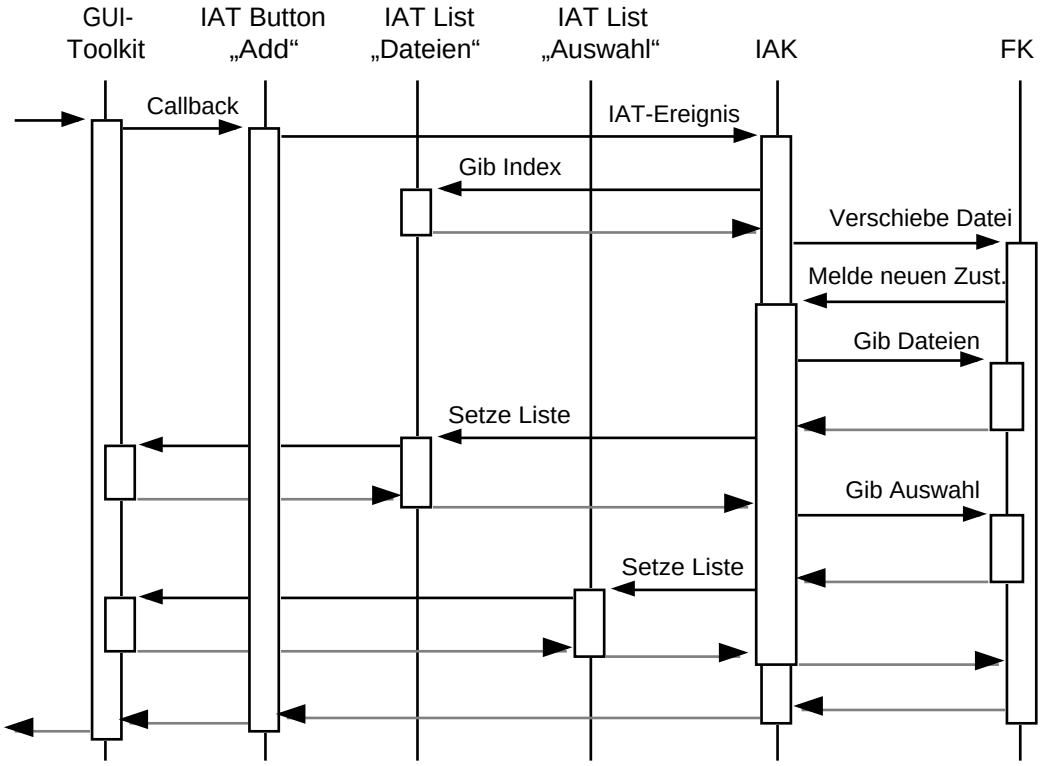


Abb. 12 - Kontrollfluß nach einem Mausklick auf „Add“

Die Koppelung von IAK und IATs ähnelt der Beziehung zwischen IAK und FK. Die Interaktionskomponente besitzt die volle Kenntnis über die Schnittstelle der IAT-Klassen. Die Interaktionstypen melden die GUI-Events als „höhere“ Ereignisse, sollten aber mit der IAK nur lose gekoppelt werden, um sie als wiederverwendbare Bausteine benutzen zu können. Aus diesem Grund darf die IAK nur unter einer abstrakten Schnittstelle bekannt sein. Im Prinzip würde sich - wie schon bei der Koppelung von FK und IAK (siehe Kap. 3.1) - wieder das Beobachtermuster anbieten. Tatsächlich ist dies in frühen WAM-Konstruktionen häufig so realisiert worden. Dieses Muster sollte aber eigentlich für eine Beziehung 1 : n - eine unbestimmte Zahl von Beobachtern schaut auf ein Objekt - benutzt werden, um von diesem eine Zustandsänderung gemeldet zu bekommen. In diesem Fall soll aber mit der IAK nur ein Objekt benachrichtigt werden, so daß eine Koppelung durch Kommando-Objekte nach dem Befehlsmuster (vgl. [GHJ+94]) die bessere Lösung darstellt[9]. Das Befehlsmuster ist dem Beobachtermuster immer dann vorzuziehen, wenn die lose Koppelung eine Beziehung zwischen einer bekannten Anzahl von Objekten realisieren soll, die z.B. in C über einen Zeiger auf eine Funktion als sog. „Callback“ realisiert werden kann. Gamma et al. schreiben:

Use the Command pattern when you want to parameterize objects by an action to perform [...]. You can express such parameterization in a procedural language with a ***callback*** *function, that is, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for callbacks.*

[9] Eine weiterführende Diskussion über sogenannte „Reaktionsmuster“ ist in [RW96] zu finden.

Bei der Benutzung des Befehlsmusters ergibt sich außerdem der Vorteil, daß der benachrichtigende Absender mit dem Kommando-Objekt Parameter übergeben kann, die direkt beim Empfang ausgewertet werden können.

Damit der IAK ein Ereignis von einem IAT gemeldet wird, muß sie für dieses Ereignis ein Kommandoobjekt erzeugen. In diesem Kommandoobjekt ist eine Referenz auf die Interaktionskomponente gespeichert, so daß bei der Ausführung des Kommandos eine bestimmte Methode der IAK aufgerufen wird. Dies kann dadurch erreicht werden, daß für jedes Kommando eine eigene Klasse geschrieben wird, die in voller Kenntnis der IAK ein abstraktes Kommandoprotokoll erfüllt. In Sprachen wie C++ wird typischerweise aber nur ein kleiner Satz standardisierter Kommandoklassen benutzt, die das Protokoll einer abstrakten Kommandschnittstelle erfüllen, die typischerweise Methoden mit der Bezeichnung `Execute` oder `DoIt` definieren. Einem Kommando-Objekt wird ein Zeiger auf die gewünschte Methode der IAK übergeben, über den die Methode aufgerufen werden kann. Dem IAT wird dann das Kommandoobjekt unter dem Typ seiner abstrakten Oberklasse übergeben. Das Kommandoprotokoll wird ausgeführt, indem das IAT-Objekt bei Auftreten eines GUI-Events die Methode `Execute` aufruft. In der konkreten Implementation der Kommandoklasse wird dann die gewünschte Methode der IAK aufgerufen.

3.3 Probleme mit konkreten IAT-Bibliotheken

Am Arbeitsbereich Softwaretechnik existiert als Bestandteil des WAM-Rahmenwerkes „Bib V3.0“ auch eine IAT-Klassenbibliothek names „IAT Motif“, die das API des Motif-Toolkits kapselt. Dabei wird die Sprache C++ benutzt. Ein Beschreibung von „IAT Motif“ findet man im Rahmen der Dokumentation des WAM-Rahmenwerkes „Bib V3.0“ ([Lil97]).

Im Zeitraum von drei Jahren wurden die IAT-Klassen (weiter-) entwickelt und es kamen auch einige neue Klassen hinzu. Derzeit umfaßt „IAT Motif“ im wesentlichen die folgenden Klassen:

`IATBoard`	`IATBoxes`	`IATButton`
`IATCascadeButton`	`IATCheckBox`	`IATCheckButton`
`IATFile`	`IATFrame`	`IATGrids`
`IATList`	`IATMEdit`	`IATMenu`
`IATMenuItem`	`IATPopup`	`IATRadioBox`
`IATRequest`	`IATSEdit`	`IATScrollbars`
`IATSeparator`	`IATText`	`IATWindow`

Anhand der Klassennamen läßt sich bereits erkennen, welchen GUI-Baustein ein Objekt der jeweiligen IAT-Klasse kapselt. Die IAT-Objekte und damit die gekapselten Widgets lassen sich hierarchisch anordnen, so daß mit ihnen eine Oberfläche erstellt werden kann, wie sie z.B. in Abb. 11 dargestellt wird.

Einige Klassen wie z.B. „IATBoard“ oder „IATGrid“ kapseln Widgets, die lediglich für das Layout und das Aussehen der Oberfläche zuständig sind. Sie sind nicht in der Lage, den Zustand des Werkzeuges oder das Material zu präsentieren oder ein Ereignis für die Handhabung des Werkzeuges anzustoßen.

Das konkrete Layout der gekapselten Widgets kann komplett in einer Ressourcenbeschreibung verwaltet werden, die unabhängig von der IAK eingelesen und ausgewertet wird. Diese Ressourcen werden speziell in einem Format für Motif definiert und müßten bei der Benutzung eines anderen Toolkits konvertiert oder neu erstellt werden.

```
class tIATList : public tIATBase
{
 public:
                      tIATList      ( tIATContext *      pParent,
                                      tString            sName );
  virtual            ~tIATList      ( );

  virtual void        Callback      ( XtPointer          Calldata );
  virtual void        GetResList    ( tOrdCol<tResource*> & );
  virtual tString     GetSelText    ( long               No );
  virtual long        GetSelIndex   ( long               No );
  virtual long        GetSelNumber  ( );
  virtual tString     GetText       ( );
  virtual long        GetIndex      ( );
  virtual void        SetIndex      ( long               Index );
  virtual void        SetList       ( tCollection< tString * > * );

  virtual void        RegisterCmd   ( tSelectItemCommandBase * );
  virtual void        RegisterCmd   ( tActivateItemCommandBase * );

 protected:

  ...
};
```

Abb. 13 - Schnittstelle der Klasse `IATList`

Bei der Betrachtung der Schnittstelle von `IATList` fällt auf, daß in der Methode `Callback` mit der Datenstruktur `XtPointer` eine Abhängigkeit zum verwendeten Toolkit besteht. Dies bedeutet, daß bei einer Portierung auf ein anderes Fenstersystem und dem Austausch des Toolkits auch die Schnittstelle dieser IAT-Klasse geändert werden müßte. Dieses Problem läßt sich allerdings durch die oben beschriebene Konstruktion per Adaptermuster nach [Frö96] lösen.

Daneben exisitieren aber noch weitere, z.T. gravierendere Schwächen, z.B. die bereits erwähnte Problematik, überhaupt eine sinnvolle Menge von Interaktionstypen zu finden. In den bestehenden Implementationen ergab sich das größte Problem dadurch, daß jeder Interaktionstyp immer genau ein Widget kapselt. Die Klassen von „IAT Motif“ zeigen, wohin das führen kann. Bei der Auswahl der Interaktionstypen wurden im wesentlichen die gängigsten Motif-Widgets abgebildet. Benutzt die IAK diese Interaktionstypen aus „IAT Motif“, läßt sich bei genauer Betrachtung das Toolkit durch die reine Verwendung der IATs erkennen. Einige Interaktionstypen erfüllen nicht einmal einen konkreten Verwendungszweck im Rahmen der Werkzeugkonstruktion (z.B. `IATSeparator` oder `IATFrame`).

Zusätzlich kann es zu Problemen bei der Einhaltung der Styleguides der verschiedenen Fenstersysteme kommen. Wie in Kap. 2 beschrieben, definieren die Styleguides der verschiedenen Fenstersysteme Regeln, wie das Aussehen der Applikationen für diese Plattform gestaltet werden muß. Für das Look and Feel des Apple Macintosh wird vorgeschrieben, daß eine Menüleiste immer auch einige standardisierte Menüeinträge in einer bestimmten Reihenfolge enthält (Abb. 14). Dies läßt sich in der IAK durch eine Aggregation von Objekten des Typs `IATMenu`, `IATMenuItem` und `IATSeparator` erreichen. Der Styleguide von Windows sieht allerdings einen anderen Aufbau der Menüleiste vor, so daß eine IAK für dieses Fenstersystem eine andere Aggregation der IAT-Menüobjekte vornehmen müßte. Das Werkzeug könnte auf beiden Fenstersystemen aber die gleiche Handhabung bieten. Eventuell ließen sich einige solcher Fälle durch einen geschickten Gebrauch von Ressourcen umgehen.

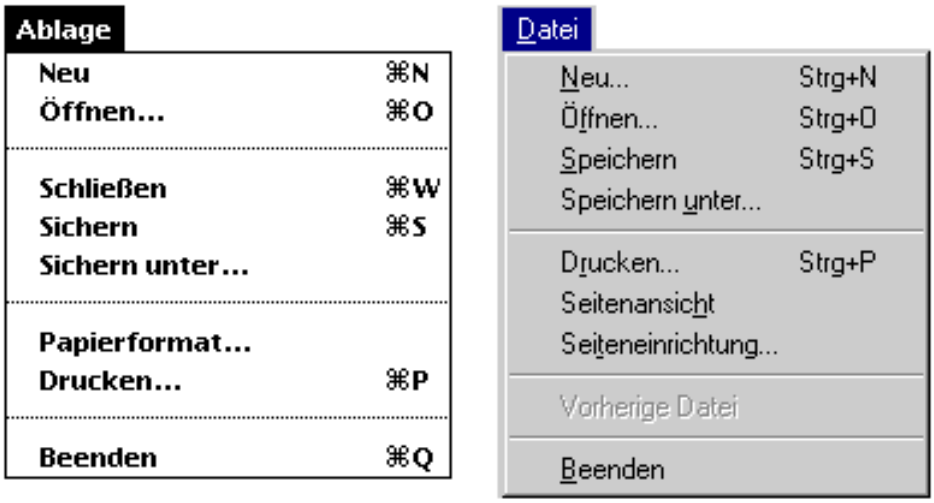


Abb. 14 - Standardmenüs unter MacOS und MS Windows

Die Orientierung an den Widget erweist sich noch in einem weiteren, vielleicht gewichtigeren Punkt als schlechtes Kriterium, um Interaktionsformen zu bilden: verschiedene IAT-Klassen setzen z.T. dieselben Handhabungsformen oder sogar gleich mehrere um.

Zur Veranschaulichung soll noch einmal das Beispiel aus Abschnitt 3.2 mit dem Werkzeug zur Dateiauswahl dienen (siehe Abb. 11). Das Klicken auf den Knopf mit der Beschriftung „Add“ bewirkt das Gleiche wie ein Doppelklick auf einen Eintrag in der oberen Listbox. Aus Sicht der IAK läßt sich also über zwei verschiedene Interaktionstypen eine Funktionalität aufrufen. Technisch kann dies gelöst werden, indem beiden IAT-Objekten dasselbe Kommando-Objekt übergeben wird[10]. Wenn allerdings die Oberfläche geändert werden soll, um z.B. noch einen Menüeintrag „Add“ einzufügen, hinter dem wieder die selbe Funktion wie beim Knopf und dem Doppelklick in die Listbox steht, muß auch die IAK geändert werden. An der Handhabung des Werkzeuges hätte sich aber nichts geändert.

Eine bessere Abstraktion würde also darin bestehen, beim Entwurf der Interaktionstypen nicht von den Widgets eines Toolkits auszugehen, sondern die verschiedenen Möglichkeiten der Handhabung graphischer Benutzungsschnittstellen in den Mittelpunkt zu stellen.

[10] Allerdings kann dann nicht mehr unbedingt für den Einsatz des Befehlsmusters argumentiert werden. Für die Benutzung von Kommandobjekten wurde u.a. angeführt, daß immer nur eine bekannte Anzahl von Interaktionstypen die IAK benachrichtigen wird.

4 Interaktionsformen

4.1 Trennung von Interaktion und Präsentation

Der vorgeschlagene Satz von Interaktiontypen in [BKS+86] läßt vermuten, daß man sich mit der Beschränkung auf wenige IATs eher an der Handhabung der gekapselten Widgets orientiert hat. Allerdings lassen Namen wie „Button“ und „List Scroll“ auch erkennen, daß bereits in dieser ersten Umsetzung nicht unbedingt die Trennung von Handhabung und Präsentation im Vordergrund stand. Bei den Klassen in der „IAT Motif“ Bibliothek wurde dieses Prinzip praktisch aufgegeben und nur noch die Kapselung der Widgets realisiert, so daß sie letztendlich nur eine Einbettung des Motif-API in eine Klassenbibliothek darstellt. Trotzdem kann man die einzelnen IAT-Klassen in Handhabungs- und Präsentationskategorien einteilen und versuchen, sie nur noch unter diesem Aspekt zu betrachten.

	Auslösen einer Aktion	Wert eingeben oder darstellen	Wert "1 aus n" selektieren	Werte auflisten und markieren	Layout und Erscheinungsbild
IATBoard					•
IATBoxes					•
IATButton	•				
IATCascadeButton			•	•	
IATCheckBox			•		•
IATCheckButton			•		
IATFile		•			
IATFrame					•
IATGrids					•
IATList	•		•	•	
IATMEdit		•			
IATMenu					•
IATMenuItem	•				
IATPopup			•		
IATRadioBox			•		
IATRequest	•				
IATSEdit		•			
IATScrollbars					•
IATSeparator					•
IATText		•			
IATWindow	•	•			•

Abb. 15 - Einteilung der Interaktionstypen aus „IAT Motif“ in Kategorien

Wie in Abb. 15 zu erkennen ist, lassen sich einige IAT-Klassen lediglich in die Kategorie „Layout und Erscheinungsbild“ einordnen. Damit sind sie für die Handhabung und Präsentation eines Werkzeuges bzw. des Materials nicht von Bedeutung. Betrachtet man die übrigen Klassen unter dem Gesichtspunkt der Handhabung und nicht unter dem Aspekt der Einbettung eines Toolkits, lassen sich diese Interaktionstypen in die vier Bereiche „Aktivieren“, „Wert darstellen und verändern“, „Wert 1 aus n auswählen“ sowie „Werte auflisten und markieren“ einordnen.

„Aktivieren“ bedeutet, daß IAT-Objekte einer IAK „höhere“ Ereignisse schicken können, die diese in einen Operationsaufruf der FK umsetzen muß. In die Kategorien „Werte darstellen und verändern“ bzw. „Wert 1 aus n auswählen“ fallen im wesentlichen die IAT-Klassen, mit denen das Material dargestellt und manipuliert werden kann. In die Kategorie „Werte auflisten und markieren“ gehören mit `IATList` und `IATCascadeButton` nur zwei Interaktionstypen. Diese Handhabungsform tritt aber so häufig auf, daß die Aufstellung dieser Kategorie gerechtfertigt erscheint. Budde et al. hatten dies ebenfalls erkannt und aus diesem Grund den Interaktionstyp „List Scroll“ eingeführt.

