Die Integration von Dialogablauf-Beschreibungen in eine objektorientierte User-Interface-Architektur

Josef Voss
FernUniversität Hagen
Praktische Informatik III
Postfach 940
D- 5800 Hagen

1 Einleitung

Die Realisierung anspruchsvoller graphischer Benutzeroberflächen hat sich als komplexer und damit aufwendiger Teil der Softwareerstellung erwiesen. Der Anteil an der Gesamtentwicklung eines Softwareprojekts wird mit bis zu 50% angegeben [11]. Insbesondere bei Benutzeroberflächen, die dem Prinzip der "direkten Manipulation" folgen, ergeben sich hohe technische Anforderungen:

- Nebenläufige Benutzeraktivitäten: Komplexe Anwendungen bestehen aus einer Reihe von Teilen, die ein Benutzer nicht streng sequentiell bearbeiten muß, sondern zwischen denen er jederzeit wechseln kann.
- Schnelles, anwendungsspezifisches Feedback (semantisches Feedback): Insbesondere, wenn auf die Bewegung der Maus eine anwendungsspezifische Reaktion erfolgen soll, etwa zur Visualisierung gültiger Positionen beim Verschieben eines Objekts.
- Komplexe Abhängigkeiten zwischen verschiedenen Dialogteilen: Beispielsweise können Selektionen und Veränderungen in einem Fenster auch Konsequenzen für das Erscheinungsbild anderer Fenster haben.

Aus der Sicht eines Entwicklers stellt sich eine Benutzeroberfläche unter den Gesichtspunkten Bildschirmlayout, Dialogablauf und Softwarestruktur.
dar. Werkzeuge und die darin verwendeten Modelle müssen den Entwickler in diesen drei Bereichen unterstützen. Der folgende Abschnitt gibt einen kurzen Überblick, wie weit bei existierenden Werkzeugen die Unterstützung geht. Ein eigener Abschnitt ist dabei objektorientierten Architekturen gewidmet. Daran anschließend wird das Modell DIWA mit seiner Softwarestruktur vorgestellt. Der wesentliche Aspekt des Modells ist die Integration einer objektorientierten Architektur mit high-level Ablaufbeschreibungen. Dabei wird insbesondere deutlich, welche Rolle Vererbung bei der Beschreibung und Implementierung von Objektstruktur und Dialogverhalten spielen kann.

2 Werkzeuge zur User-Interface-Entwicklung - Überblick

2.1 Klassische User-Interface-Management-Systeme

Unter der Bezeichnung User-Interface-Management-System (UIMS) sind Werkzeuge entwickelt worden, die im wesentlichen aus einem Compiler oder Interpreter für textuelle Beschreibungen von Dialogabläufen bestehen. Einige derartige Systeme sowie das Seeheim-Modell, eine Art UIMS-Referenzarchitektur, sind in [13] beschrieben. Als wichtigste, aus heutiger Sicht allerdings nicht unumstrittene, Grundideen des UIMS-Ansatz kann man festhalten:

- User-Interface (UI) und Anwendungskomponente können voneinander getrennt werden; und die beiden Komponenten sind weitgehend unabhängig. Das heißt, daß man anwendungsunabhängige, wiederverwendbare UI-Komponenten auf verschiedene Anwendungen aufstecken kann.
- Eingaben und Ausgaben werden aus einer sprachlichen Sicht als Folgen von Eingabe- bzw. Ausgabezeichen, gesehen.
- Ein Dialogablauf ist im wesentlichen durch die erlaubten Folgen von Benutzereingaben festgelegt. Diese lassen sich programmiersprachenunabhängig spezifizieren, als Grammatik [12], Zustands-Transitions-Netz [9], Event-Response-System [8], Eventhandler [1], [4]. Die flexibelste Technik zur Ablaufbeschreibung sind Eventhandler. Das UI wird dabei als ein System kooperierender Prozesse aufgefaßt, die jeder für sich Benutzereingaben verarbeiten.

2.2 Probleme im Einsatz von UIMSen

UIMSe haben bisher noch nicht den Status einsetzbarer Werkzeuge erreicht. Im Grunde ist dafür deren mangelnde Flexibilität verantwortlich. Im einzelnen kann man folgende Gründe anführen:

- Die Idee der Unabhängigkeit von UI und Anwendung läßt sich nicht aufrecht erhalten, da ein UI im Detail eine Vielzahl von anwendungsspezifischen Reaktionen zeigen muß. Die logische Trennung in verschiedene Softwarekomponenten (Module, Objekte) bleibt aber sinnvoll.
- In UIMSen eingesetzte abstrakte Beschreibungen beschränken sich auf Dialogabläufe. Das Bildschirmlayout und graphische Ausgaben werden nicht erfaßt. Diese erfordern nach wie vor direkte Programmierung oder sind vom Entwickler nur wenig beeinflußbar.
- Semantisches Feedback erfordert, daß UI und Anwendung intensiv kommunizieren oder daß anwendungsspezifisches Wissen ins UI verlagert wird. Erfolgversprechend ist nur der zweite Ansatz. Allerdings muß dazu die UI-Software offen und erweiterbar sein.
- Anspruchsvolle Oberflächen lassen sich nicht allein durch die Kombination vorgefertigter Dialog-Bausteine (man spricht auch von interaction techniques) erstellen, sondern erfordern die Erzeugung von neuen und die Anpassung existierender Bausteine. Diese Möglichkeiten sind in der Regel nicht gegeben.
- UIMSe sind aufwendig zu erstellen und im Bezug auf eingesetzte Windowsysteme und die unterstützten Dialogarten schwer auf dem neusten Stand zu halten [14]. Ziel muß daher die Entwicklung flexibler, erweiterbarer Systeme sein.

2.3 Toolkits

Als effektiv einsetzbare Alternative zu UIMSen sind sogenannte Toolkits entstanden. Das sind Bibliotheken mit Prozeduren, bzw. Klassen und Methoden, die dem Entwickler einen komfortableren Zugang zu einem Windowsystem zur Verfügung stellen. Typischerweise wird damit die Erstellung und Verwendung von Menüs, Fenstern, Scrollbars usw. erleichtert. Ein Beispiel für ein Toolkit ist die Macintosh Toobox. Der Einsatz von Toolkits stellt den Entwickler allerdings vor die Aufgabe, eine Vielzahl (oft mehrere hundert) Prozeduren bzw. Methoden zu überschauen und für sein konkretes User-Interface in der richtigen Weise zu kombinieren. Dazu muß er sich weiterhin auf der Programmiersprachenebene bewegen.

Auf objektorientierten Programmiersprachen basierende Toolkits unterscheiden sich von den auf Prozeduren beruhenden dadurch, daß sie zum einen eine größere Flexibilität bieten. Man benutzt nicht mehr nur die vorgegebene interaction techniques, sondern hat die Möglichkeit, durch Unterklassenbildung Änderungen an den vordefinierten Bausteinen vorzunehmen. Zum anderen

liegt einem objektorientierten Toolkit in der Regel ein Architekturkonzept zugrunde, das ein standardisiertes Muster vorgibt, aus welchen Objekttypen und nach welchen Strukturierungsprinzipien ein User-Interface aufgebaut wird. Die hierbei zu erkennende Idee, ein User-Interface als System kooperierender Objekte aufzufassen, ist auch für neuere Werkzeugentwicklungen von großer Bedeutung. Deshalb gehen wir im nächsten Abschnitt etwas ausführlicher darauf ein.

Das Ziel offener und erweiterbarer Systeme ist mit objektorientierten Toolkits erreicht. Allerdings sollte als wesentlicher Nachteil gegenüber UIMSen noch einmal festgehalten werden, daß die Möglichkeit, Dialogabläufe abstrakt zu beschreiben, bei der Verwendung von Toolkits nicht mehr gegeben ist. Dialogabläufe sind nur noch implizit im Programm enthalten.

3 Objektorientierte User-Interface Architekturen

Die Idee für objektorientierte User-Interface Architekturen resultiert aus dem Bestreben, die UI-Komponente nicht aus großen, in ihrer Komplexität nicht durchschaubaren Modulen (nach dem Seeheim Modell sind dies Presentation, Dialog-Control und Application-Interface-Model) aufzubauen, sondern aus kleinen Einheiten, die jede für sich einen in sich abgeschlossenen Teil des UI behandeln.

Man kann verschiedene Ansätze für eine globale Strukturierung des UI unterscheiden. Auf der einen Seite werden verschiedene spezialisierte Objekttypen identifiziert, die jeweils nur einen Teilaspekt - Eventverarbeitung, Bildschirmdarstellung, lokale Datenhaltung usw. - behandeln und die untereinander relativ frei kombiniert werden. Ein Beispiel ist MacApp [15], eine Erweiterung der Macintosh Toolbox, das die Objekttypen Window, Document, Command, View, Frame und Application unterscheidet.

Auf der anderen Seite stehen Modelle, die gerade die gemeinsamen Aufgaben verschiedener UI-Bestandteile betonen, indem sie ein User-Interface aus gleichartigen Objekten aufbauen. Solche Objekte umfassen in ihrem Inneren wiederum die oben aufgeführten Teilaspekte Eventverarbeitung, Bildschirmdarstellung und lokale Datenhaltung.