Drei Interaktionstypen fallen gleich in mehrere der relevanten Kategorien. Zur Listendarstellung durch `IATList` wurde bereits erwähnt, daß mit diesem IAT Zeichenketten aufgelistet und markiert werden können. Zusätzlich kann ein Doppelklick auf einen Listeneintrag auch eine Aktion auslösen, so daß `IATList` zusätzlich in die Kategorie „Aktivieren“ fällt. Der IAT `IATCascadeButton` bietet eine sehr ähnliche Funktionalität. Der dritte Interaktionstyp aus „IAT Motif“, der in mehr als eine Kategorie eingeordnet werden kann, ist `IATWindow`. Ein Fenster kann in seiner Titelzeile den Wert einer Zeichenkette darstellen. Außerdem wird i.d.R. durch Anklicken des „Close Button“ eine Aktion ausgelöst, die z.B. als Schließen des Softwarewerkzeuges interpretiert werden kann. Damit gehört `IATWindow` in die Handhabungskategorien „Aktivieren“ und „Wert verändern oder darstellen“. Außerdem ist ein Fenster das oberstes Layoutelement für seine Widgets, so daß dieser IAT auch noch in die Kategorie „Layout und Erscheinungsbild“ fällt.

Interaktionsformen

Mit der Einteilung der IAT-Klassen in die vier Handhabungskategorien kann man eine neue Abstraktion für die in den IATs gekapselten Widgets finden. Ein Widget soll aus der Sicht eines Werkzeuges nur noch unter dem Aspekt der Handhabung betrachtet werden. Die Umsetzung der Präsentation muß komplett außerhalb der Interaktionskomponente verwaltet werden. Das Ziel ist es, die Anbindung eines GUI-Toolkits nicht nur zu kapseln, sondern tatsächlich vollständig zu verbergen, so daß es nicht einmal durch eine implizite Benutzung erkannt werden kann. Die Interaktionskomponente darf die Oberfläche eines Werkzeuges nicht mehr implementieren, indem sie Interaktionstypen benutzt, vielmehr soll eine Anbindung der Widgets erfolgen, indem sie über den Typ ihrer Handhabung mit der IAK verbunden werden. Der IAK muß also ein Ausdrucksmittel zur Verfügung gestellt werden, um die Handhabung des Werkzeuges zu definieren.

Kapitel 4

Ausgehend von den Kategorien in Abb. 15 liegt der Gedanke nahe, sich einen neuen, kompakten Satz von Interaktionstypen zu definieren, um die einzelnen Formen der Handhabung abstrakt zu definieren. Um sich von der bisherigen Umsetzung der Interaktionstypen abzugrenzen, sollen diese „Handhabungstypen“ als *Interaktionsformen* (IAF) bezeichnet werden. Der wesentliche Unterschied zu den bisher bekannten Interaktionstypen besteht darin, daß ein IAF-Objekt nicht mehr für die Kapselung eines einzelnen Widgets zuständig ist. Eine Interaktionsform repräsentiert vielmehr ein „virtuelles GUI-Element“, das tatsächlich genau eine Art der Handhabung umsetzt, dabei aber keinerlei Annahmen über die Präsentation in der Benutzungsschnittstelle macht.

Damit besteht auch keine Notwendigkeit mehr, IAF-Objekte hierarchisch anzuordnen. Dies war ja bei der Benutzung der Interaktionstypen eine Randbedingung, mit der die hierarchische Anordnung der gekapselten Widgets eines Toolkits erzielt werden konnte.

Tatsächlich ist es möglich, nur mit den Interaktionsformen „Aktivieren“, „Wert eingeben oder darstellen“, „Wert selektieren“ sowie „Werte auflisten und markieren“ die Handhabung jeder denkbaren Benutzungsschnittstelle zu beschreiben, die mit den aufgelisteten Interaktionstypen aus „IAT Motif“ implementiert werden kann.

Interaktionsform (IAF)

Eine Interaktionsform definiert die abstrakte Handhabung eines Werkzeuges. Eine IAK gebraucht Interaktionsformen, um die mögliche Interaktion mit dem Werkzeug zu beschreiben.

Eine Interaktionsform repräsentiert eine fachliche Umgangsform mit einem Werkzeug und hat keine Seiteneffekte, die sich auf das Werkzeug auswirken.

Interaktionsformen werden benutzt, um das Werkzeug mit einer konkreten Umsetzung der Handhabung durch ein GUI-Toolkit zu verbinden.

Interaktionsformen machen ein Werkzeug unabhängig von der technischen Präsentation durch das Toolkit, die beliebig ausgetauscht werden kann.

Bei der Konstruktion einer IAK soll sich der Entwickler nicht mehr an der Vorstellung einer konkreten Oberfläche orientieren, wie er es bisher bei der Auswahl der IAT-Objekte getan hat. Stattdessen wählt er aus der Menge der Interaktionsformen die IAFs aus, die gemäß der Funktionalität eines Werkzeuges benötigt werden. Ein Werkzeug wird also komplett ohne die Vorstellung eines GUI entwickelt, das „von außen“ angeschlossen werden muß.

In Anlehnung an die Kategorien für die Interaktionstypen in Abb. 15 führt dies zu einer kleinen Menge von IAF-Klassen[11]. Diese bieten nur noch die Funktionen an, die eine IAK für die Anbindung einer konkreten Benutzungsoberfläche benötigt, nicht aber um die Oberfläche zu implementieren.

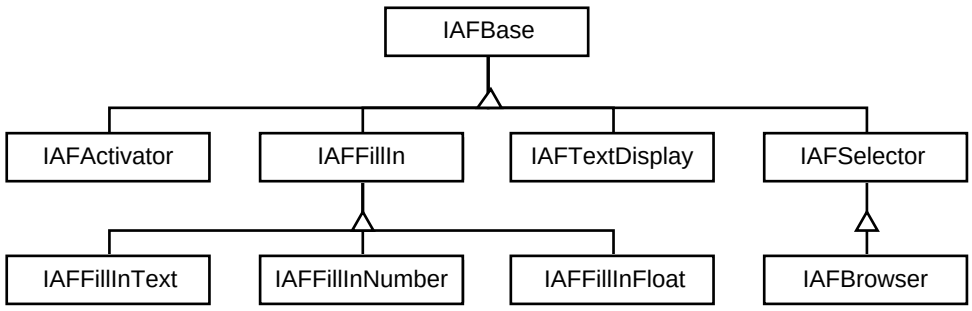


Abb. 16 - Vorschlag einer IAF-Klassenhierarchie

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

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

`IAFFillIn` definiert eine Schnittstelle für die Handhabung des „Ausfüllens“, d.h. Eingabe eines Textes. Zusätzlich werden spezielle Interaktionsformen angeboten, die den Typ des Wertes festlegen, der eingegeben werden kann[12] .

Über Objekte der Klasse `IAFTextDisplay` kann eine Interaktionskomponente Zeichenketten weiterreichen, die an der Oberfläche angezeigt werden sollen. Es besteht keine Möglichkeit der Rückkoppelung.

`IAFSelector` dient zur Anbindung der Handhabung „Auswahl 1 aus n“. Auch diese IAF-Klasse definiert nur die Verbindung mit den Widgets eines Toolkit, die konkrete Präsentation der Auswahl wird vollkommen entkoppelt.

`IAFBrowser` bindet eine spezialisierte Form der Handhabung „Selektion“ ein. Die Art der Auswahl kann dabei auch eine Mehrfachauswahl „m aus n“ umfassen. Außerdem ist die Anzahl der Werte dynamisch, die als Zeichenketten übergeben werden.

---

[11] Tatsächlich existieren noch einige weitere Klassen, mit denen die Einbindung von Graphik umgesetzt werden kann (vgl. Kapitel 5).

[12] Hier ist sicherlich noch weitere Arbeit zu leisten. Statt immer mehr verschiedene FillIn-Klassen zu verwenden, könnte z.B. auch nur eine Klasse entworfen werden, die mit einem Typ parametrisiert wird.

Da die IAF-Klassen - wie die Interaktionstypen - nur mit Basis-Datentypen umgehen können, muß eine Umsetzung fachlicher Werte durchgeführt werden, aus denen sich das Material zusammensetzt. Dies kann die IAK durchführen. Alternativ besteht die Möglichkeit, spezielle IAF-Klassen abzuleiten, die jeweils auf einen fachlichen Wert zugeschnitten sind und diesen intern in die Basistypen konvertieren. Dieser Punkt muß aber in Zukunft noch sehr viel genauer untersucht werden (siehe Anmerkung zur IAF „FillIn“).

Präsentationsformen

Mit der Einführung der Interaktionsformen besteht nun ein Ausdrucksmittel, um in der IAK nur noch eine Vorgabe für die Anbindung einer Benutzungsschnittstelle zu definieren, statt diese mit IAT-Objekten zu implementieren. Was fehlt, ist eine Möglichkeit, die Oberfläche außerhalb des Werkzeuges zu erstellen und an die „IAF-Schnittstelle“ der IAK anzuschließen. Dabei soll für die Umsetzung der konkreten Interaktion und Präsentation auf die existierenden GUI-Toolkits zurückgegriffen werden, um den Aufwand bei der Implementierung möglichst gering zu halten.

Bei der Aufteilung der Interaktionstypen in Interaktions- und Präsentationsanteile wird die Interaktion durch die IAFs beschrieben. Es liegt also nahe, auch die Präsentation in speziellen Typen zu kapseln, die als *Präsentationsformen* (PF) bezeichnet werden sollen.

Präsentationsform (PF)

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

Präsentationsformen kapseln die verschiedenen Widgets eines GUI-Toolkits und erfüllen dabei ein Protokoll, um mit den Interaktionsformen gekoppelt zu werden.

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

Präsentationsformen werden außerhalb eines Werkzeuges verwaltet und mit den Interaktionsformen innerhalb der IAK dieses Werkzeuges verbunden.

In den konkreten PF-Klassen wird letztendlich die Anbindung an das verwendete GUI-Toolkit durchgeführt. Die PF-Objekte werden mit passenden IAF-Objekten innerhalb einer einbettenden IAK gekoppelt, so daß die Handhabung eines Werkzeuges mit seiner Präsentation verbunden wird. Widgets, die keinen konkreten fachlichen Umgang mit dem Werkzeug realisieren, können mit keiner Interaktionsform verbunden werden. Sie sollen aber trotzdem als Präsentationsformen bezeichnet werden.

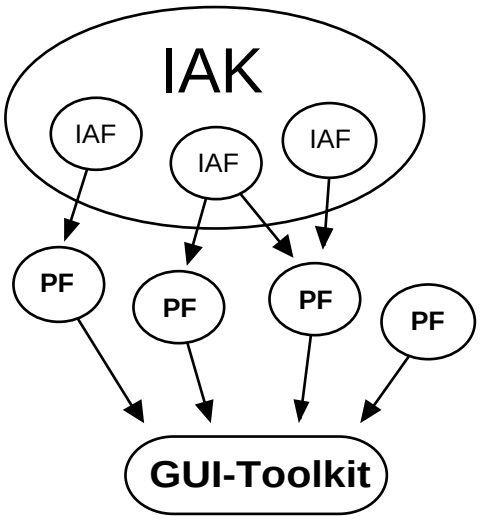


Abb. 17 - Koppelung von IAF, Präsentationsformen und Toolkit

Es gilt das gleiche Prinzip wie bei den Interaktionstypen: jedes PF-Objekt kapselt ein Widget. Die Konsequenz ist, daß für jedes Widget eine konkrete PF-Klasse exisitiert, die wieder jeweils in eine oder mehrere Handhabungskategorie(n) gemäß der Interaktionsformen eingeordnet wird. Diese Kategorien werden technisch als Basisklassen umgesetzt, die aber rein virtuell gehalten werden können. Die Hierarchie der PF-Basisklassen entspricht exakt dem IAF-Klassenbaum. So wird eine mögliche Benutztbeziehung zwischen IAF und PF über die gleiche Position der PF-Basisklasse im deckungsgleichen Klassenbaum definiert. Das Rahmenwerk stellt eine Zuordnung zwischen IAF-Objekten und PF-Objekten her, wenn der Typ einer Interaktionsform zum Basistyp eines konkreten PF-Objektes paßt. Die Widgets werden in die PF-Hierarchie per Adaptermuster eingebettet, wie es es im folgenden Abschnitt 4.2 erläutert wird. So wird der PF-Klassenbaum jeweils für jedes spezielle GUI-Toolkit um zahlreiche konkrete PF-Klassen erweitert, die eine Verbindung zu den Klassen des Toolkits herstellen müssen.

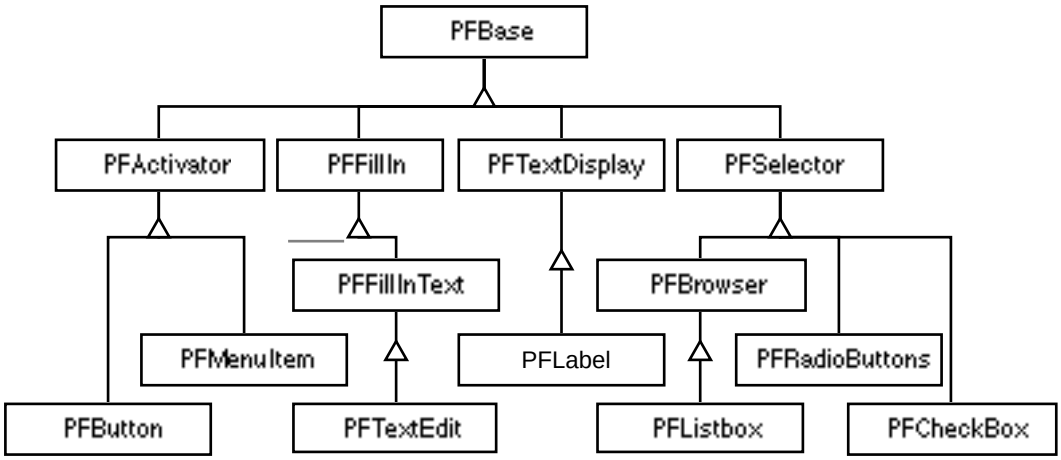


Abb. 18 - Teil eines PF-Klassenbaumes mit abgeleiteten Klassen zur Toolkit-Anbindung

Für PF-Objekte, deren Typ gemäß der Einteilung in Abb. 15 lediglich in die Kategorie „Layout und Erscheinungsbild“ fällt, kann kein passendes IAF-Pendant gefunden werden. Es wird auch keine PF-Basisklasse angeboten, da diese nur für die Verknüpfung mit den IAFs benötigt werden. So werden diese Präsentationsformen vollständig von der IAK eines einbettenden Werkzeuges entkoppelt.

In Abb. 17 werden die drei Möglichkeiten der Benutztbeziehung zwischen IAF- und PF-Objekten dargestellt:

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

In der folgenden Abbildung werden die zweite und dritte Möglichkeit noch einmal anschaulich dargestellt:

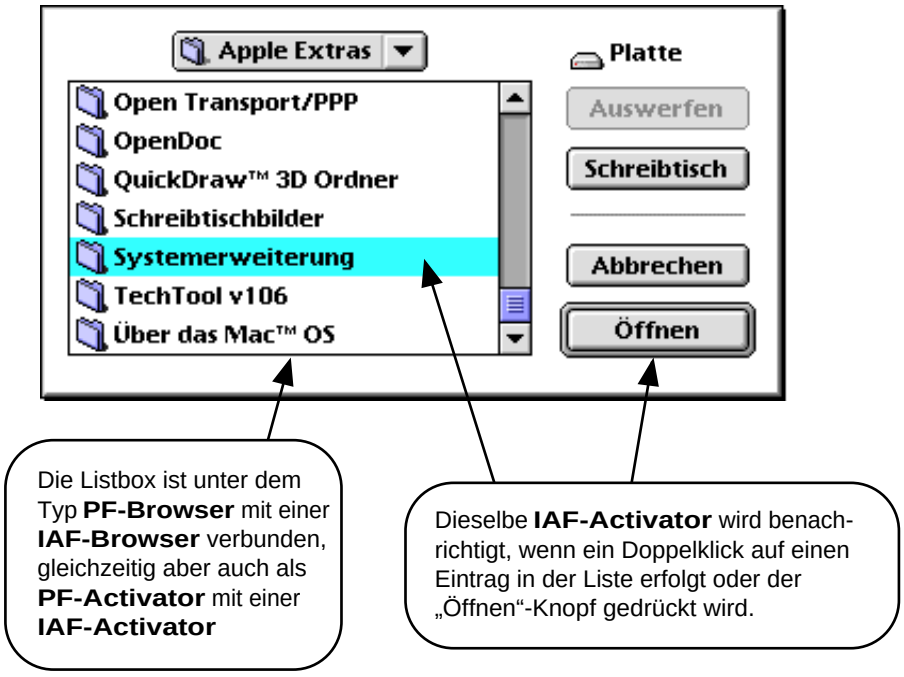


Abb. 19- Verbindung zwischen Interaktions- und Präsentationsformen

4.2 Die Architektur von IAF und PF

Fachlich bieten die Interaktionsformen dem Entwickler ein neues Ausdrucksmittel. Er orientiert sich nun nicht mehr an den Widgets, die durch die Erzeugung der Interaktionstypen in der Benutzungsschnittstelle erscheinen, sondern benutzt Interaktionsformen gemäß der Umgangsformen, die das Werkzeug bietet. In der technischen Benutzung ähneln sie sich dann aber doch. Es werden Kommando-Objekte und Werte übergeben, nach einer gemeldeten Werteänderung kann der neue Wert vom jeweiligen IAF-Objekt erfragt werden. Der Entwickler muß sich allerdings keine Gedanken mehr über die hierarchische Anordnung der Widgets bzw. Interaktionstypen machen, so daß die IAK insgesamt weiter vereinfacht wird.

Durch die Aufteilung der IAT-Klassen in Interaktion und Präsentation kommt in der Anbindung des GUI eine weitere Indirektion zwischen IAK und Toolkit hinzu, die im Vergleich mit der Architektur der Interaktionstypen erhebliche Änderungen nach sich zieht.

Die IAF-Klassen stellen eine Adapterfunktionalität zur Verfügung, mit der die IAK über einen IAF-Objektadapter in der ersten Indirektion mit einem PF-Objekt verbunden wird. Dabei muß die Möglichkeit berücksichtigt werden, daß über diesen Adapter auch eine Verbindung zu mehreren PF-Objekten hergestellt werden kann (z.B. eine IAF-Activator, die mit einem Button und einem Menüeintrag verbunden wird, die dieselbe Aktion auslösen).

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

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

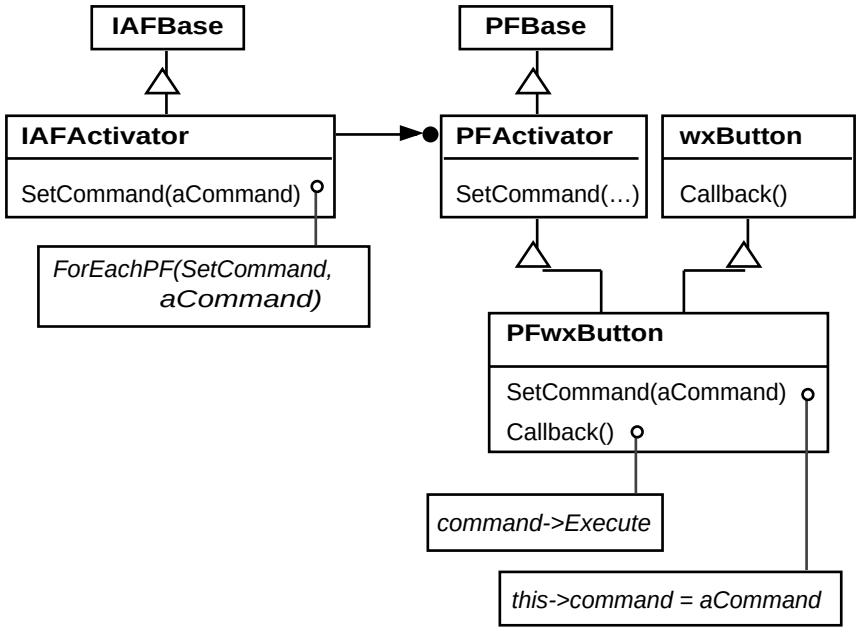


Abb. 20 - Benutzung eines PF-Klassenadapters

In Abb. 20 wird die Anbindung der Klasse `IAFActivator` an eine Widget-Klasse des Toolkits „WxWindows“ (vgl. [Sma97]) dargestellt. `IAFActivator` ist in diesem Fall der Adapter für ein oder mehrere Objekte, deren Typ von PFActivator abgeleitet wurde.

Beim Einsatz von Objektadaptern muß für jede mögliche Verbindung zwischen den Widget-Klassen und den Interaktionsformen mindestens eine PF-Klasse geschrieben werden, mit der ein PF-Objektadapter erzeugt werden kann (vgl. Abb. 21). Die Widgetklassen dienen dabei nicht als Basisklassen, sondern werden lediglich benutzt. Die konkreten PF-Klassen werden nur von den PF-Basisklassen abgeleitet, so daß es zu einer einfachen Vererbungsbeziehung kommt.

Die Beziehung zwischen Interaktions- und Präsentationsformen ähnelt in beiden Fällen dem Proxy-Muster (vgl. [GHJ+94]), da sich weite Teile der Schnittstellen von IAF und zugehörigen PF überschneiden. Die Interaktionsformen reichen im wesentlichen die Methodenaufrufe an die PF-Objekte nur durch, um ihrer IAK keinen direkten Zugriff auf die Präsentationsform zu gewähren. Mit der Möglichkeit der Aggregation von PF-Objekten bekommt die Architektur zusätzlich den Charakter eines Kompositionsmusters (vgl. ebenfalls [GHJ+94]). Dabei kann vor der IAK verborgen werden, wie groß die Anzahl der Präsentationsformen ist, die mit einer IAF verbunden sind. Sind es mehrere, muß die Interaktionsform intern für die Synchronisation ihrer Präsentationsformen sorgen.

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

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

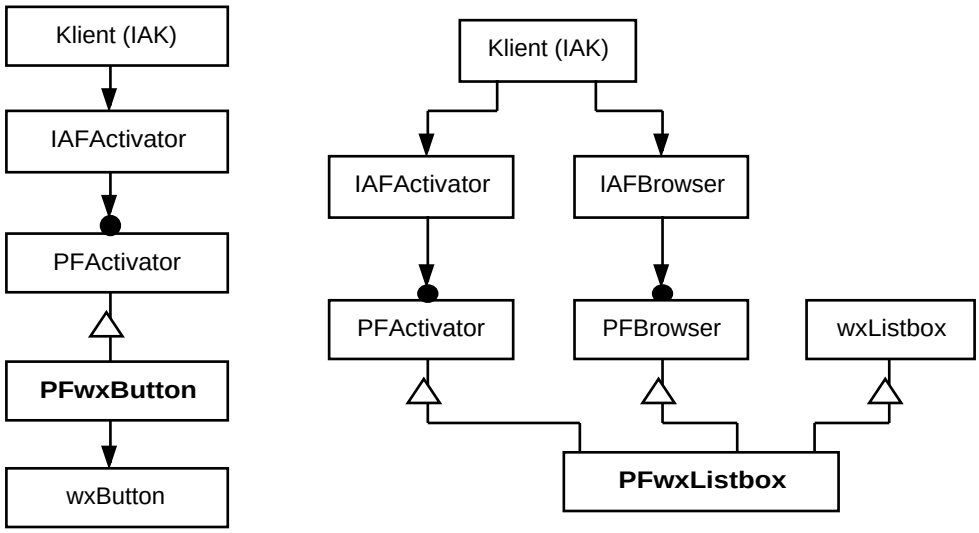


Abb. 21 - PF-Objektadapter und Klassenadapter mit drei Basisklassen

Werden Klassenadapter benutzt, ergibt sich ein wesentlicher Vorteil, den Lippert bei der Implementation der Präsentationsformen in Java erkannt hat (vgl. [Lip97]).

Lippert argumentiert folgendermaßen:

[…] bietet sich in Java die Möglichkeit an, die Präsentationsform-Klassen von den Widget-Klassen erben zu lassen. […] Da eine solche Konstruktion innerhalb des Frameworks liegt und vor dem Benutzer des Frameworks komplett verborgen ist, halte ich eine solche Konstruktion für vertretbar. [...] Der wohl wichtigste (Vorteil) ist die Tatsache, daß sich diese von den AWT-Klassen[13] 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.

[13] Das verwendete GUI-Toolkit in Java heißt AWT („Abstract Windowing Toolkit“).

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

Die Widget-Klassen bilden jeweils die Oberklasse für eine konkrete PF-Klasse und werden so durch Implementationsvererbung eingebunden. Durch diese Verknüpfung mit der Widgetklasse können die Schnittstellen der einzelnen PF-Klassen ziemlich umfangreich werden. Lippert berichtet bei der Benutzung des Java AWT-Toolkits von PF-Klassen, die alleine durch die AWT-Wurzelklasse um 130 (!) Methoden erweitert werden. Dies sei aber vertretbar, da die Präsentationsformen komplett im Rahmenwerk gekapselt werden und ein Entwickler nur die Interaktionsformen benutzt, mit den Schnittstellen der Präsentationsformen also nicht in Berührung kommt.

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

4.3 Dynamik zur Laufzeit

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

Interaktionsformen werden innerhalb einer IAK erzeugt und benutzt. Dies ist notwendig, damit die IAK den Zustand des Werkzeuges und des Materials weitergeben kann. Außerdem müssen Eingaben des Benutzers an die Funktionskomponente weitergeleitet werden, die dann ihrerseits den Zustand des Materials oder des Werkzeuges ändern kann und dies wieder der IAK meldet. Anders sieht es mit den Präsentationsformen aus. Für diese wurde bisher immer nur die möglichst lose Kopplung mit der IAK betont. In den dargestellten Klassendiagrammen kann man erkennen, daß die IAF-Klassen die PF-Klassen benutzen. Gleichzeitig wurde aber nur erwähnt, daß die PF-Objekte außerhalb der IAK verwaltet werden. Wie also werden die IAFs einer IAK mit ihren Präsentationsformen verbunden?

Die Interaktionsformen können eindeutig identifiziert werden, da ihnen innerhalb einer IAK zusätzlich zur Objektidentität ein eindeutiger Bezeichner zugewiesen wird. Daneben besitzen auch die Präsentationsformen einen Bezeichner. Mit Hilfe dieser beiden Bezeichner wird die Zuordnung von IAF und PF hergestellt. Ein IAF-Objekt wird mit einem PF-Objekt genau dann verbunden, wenn ihre Typen im IAF-PF-Rahmenwerk als zusammengehörig definiert wurden und beiden Objekten der gleiche Bezeichner zugeordnet wurde. Lippert nennt dies in [Lip97] „Interaktionsname“. Die mögliche Zuordnung von IAF- und PF-Klassen wurde im Kapitel über die Architektur implizit dadurch gezeigt, daß der Teil des PF-Klassenbaumes, der die Basisklassen umfaßt, deckungsgleich mit dem IAF-Klassenbaum dargestellt wurde.

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

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

Präsentationsformen im Kontext eines Werkzeuges

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

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

Im Rahmenwerk erzeugt der Basiskonstruktor einer IAK das IAF-Kontextobjekt, das intern zusätzlich ein Objekt vom Typ `PFsContext` erzeugt[14]. Die IAK muß im Konstruktor jeder Interaktionsform das IAF-Kontextobjekt als Parameter übergeben. Bei diesem registrieren sich die Interaktionsformen, so daß das IAF-Kontextobjekt Kenntnis über alle IAF-Objekte einer IAK bekommen kann.

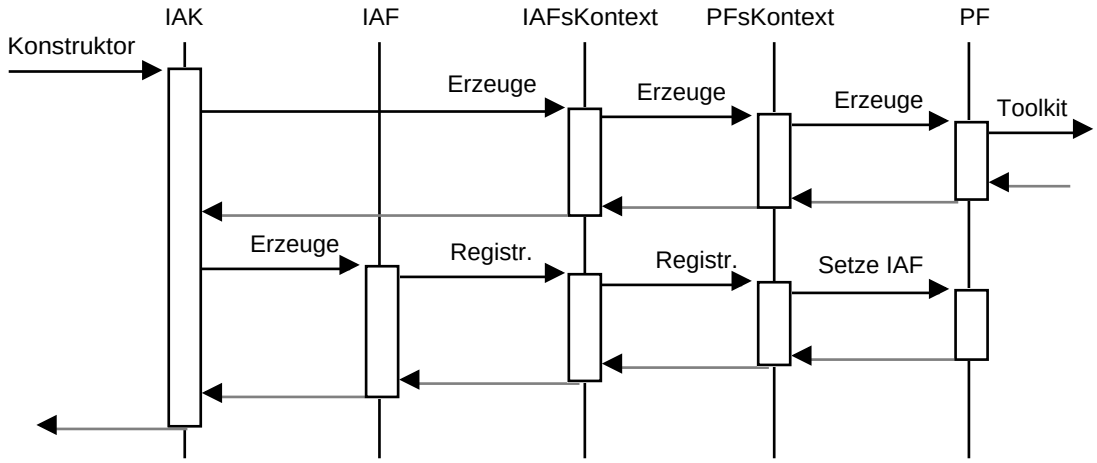


Abb. 22 - Initialisierung einer IAK[15]

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

IAF, PF und Interface Builder

Das PF-Kontextobjekt hat bei seiner Erzeugung auch die komplette Menge seiner PF-Objekte erzeugt. Dies könnte per Hand programmiert werden, indem sich alle benötigten PF-Kontextklassen bei einer zentralen Instanz im IAF-PF-Rahmenwerk anmelden, die bei Bedarf für die späte Erzeugung der Kontextobjekte benutzt werden. In den beiden „Referenzimplementationen“ in C++ (siehe Kap. 6) und in Java (vgl. [Lip95]) sind die Rahmenwerke aber so angelegt, daß die PF-Kontextobjekte serialisierte Präsentationsformen einlesen, die in einem GUI-Builder definiert und abgespeichert wurden. Dies bietet den

[14] Bei der Beschreibung der Implementation in Kap. 6 wird für diese Aufgabe ein Singleton-Objekt vom Typ „GUI-Manager“ vorgestellt.

[15] In der Java-Implementation ist die Klasse `IAFsKontext` nur ein Interface, das durch die konkrete Klasse `PFsKontext` implementiert wird. So muß nur ein Kontextobjekt verwaltet werden.

Entwicklern eine enorme Erleichterung, da nur noch die Einbindung der Interaktionsformen in der IAK per Hand kodiert werden muß. Änderungen an der konkreten Oberfläche werden im GUI-Builder vorgenommen und gesichert und können direkt ausprobiert werden, ohne eine einzige Klasse des Werkzeuges neu übersetzen zu müssen.

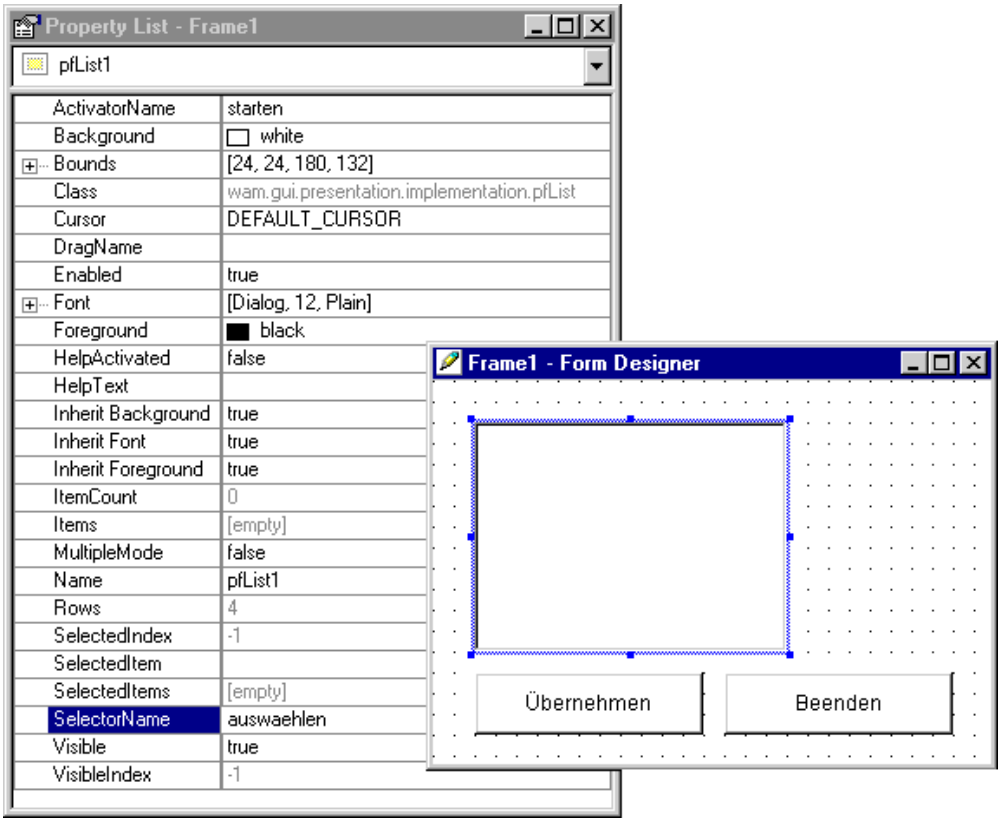


Abb. 23 - PF-Objekte in einem Interface Builder für Java

Setzen von Werten und Kommandos

Nachdem die IAF- und PF-Objekte erzeugt und verknüpft wurden, muß die IAK ihre Interaktionsformen initialisieren. Die Materialien werden in Werte von Basisdatentypen konvertiert, die von den standardisierten Interaktionsformen bzw. Präsentationsformen verarbeitet werden können. Diese Werte werden dann den IAF-Objekten übergeben, die sie an die PF-Objekte durchreichen. Die Präsentationsformen stellen die Werte an der Oberfläche dar und bieten ggf. auch die Möglichkeit, sie zu verändern. Wie in Abschnitt 4.1 angedeutet wird, kann für spezielle, fachliche Werte auch eine andere Lösung gefunden werden.

In einem vereinfachten Beispiel könnte z.B. ein fachlicher Wert „Adresse“ in vier Zeichenketten konvertiert werden. Diese würden an vier Interaktionsformen vom Typ `IAFFillInText` weitergegeben werden, die sie an ihre angeschlossenen Präsentationsformen durchreichten. Die PF-Objekte müßten die Adresse in vier Texteingabefeldern anzeigen, um es dem Anwender zu ermöglichen, dort Änderungen vorzunehmen. Nach einem Ereignis, das als ein „Handlungsabschluß“ für das Editieren der Adresse definiert worden ist und z.B.

durch einen Mausklick auf einen Button ausgelöst wurde, müßte die IAK die vier Zeichenketten von den IAF-Objekten erfragen. Wenn möglich, muß sie die Zeichenketten dann wieder zu einem fachlichen Wert „Adresse“ zusammensetzen, der schließlich an die FK weitergereicht wird.

Wie schon bei den Interaktionstypen werden für die Verarbeitung von Ereignissen wieder Kommando-Objekte nach dem Befehlsmuster benutzt (vgl. Kap. 3.2). In der Initialisierungsphase erzeugt eine IAK Kommando-Objekte, die den Interaktionsformen für die verschiedenen Ereignisse übergeben werden. Dieses Vorgehen kommt vor allem zum Einsatz, wenn IAF-Aktivator-Objekte benutzt werden. Zudem wurde vorgesehen, daß auch IAF-Objekte, die einen Wert manipulieren, ebenfalls Kommando-Objekte benutzen, um eine Änderung des Wertes signalisieren zu können.

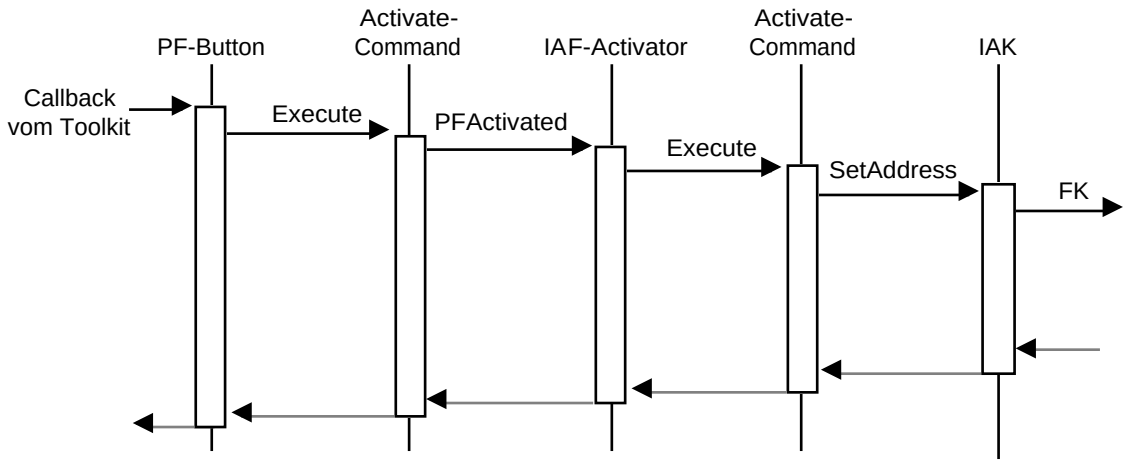


Abb. 24 - Auslösen des Kommando-Mechanismus nach Klick auf einen Button

In Abb. 24 wird die Verarbeitung eines Ereignisses dargestellt, das durch einen Mausklick auf einen Button ausgelöst wurde. Alternativ könnte auch nur ein Kommando-Objekt benutzt werden, das der IAF von der Interaktionskomponente gesetzt und an die PF durchgereicht wurde. So könnte der Aufruf von `Execute` in der Präsentationsform direkt einen Methodenaufruf in der IAK bewirken. Lippert merkt dazu an, daß es dann aber zu Problemen bei der Synchronisation zwischen IAF und PF kommen kann.

4.4 Aspekte der Werkzeugkonstruktion

Durch die hierarchische Anordnung der Präsentationsformen ergibt sich ein Problem, das auch schon bei der Benutzung von Interaktionstypen gelöst werden mußte. Die Konstruktion von Werkzeugen nach der WAM-Metapher sieht vor, daß Werkzeuge hierarchisch angeordnet werden können. Dabei wird vom „Kontext-Werkzeug“ (bzw. „Kontext-IAK“) gesprochen, das ein „Subwerkzeug“ („Sub-IAK“) einbettet.

Jede Interaktionskomponente bildet natürlich auch für ihre IAF-Objekte einen Kontext. Die Interaktionsformen einer Sub-IAK exisitieren unabhängig von den IAF-Objekten der einbettenden IAK. Die zugehörigen PF-Objekte werden außerhalb der IAK verwaltet. Dabei wird ein Widget, z.B. ein Fenster oder ein Rahmen, als Elternobjekt der PF-Objekte definiert. Dieses Widget wird im folgenden auch wieder als Präsentationsform bezeichnet.

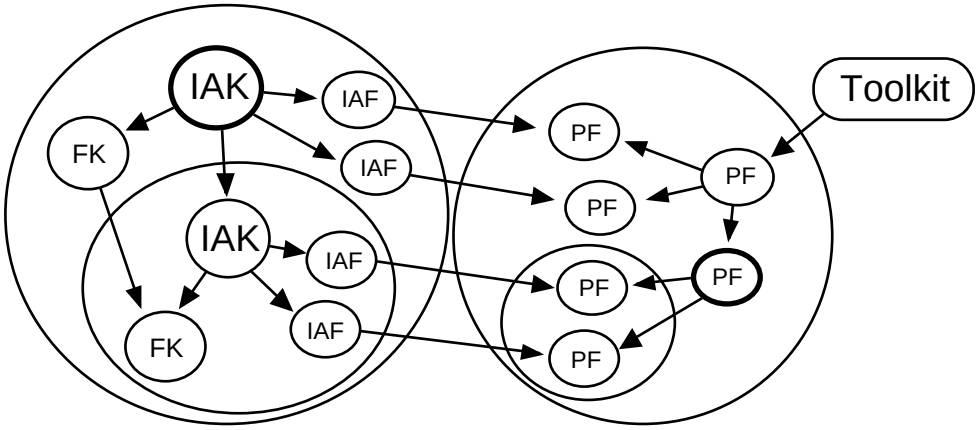


Abb. 25 - Einbettendes und Subwerkzeug, Interaktionsformen und PF-Baum

Bei der Benutzung solch einer Werkzeugkonstruktion liegt das Problem auf der technischen Ebene darin, daß ausgehend von dem Kontext-Werkzeug **sämtliche** PF-Objekte in einer baumartigen Struktur angeordnet sein können. Die Verwaltung der Präsentationsformen läßt sich nicht auf den Kontext eines Werkzeuges beschränken. Es besteht die Möglichkeit, daß einige PF-Objekte auch über die Grenzen eines Werkzeuges hinaus miteinander verbunden werden müssen (im folgenden Beispiel dargestellt). Den Präsentationsformen eines Subwerkzeuges muß aus dem Kontext des einbettenden Werkzeuges eine PF zur Verfügung gestellt werden, die ihnen als Wurzelobjekt dient.

Eine einbettende IAK und die einbettende PF für die Präsentationsformen der Sub-IAK sind in der schematischen Darstellung (Abb. 25) fett umrandet dargestellt. In Abb. 26 ist ein Werkzeug abgebildet, das einen vergleichbaren Fall in der konkreten Umsetzung zeigt.

Das einbettende Kontext-Werkzeug („AufgabenVerwalter“) verwaltet das Material, das aus einer Liste von Aufgaben besteht. Die dazugehörigen Präsentationsformen stellen das Fenster, die Menüleiste und die beiden Buttons „Erzeugen“ und „Loeschen“ dar. Eingebettet werden zwei Subwerkzeuge. Das erste listet die Aufgaben auf und bietet dem Benutzer die Möglichkeit, eine Aufgabe zu selektieren. Es meldet seinem Kontext-Werkzeug, wenn eine Selektion erfolgt ist. Das zweite Subwerkzeug kann dazu benutzt werden, eine Aufgabe zu bearbeiten. Zu ihm gehören die Präsentationsform für die Eingabe eines Stichwortes, mit dem die Aufgabe bezeichnet wird und die PF, um einen weiteren Text für die Aufgabe zu editieren. Außerdem kann über einen PF-Button signalisiert werden, daß Änderungen gesichert werden sollen. Zu jeder Präsentationsform gehört mindestens ein IAF-Objekt, das nur

in der IAK seines Werkzeuges bekannt ist. Das Toolkit gibt aber vor, daß die Widgets hinter den Präsentationsformen über die Grenzen der Werkzeuge hinaus hierarchisch miteinander verbunden werden müssen. Der Listbox des Aufgabenlisters muß als Elternobjekt das Fenster des einbettenden Aufgabenverwalters übergeben werden.

Abb. 26 - Einbettendes Werkzeug mit Subwerkzeugen

Technisch entsteht also die Notwendigkeit, die Präsentationsformen über die Werkzeuggrenzen hinaus anzuordnen. Da ein Subwerkzeug - wie z.B. der Aufgabenlister - unabhängig von seinem einbettenden Werkzeug (dem Aufgabenverwalter) entwickelt werden soll, muß eine Möglichkeit angeboten werden, um die Einbettung der Präsentationsformen im Kontext-Werkzeug zu spezifizieren.

Im Beispiel des Aufgabenverwalters ist das Fenster dazu bestimmt, die Präsentationsformen der Subwerkzeuge aufzunehmen. Das PF-Objekt vom Typ `PFWindow` steht in der Kontext-IAK aber nur in Verbindung mit einer IAF-Activator, die eine Aktivierung auslöst, wenn das komplette Werkzeug über den Schließen-Knopf des Fensters beendet werden soll. Da die IAK aber nicht weiß, daß diese Interaktionsform auch mit der einbettenden Präsentationsform verbunden ist, hat sie keine Möglichkeit, das Fenster irgendwie als das Eltern-Objekt zu bestimmen.

Aus diesem Grund wird die Präsentationsform „Composite“ als PF-Basistyp eingeführt, die aber keiner Interaktionsform zugeordnet werden kann. Die zugehörige Klasse `PFComposite` besitzt eine kompakte Schnittstelle, mit der lediglich ein Bezeichner verwaltet werden kann. Allen PF-Objekten muß bei der Erzeugung ein PF-Composite als Elternobjekt übergeben

werden. Im Interface Builder geschieht dies implizit, wenn z.B. eine PF-Button in einer PF-Window plaziert wird, deren Typ von `PFComposite` abgeleitet wurde. Einen Button in ein Texteingabefeld zu setzen, ist hingegen nicht möglich.

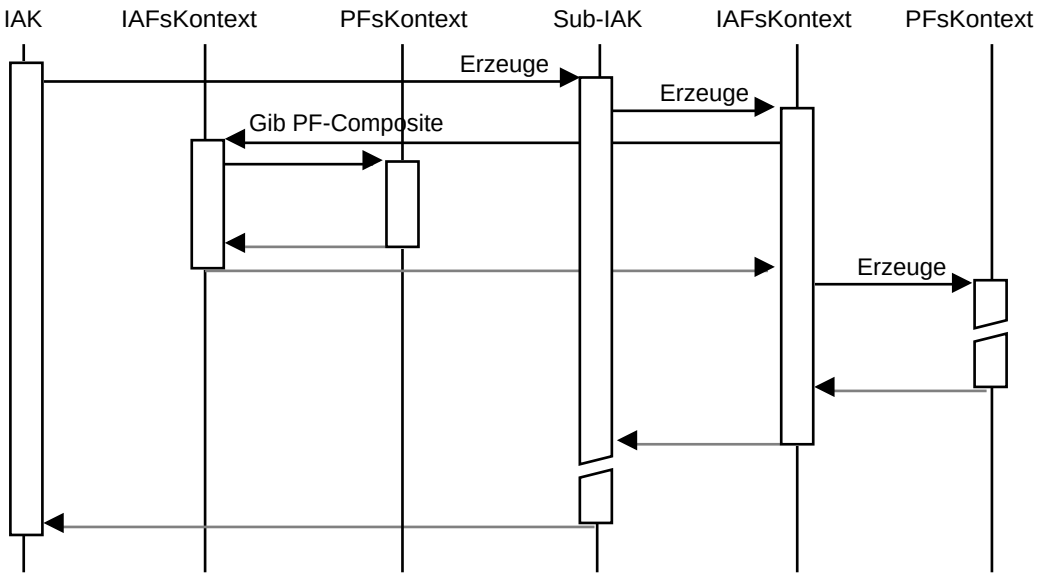


Abb. 27 - Erzeugung einer Sub-IAK und Verknüpfung der Präsentationsformen

Wird eine Sub-IAK erzeugt, muß die Kontext-IAK den Namen des Subwerkzeuges angeben. Daraufhin wird in der Menge der Präsentationsformen des einbettenden Werkzeuges die PF-Composite gesucht, die den Namen des Subwerkzeuges trägt. Die Präsentationsformen, die ein PF-Kontextobjekt für die Interaktionsformen der Sub-IAK erzeugt (vgl. Abschnitt 4.3), werden in der PF-Hierarchie dieser PF-Composite zugeordnet. Der Name des Subwerkzeuges kann dabei nicht durch den Typ des Werkzeuges bestimmt werden. Dies würde die Möglichkeit verhindern, zwei Subwerkzeuge des gleichen Typs gleichzeitig in einem Kontext-Werkzeuge einzubetten, da dann über den Namen keine eindeutige Zuordnung zu den zwei benötigten PF-Composite möglich ist.

Um die Präsentationsformen, die in einem Interface Builder für ein kleines, abgeschlossenes Werkzeug definiert werden, in die Oberfläche des einbettenden Werkzeuges zu integrieren, wird im Interface Builder mit einer PF-Composite im Kontext des einbettenden Werkzeuges lediglich ein Platzhalter definiert.

Hier läßt sich ein Zusammenhang zwischen der fachlich motivierten Anordnung von Werkzeugen und der technisch bedingten Verknüpfung der Widgets bzw. Präsentationsformen erkennen. Ein wesentliches Merkmal der WAM-Architektur lautet Wiederverwendung. Kleine Werkzeuge können wiederverwendet werden, indem spezielle, einbettende Werkzeuge sie benutzen. Über die PF-Composite kann die Anordnung der Werkzeuge auf der Ebene der Präsentationsformen nachvollzogen werden, so daß auch hier eine Wiederverwendung möglich wird.

5 Einbindung von Graphik

Die bisher vorgestellten Interaktionsformen stellen eine gute Abstraktion dar, um die „einfacheren“ Widgets eines Toolkits zu verbergen, wie z.B. Buttons, Listboxen, Menüleisten oder Texteingabefelder. Mit „einfach“ ist gemeint, daß es sich bei den abgebildeten Formen der Interaktion neben der Aktivierung immer um die Präsentation und Manipulation von Werten handelt, die für diesen Zweck in Zeichenketten umgewandelt werden können. Für anspruchsvolle Benutzungsschnittstellen, die den heutigen Anforderungen entsprechen sollen, genügt dies aber häufig nicht mehr.

Wie Weiß in [Wei97] schreibt, gibt es auch im klassischen Kontext von WAM-Applikationen Bedarf an graphischen Komponenten, z.B. im Bankenbereich. Als Beispiele nennt er die Darstellung von Börsenkursen und Statistiken, im Zusammenhang von Immobiliengeschäften könnte eine photograpische Darstellung der Immobilie erwünscht sein. Aber nicht nur die Darstellung von Graphik soll unterstützt werden; vielmehr besteht in vielen Anwendungsfällen auch der Bedarf, Graphiken zu bearbeiten, z.B. um Organigramme zu erstellen oder die graphische Darstellung von Statistiken nachträglich zu bearbeiten.

In seiner Arbeit beschreibt Weiß die Erweiterung der Interaktiontypen in diese Richtung, die er für das WAM-Rahmenwerkes „BibV30“ (vgl. Kap. 3) vorgenommen hat. Er orientierte sich dabei an bewährten technischen Entwürfen, die er vor allem den Entwurfsmustern des Rahmenwerks „Unidraw“ entliehen hat, das in [VL90] vorgestellt wird. Die Entwickler von Unidraw wurden ihrerseits u.a. von Smalltalk und MacApp geprägt.

Also müssen auch die Interaktions- und Präsentationsformen, die ja als eine Weiterentwicklung der Interaktionstypen gelten sollen, ein Konzept anbieten, um graphische Darstellungen und Manipulationen realisieren zu können. In diesem Abschnitt soll hierfür ein Ansatz vorgestellt werden, der in einem GUI-Builder für Präsentationsformen bereits praktisch erprobt wurde (siehe Kap. 6). Es soll aber betont werden, daß die folgende Architektur noch weiter diskutiert werden muß, da ihre Umsetzung im Gegensatz zu den bisher vorgestellten Interaktions- und Präsentationsformen tatsächlich nur in diesem einen Fall erfolgte und die Erfahrungen mit diesem Konzept demzufolge ziemlich rar sind.

Zuvor will ich aber noch eine wesentliche Forderung für den Entwurf von Werkzeugen und Materialien nach der WAM-Metapher aufstellen: Sollen fachliche Werte eines Materials mit graphischen Mitteln angezeigt und vor allem manipuliert werden, sollte das Material auch Attribute besitzen, um die graphischen Eigenschaften des Materials zu beschreiben. So können zu einem Material auch Eigenschaften wie beispielsweise Farbe, Textattribute (z.B. fett, kursiv, unterstrichen) oder geometrische Werte gehören. Würde man diese Attribute erst in den Präsentationsformen behandeln, träten sofort wieder Probleme auf, wenn es etwa darum geht, die graphischen Attribute zusammen mit den Materialien zu

speichern. Im Beispiel der oben erwähnten Organigramme läßt sich leicht erkennen, daß geometrische Aspekte sehr wohl zum Material und damit zu den fachlichen Werten gehören. Im weiteren Verlauf dieses Kapitels wird also angenommen, daß die Materialien der Anwendungswelt auch graphische Werte umfassen können[16].

Wie in der Erweiterung der Interaktionstypen durch Weiß sollen auch die graphischen IAF und PF in der Art der Entwurfsmuster nach Unidraw zusammenwirken. Aus diesem Grund werden hier auch Bezeichnungen wie „Canvas“, „View“ oder „Manipulator“ eingeführt, die z.T. bereits vor Unidraw schon von MacApp und Smalltalk geprägt wurden.

5.1 IAF Canvas

Die graphische Darstellung von Objekten erfolgt in Rahmenwerken wie Unidraw i.d.R. in einem speziellen graphischen Bereich, der durch ein Widget verwaltet wird und typischerweise „Canvas“ genannt wird („Leinwand“). Dabei werden primitive Zeichenoperationen angeboten, mit denen z.B. Linien, Rechtecke oder Kreise gezeichnet oder Texte dargestellt werden können. Für alle Operationen lassen sich Parameter angeben, die z.B. die Stärke und Farbe einer Linie, das Füllmuster eines Rechteckes oder den Stil einer Schrift definieren.

Für eine Canvas wird neben der Größe des Canvas-Widgets auch die Größe einer virtuellen Ausgabefläche definiert, die nicht unbedingt gleich sein muß. Die virtuelle Ausgabefläche in einem Zeichenprogramm kann z.B. eine DIN A4 Seite umfassen. Diese kann im Verhältnis 1:1 auf dem Bildschirm dargestellt werden, je nach Maßstab könnte sie aber auch verkleinert oder vergrößert erscheinen. In der Welt der Widgets existiert auf dem Bildschirm nur die Maßeinheit Bildschirmpunkt („Pixel“), die sich durch die Größe der Bildschirmfläche ergibt. Bildschirme mit 17 Zoll Bilddiagonale haben typischerweise eine Auflösung zwischen 1024 x 768 Pixel und 1280 x 1024 Pixel. Graphische Objekte werden intern aber meistens sehr viel genauer repräsentiert und geometrische Angaben wie Position und Größe üblicherweise in Millimetern oder Inch durchgeführt.

Die Darstellung in Abb. 28 zeigt ein Graphik-Werkzeug, das eine Seite mit Originalgröße DIN A4 dreifach verkleinert darstellt. Das Werkzeug arbeitet mit einer Bildschirmauflösung von 72 dpi („dots per inch“), d.h. eine horizontale Linie mit einer Länge von 72 Pixeln entspricht in der Bildschirmdarstellung genau einem Inch. Abweichungen können dabei durch die physikalische Auflösung des Bildschirms entstehen, die z.B. auch bei 68 oder 76 dpi liegen kann. Mit dem Rollbalken an der rechten Seite des Fensters kann nach oben oder unten zu einer anderen Seite gesprungen werden. Das Werkzeug in der Abbildung verwaltet über 70 Seiten, so daß die virtuelle Ausgabefläche die Breite einer DIN A4 Seite und die Höhe von mehr als 70 DIN A4 Seiten umfaßt.

[16] Dies ist sicherlich ein wesentlicher Punkt, der in weiteren Untersuchungen in einem sehr viel stärkeren Maße diskutiert werden muß.

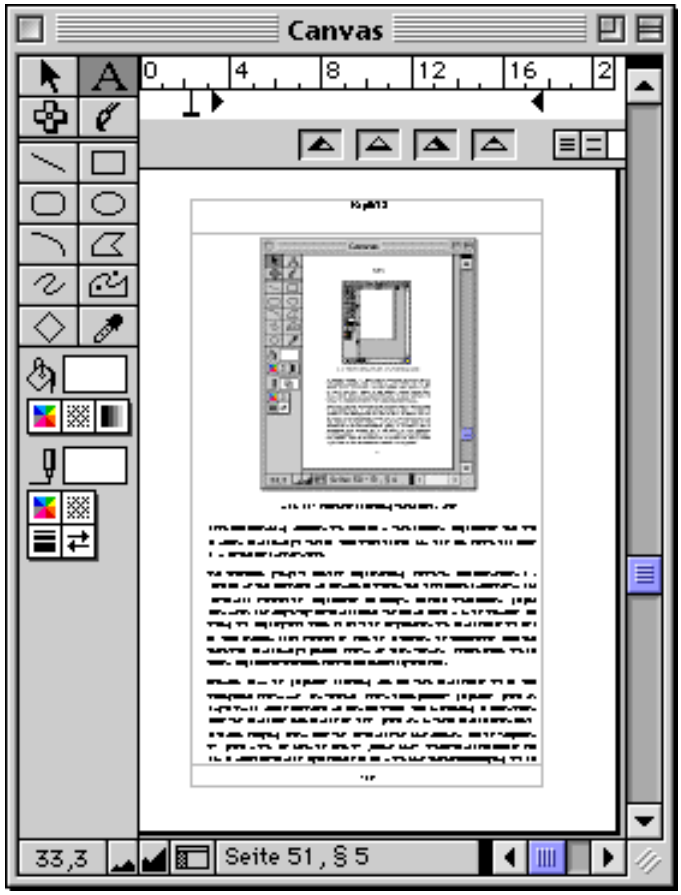


Abb. 28 - Verkleinerte Darstellung einer DIN A4 Seite

Dieser Zusammenhang zwischen der Abstraktion einer virtuellen Ausgabefläche und dem konkreten Canvas-Widget eines Toolkits deutet darauf hin, daß sich wieder das Muster von IAF und PF anwenden läßt.

Und tatsächlich gelingt es, für diese Aufgabenteilung wieder ein Zusammenwirken von Interaktions- und Präsentationsformen zu modellieren und dieses Muster anzuwenden. Eine IAF-Canvas verwaltet die Ausgabefläche im richtigen Maßstab der fachlichen, graphischen Werte. Eine zugehörige PF-Canvas führt eine Transformation in die technische Auflösung des Ausgabegerätes durch, wobei es die Möglichkeiten der Canvas-Klasse des GUI-Toolkits benutzt. Dabei verwaltet sie auch die Rollbalken, die üblicherweise rechts und unter dem Canvas-Widget plaziert werden. Sie lassen erkennen, welcher Bereich der virtuellen Ausgabefläche tatsächlich auf dem Bildschirm dargestellt wird.

Technisch kann die graphische Darstellung nur mit einer Canvas-Klasse des Toolkits durchgeführt werden. Wie oben erwähnt, werden dafür primitive, graphische Operationen angeboten. Da nur die Präsentationsformen mit dem Toolkit in Berührung kommen dürfen, muß eine Canvas-PF ihrer Canvas-IAF diese Operationen in einem standardisierten Protokoll zur Verfügung stellen. Muß eine IAF-Canvas eine Linie zeichnen, ruft die entsprechende Operation der IAF intern die Methode „Zeichne Linie“ der PF-Canvas-Basisklasse auf.

Ein konkretes PF-Canvas-Objekt setzt die Position der Linie unter Berücksichtigung des aktuellen Vergrößerungmaßstabes in das Koordinatensystem des Ausgabegerätes um. Dabei wird geprüft, ob diese Position überhaupt in den gerade angezeigten Teilbereich der virtuellen Ausgabefläche im Canvas-Widget fällt. Dann ruft die konkrete PF die entsprechende Operation des Toolkits auf, um die Ausgabe auf dem Bildschirm durchzuführen[17].

Dieses Vorgehen birgt den bekannten Nachteil der „Virtual Toolkits“ in sich. Die Graphik-Operationen, die eine IAF Canvas anbieten kann, müssen auch von möglichst vielen Toolkits bzw. Fenstersystemen umgesetzt werden können. Anders formuliert kann die Menge der Graphik-Primitiven nur die Schnittmenge der Operationen sein, die in den bekannten GUI-Toolkits angeboten werden. Sicherlich darf man sich dabei nicht nur auf einige wenige Funktionen beschränken, da dies in vielen Fällen zu recht „schlichten“ Benutzungsschnittstellen führt, die von den Anwendern oft nicht angenommen werden. Auf der anderen Seite sollte man aber möglichst auf die ganz ausgefeilten Möglichkeiten eines unterstützten Toolkits verzichten, um den Aufwand bei der Einbindung eines anderen GUI-Toolkits in Grenzen zu halten bzw. den Austausch des Toolkits überhaupt erst zu ermöglichen.

5.2 Darstellung durch Views

Damit wurde eine Möglichkeit beschrieben, mit der eine IAK über eine IAF-Canvas graphische Ausgaben durchführen kann, ohne ein konkretes Toolkit zu benutzen. Unidraw hat aber gezeigt, daß komplexe Graphiken nicht komplett von einer Instanz, in diesem Fall wäre dies die IAK, verwaltet werden müssen. Als Hilfsmittel wurde das Konzept der Views eingeführt, das ja auch schon in Smalltalk und in MacApp benutzt wurde. View-Klassen werden mittlerweile in fast allen GUI-Toolkits angeboten, die graphische Ausgabe und Manipulation ermöglichen.

Komplexe Zusammenhänge in einer Graphik werden so in einfachere Bausteine aufgeteilt, den Views. Dabei können Views auch hierarchisch angeordnet werden. Eine komplette Graphik entspricht einer View, die sich in einfachere Sub-Views aufteilt. Diese Sub-Views werden von ihrer einbettenden View baumartig verwaltet. Übertragen auf die WAM-Metapher könnte dies bedeuten, daß ein komplexes Material der gesamten Graphik entspricht und sich in fachliche Werte unterteilen läßt, die durch einzelne Views dargestellt werden. Diese Views können dabei wieder Sub-Views beinhalten, die einzelne Aspekte eines Wertes darstellen. Betrachtet man die Graphik selber als View, ist sie die Wurzel einer großen View-Hierarchie.

[17] So läßt sich auch der „Trick“ aus vielen bekannten GUI-Toolkits verwirklichen, die graphische Ausgabe nicht nur auf dem Bildschirm vorzunehmen, sondern auch eine Umrechnung in den Maßstab eines angeschlossenen Druckers durchzuführen. Eine spezielle PF benutzt dann die Toolkit-Klassen für die Druckausgabe, so daß ohne großen Aufwand das Drucken von Graphiken implementiert werden kann.

Die IAK erzeugt die View-Objekte und übergibt jeweils einen Fachwert, der visualisert werden soll. Für jeden speziellen Fachwert muß ein konkreter View-Typ definiert werden. Für die Erzeugung der Views in Abhängigkeit der fachlichen Werte würde sich eine Umsetzung des Fabrik-Musters (vgl. „Factory Pattern“ in [GHJ+94]) anbieten. Die Position und Größe der Views in der Graphik können die Views von ihren fachlichen Werten erfahren. Andererseits kann die IAK auch bei der Initialisierung der Views die Position und Ausmaße vorgeben, die die die Views dann für den späteren Gebrauch speichern müssen.

Durch dieses Vorgehen wird die IAK von etlichen Verwaltungsaufgaben bei der Darstellung „großer“ Materialien in komplexen Graphiken entlastet. Sie benutzt lediglich eine Interaktionsform Canvas und muß die View-Objekte verwalten. Die Wurzel der View-Hierarchie übergibt sie der IAF-Canvas, die alles weitere für die Darstellung übernimmt. Auch wird die Darstellung der Graphik nun nicht mehr von der IAK initiiert. Dies steuert vielmehr das GUI-Toolkit, das vom Fenstersystem ein sogenanntes „Redraw Event“ mit der Aufforderung bekommt, jedes Widget neu zu zeichen. Das Ereignis wird über einen - hier nicht näher betrachteten - Callback-Mechanismus an die PF Canvas weitergereicht, die ihrerseits die IAF Canvas aufruft, um die Graphik neu zu zeichnen. Die IAF iteriert über die Baumstruktur der Views und fordert jede View auf, die in den sichtbaren Bereich des Canvas-Widget fällt, sich neu zu zeichnen. Dafür muß jede konkrete View-Klasse eine Einschubmethode `Draw` implementieren, die in der virtuellen View-Basisklasse des Rahmenwerkes definiert wurde. Der View wird in der Methode `Draw` die entsprechende IAF Canvas als Parameter übergeben, damit sie überhaupt die Graphikoperationen aufrufen kann.

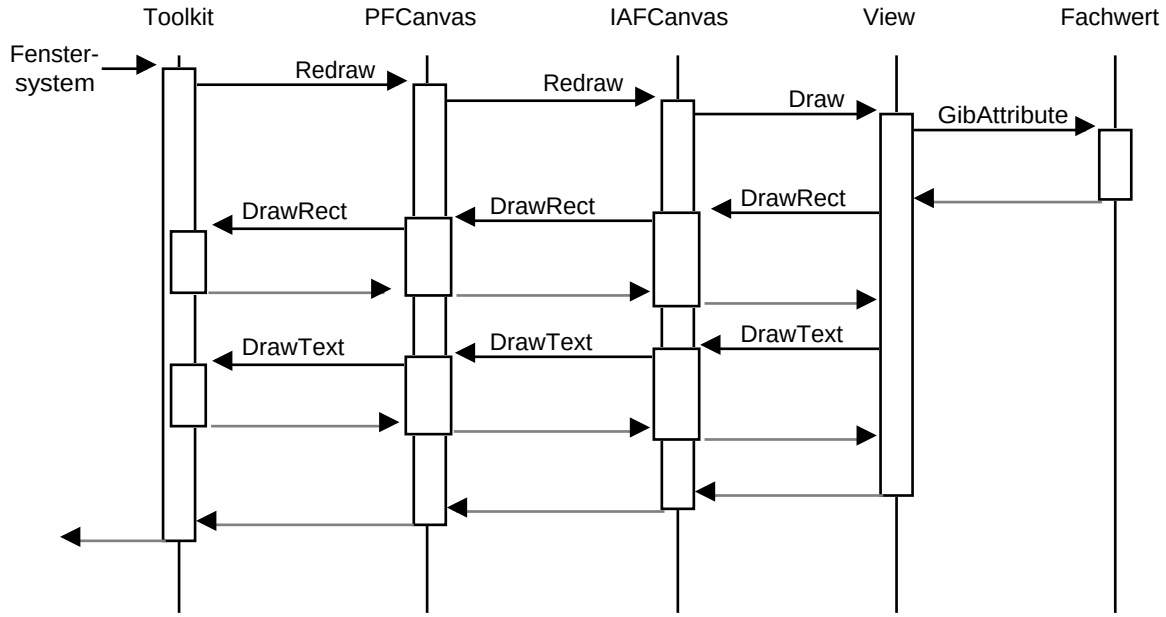


Abb. 29 - Darstellung einer Graphik, ausgelöst vom Fenstersystem

Änderungen am Material werden wieder von der Funktionskomponente an die IAK gemeldet. Um das Material in der graphischen Darstellung zu aktualisieren, muß die Interaktionskomponente ermitteln, welche Views die geänderten Werte darstellen und diese auffordern, sich neu zu zeichnen. Dies geschieht aber wieder nicht direkt, sondern die IAK teilt dem IAF Canvas mit, welche Views neu gezeichnet werden müssen. Das Rahmenwerk durchläuft wieder die View-Hierarchie und prüft, ob auch andere Views ebenfalls neu gezeichnet werden müssen, z.B. wenn sich eine eigentlich unveränderte View mit der View eines geänderten Wertes überschneidet. Alle Views, die neu gezeichnet werden müssen, werden der PF Canvas gemeldet. Fällt eine View in den gerade sichtbaren Darstellungsbereich des Canvas-Widget, wird dem Fenstersystem mitgeteilt, daß dieser Bereich nun ungültig ist. Das Fenstersystem wird daraufhin wieder ein Redraw Event generieren, so daß die „ungültigen“ Views nach dem in Abb. 29 dargestellten Vorgehen neu gezeichnet werden.

5.3 Graphisch manipulieren

Materialien graphisch darzustellen ist aber nur der erste Schritt. In vielen Anwendungsfällen müssen Werkzeuge dem Benutzer auch die Möglichkeit bieten, Materialien graphisch zu erstellen und zu bearbeiten. Im vorhergehenden Abschnitt wurde die Darstellung von Fachwerten durch View-Objekte beschrieben. Auch bei der Manipulation werden die Views eine wesentliche Rolle spielen. Dabei soll bereits jetzt noch einmal betont werden, daß die Aufgabentrennung zwischen FK und IAK auch weiterhin aufrechterhalten bleibt. Änderungen am Material kann nur die FK durchführen, die IAK ist nach wie vor nur für die Präsentation und Interaktion mit dem Werkzeug zuständig.

Mit Bezug auf die Entwurfsmuster von Unidraw soll für die Interaktionsformen das Konzept der „Manipulatoren“ eingeführt werden. Dies läßt sich wieder sehr anschaulich am Beispiel eines gängigen Zeichenprogrammes erklären. Zeichenprogramme bieten üblicherweise Funktionen an, in denen direkt in einer Canvas graphische Objekte (z.B. Linien, Rechtecke oder Kreise) erzeugt und manipuliert werden können. Ein Rechteck wird z.B. erzeugt, indem mit dem Mauszeiger die Position der linken oberen Ecke definiert und bei gedrückter Maustaste das Rechteck mit dem Mauszeiger in der Art eines Gummibandes aufgezogen wird.

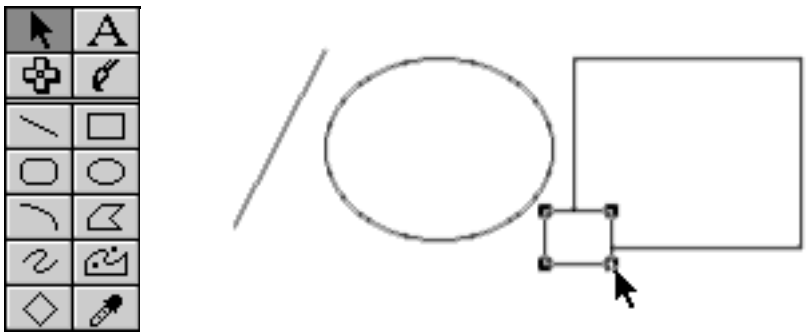


Abb. 30 - Werkzeugpalette und Darstellung in einem Zeichenprogramm

Die Funktionen, mit denen die unterschiedlichen Objekte erzeugt und manipuliert werden, können dabei in einer sogenannten „Palette“ ausgewählt werden, die das Graphikwerkzeug über eine Interaktionsform „Selector“ ansprechen kann. Die einzelnen Graphikobjekte werden wie gehabt durch Views auf einer IAF-Canvas gezeichnet.

Veränderungen an den Objekten können ebenfalls mit der Maus durchgeführt werden. Mit einem Klick auf ein Objekt wird die darstellende View als markiert gekennzeichnet. Das bedeutet, daß die View nicht nur ihren zugehörigen Wert zeichnet, sondern auch die Selektion anzeigen muß. In Abb. 30 kann man vier sogenannte „Handle“ erkennen, das sind die kleinen schwarzem Quadrate an den Ecken des Rechteckes neben dem Mauszeiger. Dadurch wird angezeigt, daß das Objekt manipuliert werden kann. In der Palette in Abb. 30 ist eine Funktion aktiviert, mit der es möglich ist, die Position und die Maße des selektierten Rechteckes zu verändern. Nun hätte die IAK die Möglichkeit, ihre FK aufzurufen, um den Wert zu manipulieren, der zu der selektierten View gehört. Z.B. könnte ein Kasten in einem Organigram um 2 cm nach unten bewegt werden. Die FK würde der IAK diese Änderung melden, die daraufhin wie beschrieben die Views veranlaßt, das Material neu darzustellen.

Manipulatoren

Solche Änderungen am Material könnten durch Aktivatoren ausgelöst werden, die über einen Menüpunkt in der Benutzungsschnittstelle realisiert werden und der IAK über ein Kommando melden, wann die Änderung durchgeführt werden soll. Stand der Kunst ist es aber, direkt mit der Maus die Manipulationen durchzuführen und dabei eine visuelle Rückkoppelung zu erhalten. Dieser Vorgang wird auch „direkte Manipulation“ genannt, für den Unidraw die „Manipulatoren“ etabliert hat.

Um z.B. das Objekt einer View zu verschieben, geschieht folgendes: der Anwender klickt mit der Maus in die View und hält die Maustaste gedrückt. Der Anwender bewegt den Mauszeiger an die Position, an der das Objekt plaziert werden soll, und läßt die Maustaste wieder los. Dabei erhält er die ganze Zeit eine visuelle Rückkoppelung, indem der Umriß des zu bewegenden Objektes unter der aktuellen Position des Mauszeigers dargestellt wird.

Den Mausklick kann nur die PF-Canvas erkennen, die das Canvas-Widget des Toolkits benutzt, so daß sie die Distanz, über die das Objekt verschoben werden soll, über die Position des Mauszeigers ermittelt. Danach muß dieser Vektor in die Koordinaten der IAF Canvas transformiert werden, damit die betreffende View gefunden werden kann. Damit ein Manipulator die Veränderungen vornehmen kann, muß er also mit beiden Kontexten der Präsentations- und der Interaktionsformen verbunden sein. Es bietet sich also wieder eine Aufteilung in IAF und PF an.

In unserem Beispiel würde eine Präsentationsform Manipulator erkennen, daß ein Mausklick in den Bereich einer PF Canvas erfolgte. Dazu muß sich dieser Manipulator bei der PF-Canvas für das Ereignis Mausklick registrieren. Die PF-Manipulator meldet ihrer IAF den Klick, die nun ermitteln muß, ob und welche View betroffen ist. Die IAF markiert die

View als selektiert und veranlaßt über die IAF-Canvas, daß die Selektion auch angezeigt wird. Danach kehrt der Kontrollfluß zur Präsentationsform und damit zum Toolkit zurück. Wird die Maus nun bei gedrückter Maustaste bewegt, meldet die PF-Manipulator dies wieder ihrer Interaktionsform, die daraufhin eine temporäre View erzeugt, um die visuelle Rückkoppelung darzustellen. Während der Anwender die Maus bewegt, stellt die PF Manipulator fest, um welche Distanz das Objekt bisher verschoben werden soll. Die temporäre View wird im Koordinatensystem der IAF-Canvas verschoben und laufend neu gezeichnet. Während dieses Vorganges könnte die IAF-Manipulator auch bei der FK anfragen, ob die Manipulation jeweils zulässig wäre, wenn der Benutzer an der aktuellen Position den Mausknopf loslassen würde. Diese Information müßte die temporäre View zusätzlich anzeigen.

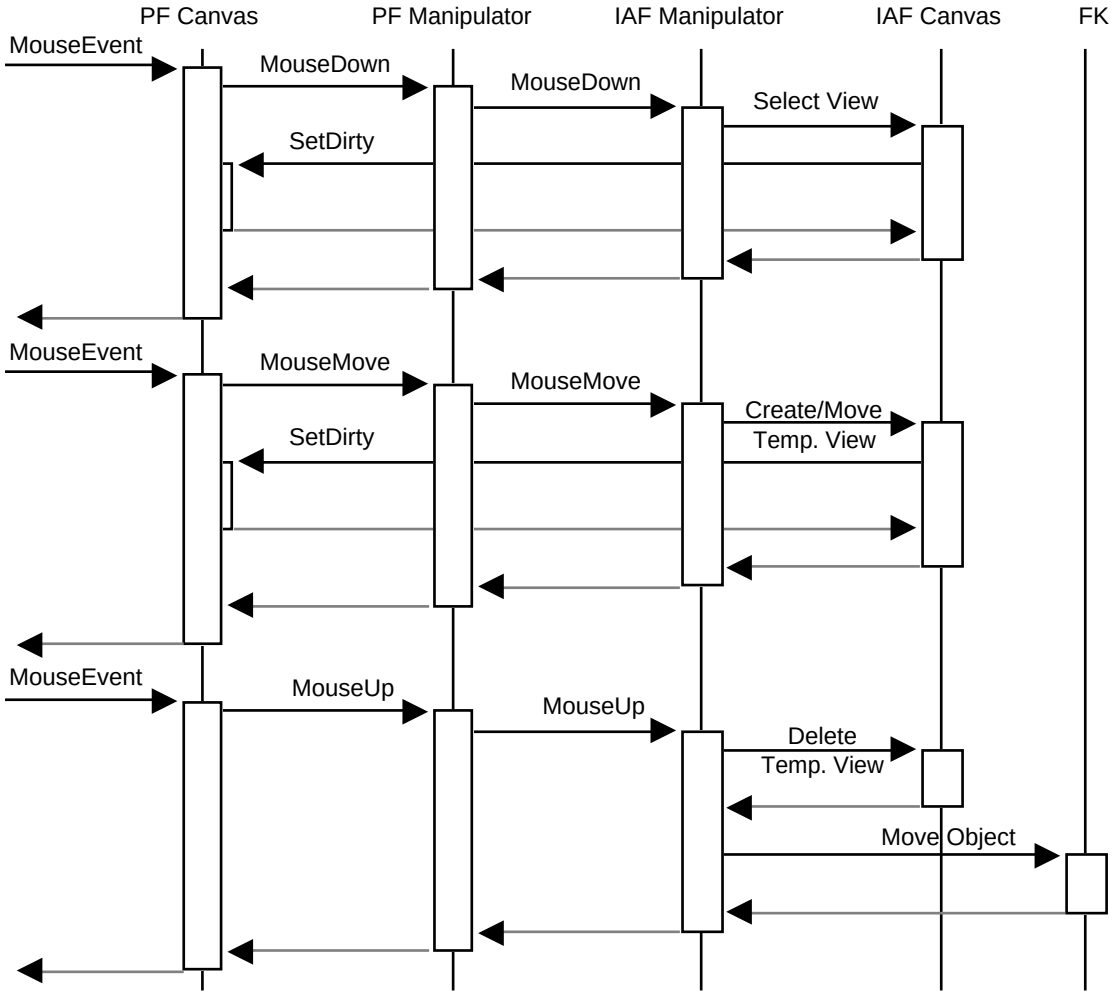


Abb. 31 - Zusammenspiel von IAF und PF bei der direkten Manipulation

Wird der Mausknopf losgelassen, ist dies das Zeichen dafür, daß das Werkzeug die Veränderung endgültig vornehmen soll. Diese Änderung darf nur die FK durchführen. Die Interaktionsform Manipulator bewirkt einen Aufruf der FK, die in diesem Fall die Position des Objektes neu festsetzt. Nur die FK kann erkennen, ob diese Veränderung fachlich überhaupt

zulässig ist, und läßt den Zustand des Objektes ggf. auch unverändert. Konnte die Manipulation jedoch durchgeführt werden, wird der IAK die Änderung gemeldet. Sie veranlaßt daraufhin, daß die betreffenden Views neu gezeichnet werden.

Dieses Beispiel hat nur eine einzige Form von Manipulatoren vorgestellt. Typischerweise werden noch viele andere Manipulatoren benutzt, um beispielsweise Objekte zu erzeugen oder in der Größe zu verändern. Denkbar sind auch Manipulatoren, die die Beschriftung oder die Farbe eines Objektes ändern können. Darüber hinaus ist es auch möglich, daß eine IAF Manipulator mit mehreren Präsenationsformen verbunden wird. Im unserem Exempel könnte sich jeweils eine PF Manipulator für Mausereignisse und eine andere für Tastaturereignisse registrieren. So hätte der Anwender die Möglichkeit, ein selektiertes Objekt auch mit den Cursortasten zu verschieben. Die IAK würde dies aber nicht betreffen, da sie nur eine IAF Manipulator „Verschieben“ benutzt.

Dies ist das erste Mal, daß zwei Präsentationsformen und zwei Interaktionsformen zusammenwirken. Es besteht jeweils eine Beziehung zwischen IAF- und PF-Canvas bzw. IAF- und PF-Manipulator. Aber auch zwischen PF-Canvas und PF-Manipulator und zwischen IAF-Manipulator und IAF-Canvas kommt es zu einer Verbindung. Dies deutet darauf hin, daß Canvas und Manipulator vielleicht zusammen modelliert werden sollten. Auf der anderen Seite wird ja gerade dadurch eine Flexibilität erreicht, daß eine IAK eine Interaktionsform Canvas mit speziellen IAF-Manipulatoren verbinden kann, und damit gezielt ihre fachlichen Anforderungen umsetzen will.

Einer Interaktionskomponente müssen also spezialisierte Klassen zur Verfügung gestellt werden, die von `IAFManipulator` und `View` abgeleitet wurden. Die Klasse `IAFCanvas` kann direkt benutzt werden. Auf Seiten der Präsentationsformen stellt das Rahmenwerk zahlreiche, von `PFManipulator` abgeleitete Klassen zur Verfügung. So kann mit ihnen z.B. eines Rechteck „aufgezogen“ werden, Objekte können mit der Maus oder den Cursortasten verschoben und in der Größe verändert werden oder der Anwender kann eine Beschriftung neu setzen.

Um diesen Abschnitt nicht zu umfangreich zu gestalten, soll an dieser Stelle die Beschreibung der Einbindung von Graphik in die Architektur von Interaktions- und Präsentationsformen enden. Im folgenden Kapitel wird ein Interface Builder für Präsentationsformen vorgestellt, der genau nach diesem Muster entwickelt wurde, so daß dann noch einmal auf diese Thematik eingegangen wird.

In einer kurzen Bewertung dieses Musters der graphischen IAF und PF soll wieder der Vergleich mit dem Vorbild Unidraw gezogen werden. Unidraw hat sich für die Realisierung umfangreicher Graphikeditoren bewährt (vgl. [VL90]). Dies muß der hier beschriebene Entwurf erst noch beweisen. Prinzipiell wird aber das Rüstzeug geboten, um zumindest kleine, fachlich motivierte graphische Werkzeuge im Sinne von WAM zu entwickeln. Bewährte Konzepte wurden so erweitert, daß dabei zumindest ein gefordertes Merkmal der Interaktionsformen, die Unabhängigkeit von einem GUI-Toolkit, eingehalten werden kann.

6 Beispiele der konkreten Umsetzung

Das Architekturmuster von IAF und PF stellt eine Weiterentwicklung des Musters der Interaktionstypen dar. Die Interaktionstypen wurden schon mehrfach in konkreten Klassenbibliotheken erprobt, das WAM-Rahmenwerk „Bib V3.0“ mit seiner „IAT Motif“ Bibliothek wurde hier schon mehrfach erwähnt. Natürlich müssen auch die Interaktions- und Präsentationsformen erst einmal im realen Einsatz zeigen, daß sie für praxisrelevante Projekte eingesetzt werden können. Das größte Problem zeigt sich wohl darin, daß durch die erweiterte Abstraktion die Entwickler mit noch komplexeren Konzepten konfrontiert werden.

6.1 Einbindung in die Bib V3.0

Die erste Anforderung, die ich an meine Implementation der Interaktions- und Präsentationsformen in C++ gestellt habe, bestand darin, daß die IAF und PF sich in das existierende Rahmenwerk am Arbeitsbereich Softwaretechnik integrieren lassen. Ich wollte zeigen, daß die Interaktionstypen komplett ersetzt werden können. Das Rahmenwerk ist in mehrere Subrahmenwerke aufgeteilt, so daß ich am Anfang untersuchen mußte, welche Klassen eventuell ausgetauscht oder lediglich angepaßt werden mußten.

Vor allem im FIAK-Application Framework (vgl. [Wei97]) wird davon ausgegangen, daß ein Werkzeug mit Interaktionstypen arbeitet. Dies spiegelt sich u.a. in der Basisklasse für Interaktionskomponenten wider, in der als Parameter für den Konstruktor ein sog. Kontext-IAT übergeben wird (Abb. 32).

Mit diesem IAT-Kontextobjekt wird die hierarchische Verknüpfung der Widgets erreicht, die durch die Interaktionstypen erzeugt werden. In meiner Lösung übergibt eine erzeugende IAK ein Objekt vom Typ `IAFsContext`. Dieses muß im Konstruktor einer Sub-IAK benutzt werden, um als Parameter bei der Erzeugung des eigenen IAF-Kontextobjektes zu dienen und intern die Anbindung der Präsentationsformen durchzuführen (vgl. Kap. 4). Diese ganze Verwaltung, zu der z.B. auch die Erzeugung des PF-Kontextobjektes und der Präsentationsformen gehört, wird im IAF-PF-Rahmenwerk von einer zentralen Instanz durchgeführt, dem sogenannten „GUI-Manager“ (vgl. Abschnitt 6.2).

```
class tIAKBase : public tObserver, public tObservable
{
  public:

    virtual tFKBase* GetFK        ( );
    virtual void SystemUp         ( );
    virtual ~tIAKBase             ( );

    [...]

  protected:

      // hier muessen alle IATs geloescht werden,
      // damit die IAK vom Bildschirm verschwindet!

      virtual void DoTerminateIATs ( );

      // erzeugt beim Zusammenbauen eines neuen Werkzeugs alle SubIAKs

      virtual void CreateSubIAKs   ( );

    // wird bei der Erzeugung von SubIAKs aufgerufen, um der
    // neuen SubIAK den IAT zu uebergeben, in den sie ihre
    // IAT einbetten soll

    virtual tIATContext* GetIAKIAT( tFKBase& );

      // Initialisierung und Zerstoeren der enthaltenen Komponenten

    tIAKBase                      ( tFKBase*, tIATContext* );
    void Init                     ( tFKBase*, tIATContext* );
    void Terminate                ( );

    [...]
};
```

Abb. 32 - Die Basisklasse für Interaktionskomponenten

Außerdem zeigte sich, daß das Toolkit „WxWindows“, welches ich verwenden wollte (vgl. [Sma97]), nicht in das White-Box-Schema der „Bib V3.0“ eingebunden werden konnte. Dieses Toolkit hat die Eigenart, ein C++ Programm bereits vor dem Aufruf von `main` wieder zu beenden. Statt dessen muß ein statisches „Application Object“ definiert werden, das bei seiner Erzeugung den Kontrollfluß vollständig behält und das Programm „hart“ beendet, nachdem der Konstruktor wieder verlassen wurde (Abb. 33). So muß der Werkzeugbaum mit den Klassen der „Bib V3.0“ innerhalb des Wurzelobjektes von „WxWindow“ erzeugt werden.

```
class IAFwxApplication : public wxApp
{
  public:

      IAFwxApplication( ) : wxApp( )
      {
         environment = new tEnvironment( );
         guiManager = new IAFwxGUIManager( this );
      }

      wxFrame *  OnInit( )
      {
         environment -> CreateMainTool( wxApp::argc, wxApp::argv );

         return guiManager -> GetMainWindow( );
      }

      virtual ~IAFwxApplication( )
      {
         delete guiManager;
         delete environment;
      }

  protected:

      tEnvironment *      environment;
      IAFwxGUIManager *   guiManager;
};
```

Abb. 33 - Application-Klasse für den Einsatz von „BibV3.0“

Da ich nicht gegen das Toolkit programmieren wollte (mit einigen Tricks hätte dieses Problem auch „gehackt“ werden können), möchte ich als konzeptionelle Erweiterung unseres WAM-Rahmenwerkes eine Vorgehensweise vorschlagen, wie Werkzeuge abhängig(!) vom Toolkit gestartet werden können.

Viele GUI-Toolkits stellen sich als Application Framework in den Mittelpunkt der Anwendungsarchitektur und haben eine eigene Ansicht darüber, wann und wie sie initialisiert werden müssen. Oft ist es leider nicht möglich, für unser Rahmenwerk eine allgemeine Wurzelklasse für den Werkzeugbaum zu entwerfen, in der Einschubmethoden die Initialisierung des Toolkits komplett steuern können. Statt dessen sollte man ganz auf die Eigenarten des jeweiligen Toolkits eingehen und speziell für jedes Toolkit einmal eine konkrete Wurzelklasse oder eine Funktion „main“ schreiben, die das Toolkit nach seinen Vorstellungen aufruft. Dabei muß an einer geeigneten Stelle das Environment-Objekt erzeugt werden, das seinerseits den Werkzeugbaum mit all seinen Objekten erzeugen kann (vgl. [Wei97]). Diese Wurzelklasse (bzw. Funktion „main“), die speziell auf das GUI-Toolkit zugeschnitten ist, kann dann in jedem Programm, das unser WAM-Rahmenwerk zusammen mit dem Toolkit benutzt, per „Copy-Paste-Reuse“ eingebunden werden. Der Einstieg in unser Rahmenwerk erfolgt, indem das Environment-Objekt erzeugt wird.

Dieses Vorgehen entspricht einer Art „defensiver Programmierung“, wie ich sie in einem der vorherigen Kapitel schon einmal angesprochen habe. Es werden eigentlich keine Annahmen darüber getroffen, mit welcher Aufrufreihenfolge ein Toolkit eingebunden werden kann. WxWindows hat gezeigt, daß man mit diesem Vorgehen eher auf die Eigenheiten der verschiedenen GUI-Toolkits vorbereitet ist.

6.2 Ein Editor für Präsentationsformen

Nachdem die Probleme zur Einbindung in das existierende Rahmenwerk mehr oder weniger gelöst waren, sollten nun die Interaktionsformen und die Präsentationsformen, die ich mit WxWindows entwickelt hatte, praktisch erprobt werden. Da von Anfang an auch eine Unterstützung der Entwickler durch einen Interface Builder vorgeschlagen wurde, lag der Gedanke nahe, einen PF-Editor für mein neues IAF-PF-Rahmenwerk zu entwickeln und es dabei auch gleichzeitig in der Praxis zu testen.

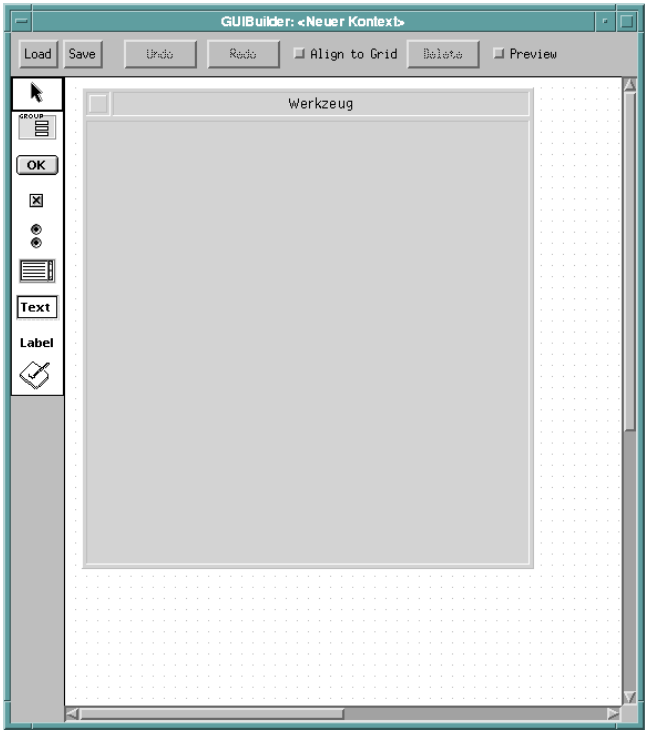


Abb. 34 - Der PF-Editor nach dem Start

Ein Interface Builder, der auch professionellen Ansprüchen genügen würde, wäre im Rahmen meiner Möglichkeiten nicht zu realisieren gewesen. So schraubte ich meine Anforderungen herunter, und stellte eigene Maßstäbe auf, an denen mein Editor gemessen werden sollte. Der Editor kann nur eine kleine Menge von Präsentationsformen erzeugen und bearbeiten, aus diesem Grund werden nur die notwendigsten GUI-Elemente angeboten. Er läßt sich trotzdem in der Art eines professionellen Graphik-Werkzeuges bedienen, z.B. werden Undo und Redo unterstützt. Der Editor kann die Präsentationsformen speichern und wieder

laden, so daß ein Entwickler, der die PF-Klassen für ein eigenes Werkzeuge benutzen will, die Präsentationsformen nicht von Hand einbinden muß. Sie können vom Rahmenwerk eingelesen werden. Die allererste Version des Editors konnte so leider noch nicht implementiert werden, da es ja noch keine Möglichkeit gab, die Präsentationsformen abzuspeichern. Ich mußte am Anfang also für mein Werkzeug eine PF-Kontextklasse schreiben, in der die Präsentationsformen manuell erzeugt und verwaltet werden.

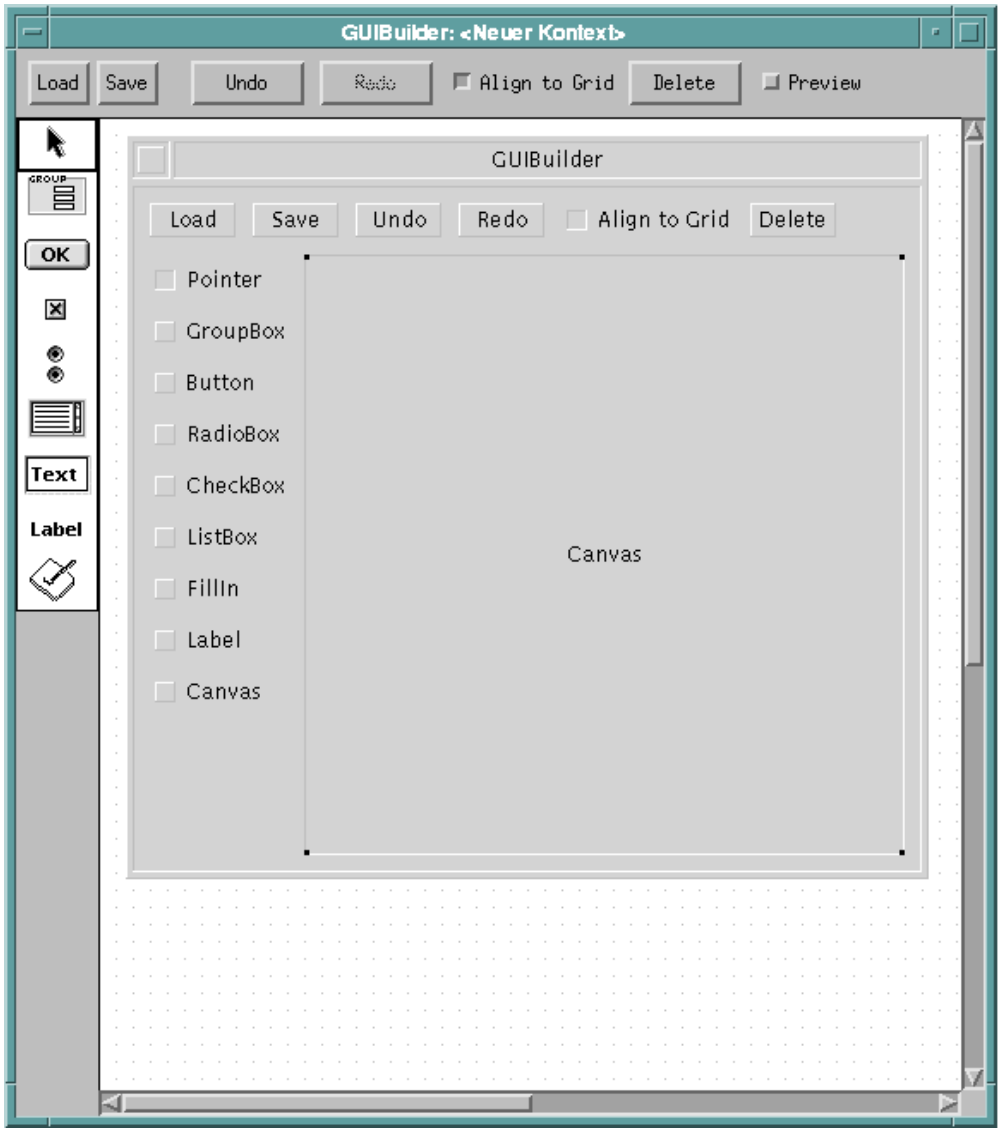


Abb. 35 - Die Präsentationsformen des PF-Editors im PF-Editor

Das Material des Werkzeuges sind Präsentationsformen. Jede Manipulation am Material führt ausschließlich die Funktionskomponente des PF-Editors durch. Dabei handelt es sich aber nicht um die PF-Klassen, die für WxWindows geschrieben wurden. Es sind Präsentationsformen, die mit keinem Toolkit in Verbindung stehen, sondern neben ihrem Typ und den Bezeichnern nur noch graphische Attribute besitzen und sich serialisieren lassen.

Sie dienen dem Rahmenwerk später als eine Art Ressourcenbeschreibung, mit der nach dem Fabrikmuster (vgl. [GHJ+94]) konkrete PF-Objekte erzeugt werden können. Bei der Erzeugung des Werkzeuges, für das diese Präsentationsformen im Editor zusammengestellt wurden, liest das GUI-Manager-Objekt die serialisierten „Ressourcen-PF“ ein und übergibt diese einem Fabrik-Objekt. Das Fabrik-Objekt wertet die Ressourcen aus und interpretiert sie für die Erzeugung der Präsentationsformen.

Die Interaktionskomponente des PF-Editors erzeugt die Interaktionsformen, die mit ihren Präsentationsformen verbunden werden. Letztere realisieren die Oberfläche des Editors (Abb. 34 und 35). Die IAK erzeugt und benutzt fünf Aktivatoren, drei Selektoren und eine IAF-Canvas. Die Aktivatoren melden, wenn der Benutzer das Material laden oder speichern, ein Redo bzw. Undo ausführen oder eine (oder mehrere) selektierte Präsentationsform(en) löschen will. Zwei Selektoren verändern den Zustand des Werkzeuges. Der eine Selektor (mit PF „Align to Grid“) zeigt an, ob die Präsentationsformen an einem Raster ausgerichtet werden sollen. Der zweite (PF „Preview“) signalisiert einen Modus, in dem parallel zum Editor ein Fenster dargestellt wird, das die Präsentationsform in ihrer konkreten Form anzeigt.

Der dritte Selektor zeigt an, welchen Manipulator der Benutzer gerade ausgewählt hat. Die IAK benutzt zwei Typen von Manipulatoren. Einen Manipulator, um die stilisierten Präsentationsformen im Editor zu selektieren und die selektierten PF in ihrer Position und Größe zu verändern („Pointer“); und einen zweiten Manipulator, um die Präsentationsformen überhaupt erst einmal zu erzeugen. Dieser zweite Typ kann parametrisiert werden, so daß nicht für jede Präsentationsform extra ein eigener, erzeugender Manipulator geschrieben werden muß.

Der Anwender kann einen erzeugenden Manipulator benutzen, indem er mit der Maus ein Rechteck aufzieht, das die Position und Ausmaße der neuen Präsentationsform bestimmt. Während dieses Vorganges wird eine temporäre View erzeugt, die das Rechteck darstellt. Nachdem der Benutzer den Mausknopf losgelassen hat, erzeugt der Manipulator ein Kommando, das signalisiert, daß nun eine Präsentationsform erzeugt werden soll. Die IAK empfängt das Kommando und erfragt bei der FK, ob an der angegebenen Position ein PF-Composite als Eltern-Objekt exisitiert. Ist dies der Fall, kann die neue Präsentationsform erzeugt werden.

Die Interaktionskomponente ruft dafür aber nicht direkt die FK auf. Sie erzeugt vielmehr ein weiteres Kommando, das einem Undo-Protokoll folgt, also die Methoden „Do“, „Undo“ und „Redo“ besitzt. Dieses Kommando ruft in der Methode „Do“ die FK auf, die an der angegebenen Position ein neues PF-Objekt erzeugt und mit dem Eltern-Objekt verknüpft. Die FK meldet der IAK den geänderten Zustand des Materials, die für die neue Präsentationsform eine neue View anlegt. Das Kommando, das die FK aufgerufen hat, wird in eine Command-Queue gestellt. So kann die IAK diesen Vorgang rückgängig machen, indem sie die Methode „Undo“ aufruft und mit der Methode „Redo“ auch wieder erneut ausführen. Wenn

die Command-Queue alle Kommandos hält, kann dieses „Undo“ und „Redo“ auch beliebig oft wiederholt werden[18]. Bedingung ist nur, daß die IAK sämtliche Aufrufe der FK, die den Zustand des Materials verändern können, indirekt über solche Kommando-Objekte vornimmt, die alle in die Command-Queue gestellt werden müssen.

Die Präsentationsformen bzw. Widgets werden mit dem IAF-Canvas durch graphische Elemente dargestellt. Für jeden PF-Typ mußte eine eigene View-Klasse geschrieben werden, die jeweils eine Präsentationsform stilisierend zeichnet.

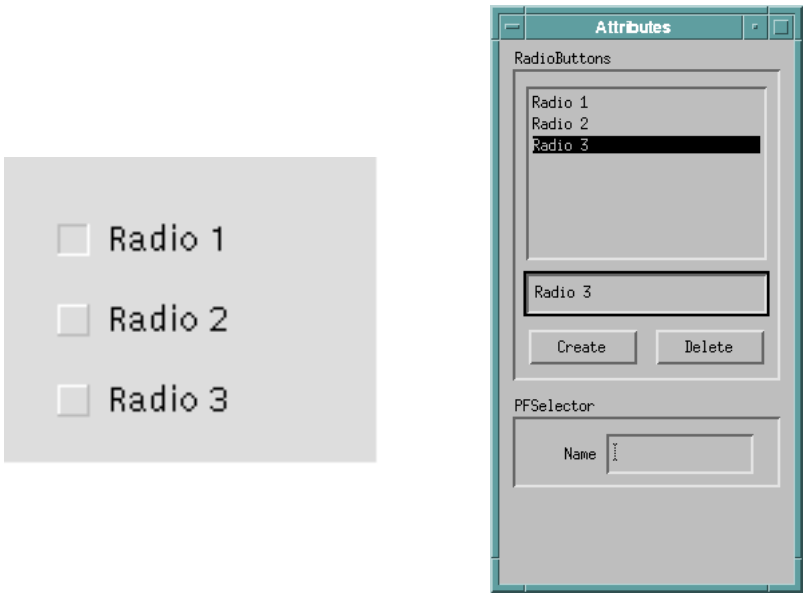


Abb. 36 - Palettenwerkzeug für eine Gruppe von Radiobuttons

Ein Klick mit der rechten Maustaste in das View-Objekt einer Präsentationsform bewirkt, daß der PF-Editor ein Subwerkzeug startet, mit dem einzelne Attribute der Präsentationsform bearbeitet werden können. Um die entsprechende Präsentationsform, also das Material für das Subwerkzeug zu ermitteln, kann die IAK vom View-Objekt die Präsentationsform erfragen. In Abb. 36 wird ein Werkzeug dargestellt, mit dem eine Gruppe von Radiobuttons, die durch die Klasse `PFRadioButtons` realisiert wird, bearbeitet werden kann. Der Typ `PFRadioButtons` wurde von PF-Selector abgeleitet, so daß im Palettenwerkzeug auch der Bezeichner eingegeben werden muß, mit dem die Verbindung zwischen PF- und IAF-Selector hergestellt werden kann. Das Palettenwerkzeug selber ist übrigens ein praktisches Beispiel dafür, daß die Präsentationsformen eines Subwerkzeuges in der PF-Hierarchie des einbettenden Werkzeuges - in diesem Fall des PF-Editors - eingeordnet werden. Das Palettenfenster gehört nämlich zum einbettenden PF-Editor. Im Palettenwerkzeug wird wieder ein spezielles Subwerkzeug benutzt, um den Namen des Bezeichners zu bearbeiten. Auch die Präsentationsformen dieses Sub-Subwerkzeuges stehen wieder direkt in der PF-Hierarchie des einbettenden Werkzeuges.

[18] Natürlich nur, bis der anfängliche oder der letzte Zustand des Materials erreicht wurde.

Der PF-Editor konnte sehr kompakt gehalten werden. Viele Aufgaben werden durch die Palettenwerkzeuge erledigt. Die aufwendigste Klasse ist der Manipulator, mit dem die View-Objekte der Präsentationsformen selektiert werden. Dieser Manipulator meldet außerdem, daß eine PF in Größe und Lage verändert oder im Palettenwerkzeug editiert werden soll. Dafür mußte eine spezielle IAF-Manipulatorklasse geschrieben werden. Auf der Seite der Präsentationsformen konnten aber ausschließlich vorgefertigte PF-Manipulatoren des Rahmenwerkes eingesetzt werden, die lediglich die Mausereignisse weiterreichen.

6.3 Portabilität

Die Präsentationsformen, die in meiner ersten Implementation das GUI-Framework WxWindows benutzen, können wie das gesamte WAM-Rahmenwerk nur unter Unix (Sun Solaris) zusammen mit dem Toolkit Motif benutzt werden. WxWindows ist in einer Version für das Apple MacOS verfügbar und liegt auch für Windows vor, für das es sogar ursprünglich entwickelt wurde.

Da der Quellcode unseres Rahmenwerkes vorliegt, war meine erster Gedanke, für eine Portierung einfach alle Klassen in einem Entwicklungssystem für den Macintosh zu rekompilieren und mit der WxWindows-Version für das MacOS zu binden. Bei diesem Versuch traf ich das erste Mal auf einen Umstand, den ich in den vorherigen Kapiteln immer angemahnt habe: WxWindows erwies sich auf dem Macintosh als nicht akzeptabel (Abb. 37)! Unter anderem wurden die graphischen Klassen überhaupt nicht unterstützt und der gesamte Eindruck der Werkzeuge hätte nicht dem Look and Feel des Macintosh entsprochen. Abgesehen von den Problemen, die ich bei der Portierung der Bib V3.0 hatte, mußte ich also das komplette GUI-Toolkit gegen ein anderes austauschen. Hätte ich für meinen Interface Builder die Klassen von WxWindows in den Mittelpunkt meiner Architektur gestellt, wäre ich in eine Abhängigkeit gekommen, die ich spätestens an dieser Stelle bereuen würde.

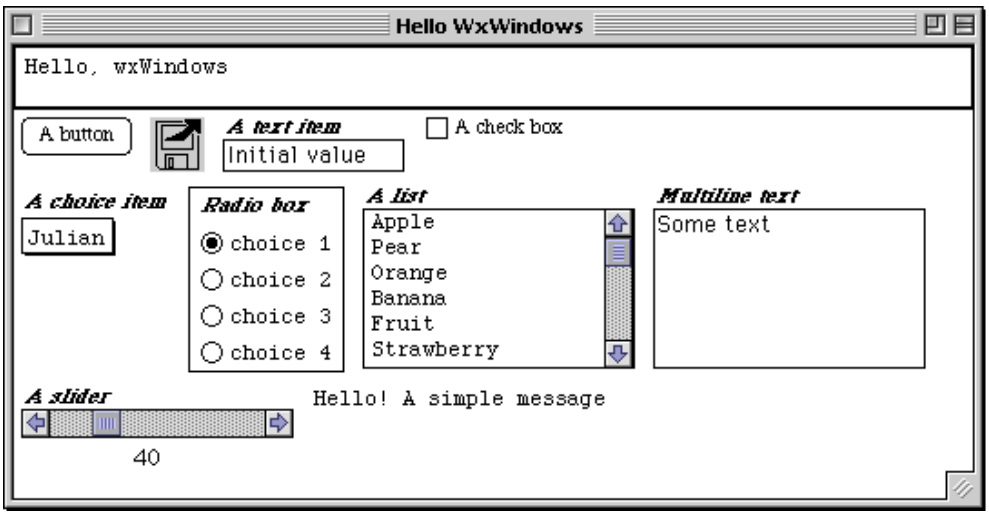


Abb. 37 - Ein Beispielprogramm für WxWindows auf dem MacOS

Als Ausweg versuchte ich nun, eine neue Implementation meiner Präsentationsformen vorzunehmen, bei der ein anderes Toolkit als WxWindows zum Einsatz kommen mußte. Dafür habe ich ein eigenes, rudimentäres Toolkit entwickelt, das auf dem API des MacOS, der sog. Toolbox beruht. So ist es mir gelungen, den PF-Editor und den Aufgabenverwalter, der in Kapitel 4 vorgestellt wurde, auf das MacOS zu portieren (siehe Abb. 38). Der PF-Editor hat in dieser Version allerdings noch einen eher prototypischen Charakter. Dabei konnten die FK- und die IAK-Klassen unverändert übernommen werden. Für die Serialisierung der Präsentationsformen gelang die Portierung aber nicht, so daß die Ankoppelung der PF-Objekte an die Werkzeuge von Hand nachprogrammiert werden mußte. So konnte ich natürlich eher auf das Look and Feel des Macintosh eingehen, z.B. wurden die Menüleisten gemäß der Vorgaben im Styleguide zusammengestellt. Außerdem zeigen sich die Werkzeuge auch im Verhalten nun mehr wie eine Macintosh-Anwendung. Mit WxWindows hätten die Werkzeuge ihre Abstammung wohl nie ganz verleugnen können.

Abb. 38 - Der portierte Aufgabenverwalter

Kapitel 6

Die Erfahrungen, die ich bei der Portierung zweier Werkzeuge auf eine andere Plattform gemacht habe, haben mich einiges gelehrt. So wird es meiner Meinung nach wohl nie möglich sein, durch ein einfaches Rekompilieren eine Anwendung so zu portieren, daß sie von Anfang an in der gewünschten Form benutzt werden kann. Technisch erscheint es wohl möglich, sofort eine lauffähige Version zu erhalten. Java z.B. bietet dafür alle Voraussetzungen. Im Detail wird man damit aber oft nicht zufrieden sein. Auch hier kann Java als Beispiel gelten, diesmal aber nicht im positiven Sinne. Wenn man das GUI-Toolkit von Java („AWT“) betrachtet, scheint dessen Entwurf an vielen Stellen nur aus schlechten Kompromissen zu bestehen. U.a. hat es das Java-System bis heute nicht geschafft, einen plattformunabhängigen Mechanismus für das Drucken anzubieten.

Das Bestreben der Entwickler muß es also sein, die Architektur der Rahmenwerke und damit der Werkzeuge so zu gestalten, daß die Komponenten eines Anwendungssystemes lose gekoppelt werden können. Nur so können notwendige Änderungen, wie sie z.B. bei der Portierung von Werkzeugen vorkommen, mit vertretbarem Aufwand durchgeführt werden.

7 Abschluß

In dieser Arbeit habe ich die verschiedenen Abstraktionsstufen vorgestellt, die sich für die Entwicklung von Benutzungsschnittstellen anbieten. In evolutionärer Weise entwickelten sich die prozeduralen API der Fenstersysteme über Funktions- und Klassenbibliotheken bis hin zu den z.T. äußerst komplexen GUI- und Application-Frameworks, die häufig sogar die Architektur einer Anwendung vorgeben.

Die Architektur der Werkzeug-Automat-Material-Metapher bricht die - zumeist monolithischen - Strukturen dieser Rahmenwerke auf und verfolgt das Ziel, vielmehr die fachlichen Aspekte der Anwendung in den Mittelpunkt des Entwurfs zu stellen. In **dem** Standardwerk über die WAM-Metapher [KGZ94] schreiben Kilberth et al. gleich zu Beginn:

Um die vorhandenen Arbeitszusammenhänge zu verstehen und sinnvolle Anwendungssysteme entwerfen zu können, ist es wesentlich, dabei vorrangig die Gegenstände zu analysieren, die zur Erledigung der alltäglichen Aufgaben im Anwendungbereich verwendet werden. Wir betrachten dabei die sog. ***Umgangsformen*** *mit den Gegenständen, d.h. die Art und Weise wie mit Gegenständen im Rahmen der verschiedenen Aufgaben (auf der „semantischen Ebene") gearbeitet wird. Diese Gegenstände führen uns zu zu den relevanten Begriffen der Anwendung und den dahinterstehenden Konzepten ihrer Verwendung. [...]*

Als Beispiel folgt die „berühmte" fachliche Modellierung eines Ordners und es wird im weiteren Verlauf dargestellt, wie sich dieses Vorgehen letztendlich durch die WAM-Architektur in den Werkzeugen widerspiegelt. Dabei wird vorgeschlagen, ein Werkzeug in eine Funktions- und eine Interaktionskomponente aufzuteilen und die Handhabung eines Werkzeuges in der Benutzungsschnittstelle mit Interaktionstypen zu implementieren, um die Klassen eines GUI-Toolkits kapseln. So kann eine lose Koppelung zwischen Werkzeug und Toolkit erreicht werden.

Die konkreten Interaktionstypen haben aber gezeigt, daß diese lose Koppelung sich leider immer noch als relativ „fest" erweisen kann. Konzeptionell hat es sich zwar bewährt, ein Werkzeug in zwei Komponenten aufzuteilen und Interaktionstypen zu benutzen; nur haben die verfügbaren IAT-Bibliotheken die Eigenart, technische Aspekte des Toolkits oder sogar des Fenstersystemes in das Werkzeug „einzuschleppen".

Mit dem von mir vorgestellten Architekturmuster der Interaktions- und Präsentationsformen wird erreicht, daß sich der fachliche Aspekt eines Werkzeuges nicht mehr nur auf die Funktionskomponente beschränken muß. Mit den Interaktionstypen hatte die IAK einen deutlich technischeren Charakter. Ein Entwickler mußte für jede (nicht unbedingt fachlich motivierte) Änderung an der Oberfläche die Interaktionskomponente und damit einen Teil des Werkzeuges ändern. Dies konnte sich bei der Entwicklung der IAK als umständlich erweisen, weil im schlimmsten Fall für jedes Fenstersystem verschiedene Versionen der IAK erstellt werden mußten. Bei fachlichen Änderungen, die lediglich die FK betreffen, müßten dann sogar mehrere Interaktionskomponenten nachgezogen werden.

Abschluß

Statt der Interaktionstypen sollten in einer IAK also Interaktionsformen benutzt werden. Sie bieten dem Entwickler die Möglichkeit, für den Umgang mit dem Werkzeug eine Schnittstelle zu definieren, mit der eine technische Realisierung der Benutzungsoberfläche von außen angeschlossen werden kann. In der IAK repräsentieren die Interaktionsformen ausschließlich die fachliche Umgangsform, erst mit den Präsentationsformen wird außerhalb des Werkzeuges eine konkrete Oberfläche realisiert. Das gesamte Werkzeug bleibt damit im Kontext der Anwendungswelt.

Dieses Vorgehen unterscheidet die Interaktionsformen wesentlich von anderen Architekturen, vor allem natürlich von denen, die nicht nach WAM entwickelt wurden. Zumeist wird immer eine Abhängigkeit von einem Application Framework erzielt, die in meinen Augen nicht erwünscht sein kann. Die Interaktionsformen und vor allem die Präsentationsformen sind im Architekturmuster der Werkzeuge so eingebunden, daß sie in der Benutzung wie in der konkreten Realisierung leicht ausgetauscht werden können.

Durch die verbesserte Entkoppelung wird noch ein zweiter Effekt erzielt, der hier auch noch einmal betont werden soll. Die Austauschbarkeit der Präsentationsformen ist eine wesentliche Hilfe, um ein Werkzeug auf ein anderes Fenstersystem zu portieren und dabei das Look and Feel der neuen Plattform einzuhalten, ohne an den fachlichen Klassen des Werkzeuges etwas ändern zu müssen. Ein Entwickler muß lediglich die Präsentationsformen so umordnen oder verändern, daß sie dem Styleguide der neuen Plattform entsprechen. Statt für jedes Fenstersystem eventuell Kompromisse eingehen zu müssen, brauchen jetzt nur noch die zugehörigen Präsentationsformen ausgewechselt zu werden. Dies soll im Idealfall ausschließlich durch den Gebrauch von Interface Buildern ermöglicht werden, so daß dieses Vorgehen eine wesentliche Erleichterung darstellen kann.

Meine konkrete technische Umsetzung dieser Architektur hat gezeigt, daß das Muster der Interaktions- und Präsentationsformen praktikabel einsetzbar ist. Mit einem Prototypen habe ich außerdem vorgeführt, daß die Portierung auf eine andere Plattform erleichtert werden kann und damit die verschiedenen Styleguides für den Entwickler nicht nur ein notwendiges Übel sind. Die Frage nach der Portabilität wird damit von mir mit einem methodischen Ansatz beantwortet, der aber immer noch einen (relativ geringen) Aufwand mit sich bringt. Meines Erachtens werden Werkzeuge niemals ohne „Kosten“ vernünftig (!) portiert werden können.

Am Arbeitsbereich Softwaretechnik wird z.Zt. ein neues Rahmenwerk für WAM in der Programmiersprache Java entwickelt. Dabei haben die Entwickler völlig auf den Einsatz von Interaktionstypen verzichtet. Bereits jetzt läßt sich erkennen, daß zumindest „einfache“ Oberflächen tatsächlich mit Interaktions- und Präsentationsformen realisiert werden können. Die Zukunft muß zeigen, ob dies auch für komplexere Benutzungsschnittstellen großer Anwendungssysteme gilt, vor allem wenn dabei auch graphische Elemente eingesetzt werden sollen. Gerade der letzte Punkt bietet noch genügend offene Fragen, um den Gegenstand dieser Arbeit weiter zu untersuchen.

Anhang

Glossar

- ***API*** - „Application Progamming Interface“; prozedurale Programmierschnittstelle einer Software-Komponente (z.B. einer Funktionsbibliothek)
- ***Application Framework*** - Objektorientiertes Rahmenwerk, das eine lauffähige generische Anwendung implementiert, die noch keine anwendungsspezifische Funktionalität enthält; wird häufig auch als Bezeichnung für reine GUI-Frameworks verwendet.
- ***Benutzungsoberfläche*** - Synonym für „Benutzungsschnittstelle“; wird manchmal auch nur als „Oberfläche“ bezeichnet.
- ***Benutzungsschnittstelle*** - Komponente einer Anwendung, die die Ein- und Ausgaben realisiert und damit dem Benutzer eine „Schnittstelle“ für die Bedienung einer Anwendung zur Verfügung stellt.
- ***Callback*** - Eine Funktion, die registriert wird, um zu einen späteren Zeitpunkt aufgerufen zu werden; wird typischerweise in GUI-Toolkits benutzt, um Ereignisse zu verarbeiten, und in prozeduralen Sprachen häufig mit Funktionszeigern realisiert.
- ***Direkte Manipulation*** - Spezielle Metapher bei der Benutzung einer graphischen Benutzungsschnittstelle; Manipulationen, die z.B. mit der Maus erfolgen, haben eine direkte Auswirkung auf den Zustand der Anwendung. Dabei wird eine unmittelbare, visuelle Rückkoppelung gegeben.
- ***Entwurfsmuster*** - Ein ~ beschreibt eine generelle Lösung für wiederholt auftretende, ähnlich geartete Probleme im objektorientierten Entwurf (vgl. [GHJ+94]).
- ***Ereignis*** - Im Kontext der Programmierung von Benutzungsschnittstellen werden „Ereignisse“ als programmiertechnische Ereignisse betrachtet, die durch Eingaben bei der Bedienung einer Anwendung, z.B. durch einen Mausklick oder das Drücken einer Taste ausgelöst werden.
- ***Event*** - (häufig auch „GUI Event“) siehe „Ereignis“
- ***Fenstersystem*** - Software-Komponente, die Funktionen für die Darstellung und Kontrolle verschiedener graphischer Bereichen auf dem Bildschirm bereitstellt; ermöglicht dem Benutzer, gleichzeitig mehrere Anwendungskontexte zu betrachten und zu kontrollieren.
- ***Framework*** - Ausdruck aus der Objektorientierung; bezeichnet eine Menge von Klassen, die die Wiederverwendung einer Problemlösung für eine bestimmte Klasse von Software ermöglicht (vgl. [GHJ+94] und [Wei97]).
- ***Graphical User Interface*** - „GUI“; klassifiziert den Begriff „Benutzungsschnittstelle“ für den Gebrauch von graphischen Elementen, die in einem Fenstersystem zur Verfügung gestellt werden.
- ***GUI-Builder*** - siehe „Interface Builder“
- ***Interface Builder*** - Interaktives Programm, um eine graphische Benutzungsschnittstelle zusammenzustellen, die aus verschiedenen Widgets besteht; entlastet den Entwickler bei der Benutzung eines Toolkits, indem z.B. fertiger Programmtext generiert wird.

Glossar

- ***Look and Feel*** - Ausdruck, der die Einheitlichkeit von Programmen im Aussehen und in der Bedienung beschreibt, die auf einer Plattform ablaufen.
- ***Main Event Loop*** - Routine, um die Ereignisse zu verarbeiten, die der Benutzer bei der Bedienung einer Anwendung auslöst.
- ***Muster*** - siehe „Entwurfsmuster“
- ***Plattform*** - Synonym für eine Kombination aus Betriebsystem und Fenstersystem; Beispiele sind Apple MacOS, Microsoft Windows oder Motif (zumeist mit Unix).
- ***Portierung*** - Programmiervorgang, um eine Anwendung auf eine andere Plattform zu übertragen; dort läuft sie i.d.R. unter einem anderen Fenstersystem ab, so daß der Entwickler eventuell mit einem anderen API programmieren muß.
- ***Rahmenwerk*** - siehe „Framework“
- ***Styleguide*** - Zusammenfassung von Regeln, die in ihrer Gesamtheit das „Look and Feel“ einer Plattform definieren; schreibt z.B. die Gestaltung von Menüleisten, das Layout von Fenstern oder das Verhalten einer Anwendung bei bestimmten Benutzereingaben vor.
- ***Toolkit*** - (häufig auch „GUI Toolkit“) Oberbegriff für Funktions- und Klassenbibliotheken und GUI-Frameworks (vgl. [Mye95]); wird in dieser Arbeit hauptsächlich als Synonym für objektorientierte GUI-Frameworks verwendet.
- ***Virtual Toolkit*** - GUI-Framework, welches das API verschiedener Fenstersysteme kapselt und damit von diesem abstrahiert; kapselt eine Anwendung vom API und ermöglicht die Portierung von Anwendungen, ohne den Programmtext ändern zu müssen.
- ***Widget*** - Oberbegriff für die verschiedenen graphischen Bausteine in einem Fenstersystem, die in einem GUI dargestellt werden; diese können Werte darstellen und vom Benutzer Ereignisse wie Tastendrücke oder Mausklicks entgegennehmen und an das Fenstersystem weiterleiten.

Literaturverzeichnis

[App87] Apple Computer, Inc. *Human Interface Guidelines: The Apple Desktop Interface*. Reading, MA: Addison-Wesley, 1987.

[BKS+86] Reinhard Budde, Karin Kuhlenkamp, Karl-Heinz Sylla, Heinz Züllighoven. *Constructing Interactive Applications in a Prolog Programming Environment*. Technical Paper, GMD, 1986.

[Frö96] Frank Fröse. *Konzepte der Fenstersystemanbindung mit Interaktionstypen*. Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1996.

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

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley, 1984.

[Goo90] Danny Goodman. *The Complete HyperCard 2.0 Handbook, 3rd Edition*. New York: Bantam Books, 1990.

[KGZ94] Klaus Kilberth, Guido Gryczan, Heinz Züllighoven. *Objektorientierte Anwendungsentwicklung*. (2., verbesserte Auflage), Wiesbaden: Vieweg, 1994.

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

[Lil97] Carola Lilienthal. *Dokumentation der Bib V3.0*. (http://swt-www.informatik.uni-hamburg.de), Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1997.

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

[Met96] Metrowerks, Inc. *PowerPlant for CodeWarrior - Users Guide*, Austin, TX, 1996.

[Mic93] Microsoft, Inc. *Microsoft Foundation Classes - Documentation*, Redmond, WA, 1993.

[Mye90] Brad A. Myers et al. *Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces*. in Computer, Vol. 23, No. 11, November 1990.

[Mye95] Brad A. Myers. *User Interface Software Tools*. in ACM Transactions on Computer-Human Interaction, Vol. 2, No. 1, March 1995.

[Mye96] Brad A. Myers. *User Interface Software Tools*. (Auflistung von GUI-Toolkits) http://www.cs.cmu.edu/afs/cs.cmu.edu/user/bam/www/toolnames.html. 1996

[Mye97] Brad A. Myers. *The Amulet Environment: New Models for Effective User Interface Software Development*. in IEEE Transactions on Software Engineering, Vol. 23, No. 6, June 1997.

[NeX91] NeXT, Inc. *NeXT Step and the NeXT Interface Builder - Users Guide*. Redwood City, CA, 1991.

Literaturverzeichnis

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

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

[Shn87] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Reading, MA: Addison-Wesley, 1987.

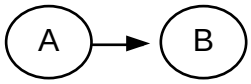
[Sma97] Julian Smart. *WxWindows - Documentation.* http://web.ukonline.co.uk/julian.smart/wxwin. 1997

[VL90] John M. Vlissides, Mark A. Linton. *Unidraw: A framework for building domain-specific graphical editors.* in ACM Transactions on Information Systems, Vol. 8, No. 3, July 1990.

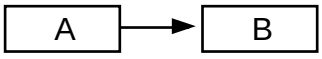
[Wei97] Ulfert Weiß. *Konzeption und technische Weiterentwicklung eines objektorientierten Frameworks nach der Werkzeug-Material-Metapher.* Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1997.

[WG95] André Weinand, Erich Gamma. *ET++ - a Portable, Homogeneous Class Library and Application Framework.* in [Lewis95]

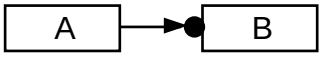
[Wil90] David A. Wilson. *Programming with MacApp.* Reading, MA: Addison-Wesley, 1990.



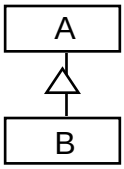
Objekt A benutzt Objekt B



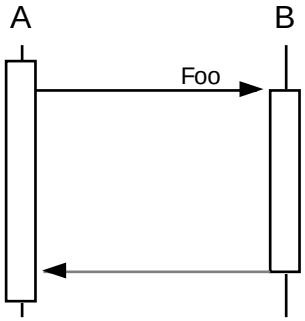
Klasse A benutzt Klasse B



Klasse A benutzt mehrere Objekte der Klasse B



Klasse A ist Basisklasse der Klasse B



Objekt A ruft Methode „Foo“ des Objektes B