Man kann als Ausgangspunkt für die zweite Richtung das MVC-Modell von Smalltalk sehen [10]. Die Abkürzung "MVC" steht für Model, View und Controller, den drei wichtigsten Komponenten des Modells, die (evtl. in spezialisierter Form) in jedem UI-Bestandteil (das sind Fenster und Teilfenster) vorkommen. Derartig strukturierte Bestandteile sind z.B Fenster zum Präsentieren und Editieren von Listen, Texten oder Graphiken. Die Aufgabenverteilung zwischen Model, View und Controller ist dabei immer gleich. Es gibt allerdings keine Instanz, in der der Zusammenhalt und gemeinsame Aufgaben von Model-View-Controller Tripeln festgehalten wird[1].

In systematischerer Weise wird eine Architektur, die homogen strukturierte Objekte herausstellt, mit dem PAC-Modell von Coutaz [3] beschrieben. Die Objekte werden "interactive objects" genannt. Jedes besteht aus einer Presentation-, einer Abstraction- und einer Control-Komponente. Interactive Objects werden hierarchisch angeordnet, eine Idee, die sich im Kern schon bei MVC beobachten läßt. Neu ist die Abstraction-Komponente, die in jedem Interactive Object die Möglichkeit bietet, Anwendungsdaten zu speichern und zu manipulieren, womit insbesondere semantisches Feedback unterstützt wird.

1.. Genaugenommen handelt es sich um View-Controller Paare. Die Zuordnung von Model-Objekten ist flexibler. Ein Model kann für mehrere View-Controller Paare zuständig sein

User-Interface-Objekte

Als Quintessenz aus den genannten objektorientierten Architekturen kann man die Herausbildung eines speziellen Objektbegriffs für eine gemeinsame Abstraktion von Komponenten innerhalb des UI sehen. Diese Abstraktion soll einheitlich strukturierte UI-Bausteine beschreiben, aus denen man ein UI vollständig zusammensetzen kann. Wir werden im folgenden für diese Bausteine die Bezeichnung User-Interface-Objekt (UIO) verwenden; dieser Begriff wird auch in [7] benutzt.

Ein idealer Begriff von UIO beschreibt nicht nur selbständig und lokal agierende Softwarekomponenten mit einheitlicher Struktur und Schnittstelle zu anderen Objekten, sondern eignet sich gleichermaßen zur Strukturierung von Ablauf, Layout und Software-Architektur und führt dabei zu kongruenten Strukturierungen. Ein UIO bündelt dann ein zusammengehöriges Stück von Layout und Ablauf in einer Softwarekomponente.

4 Das Modell DIWA

DIWA knüpft in seiner Architektur bei MVC und PAC an. Es werden ebenfalls hierarchisch angeordnete UIO's verwendet. Die Architektur wird ergänzt durch eine abstrakte, an Zustands-Transitions-Netzen orientierte Beschreibungssprache für Dialogabläufe. Ablaufbeschreibungen ordnen den Objekten Eventhandler zu. Dementsprechend verstehen wir die UIO's als parallel arbeitende Agenten. Das Aussehen und Verhalten von Objekten wird durch die Definition von UIO-Klassen festgelegt.

Die Einbeziehung abstrakter Beschreibungen in das Modell hat das Ziel, die erhaltenswerte Idee aus den UIMS-Ansätzen, Dialogabläufe explizit und auf einem Niveau oberhalb von Programmiersprachen zu beschreiben, in die objektorientierte Umgebung hinüberzuretten, ohne daß deren Flexibilität verlorengeht.

Wichtigstes Kriterium zum Identifizieren von Objekten (UIO's) und ihren Klassen sowie von Objekt-Subobjekt Beziehungen ist die Zuständigkeit eines Objekts. Ein Entwurf orientiert sich an der Frage, welche Aufgaben ein Objekt hat und welches Wissen es zur Erledigung dieser Aufgaben benötigt, was insbesondere bedeutet, welche anderen (untergeordneten) Objekte es dabei zur Unterstützung braucht. Man kann von einem verantwortlichkeitsbestimmten Ansatz zum objektorientierten Entwurf sprechen [19].

Beim UI-Entwurf sind die Objekte in der Regel auf dem Bildschirm unmittelbar zu identifizieren, wobei mit der Zuständigkeit häufig ein geometrisches Enthaltensein von Subobjekten in Objekten einhergeht. Ein guter UI-Entwurf wird immer zusammengehörende Dinge, etwa die zu einer Anwendung gehörenden Kontrollfelder oder Scrollbars, auch räumlich zusammen anordnen, z.B. in einem Fenster oder in einem Formular. Andererseits bekommt man mit einem zuständigkeitsorientierten Entwurf auch Objekte mit eher koordinierenden Aufgaben, die dann nicht unbedingt eine Bildschirmrepräsentation haben, die sich unmittelbar aus ihrer Aufgabe ergibt.

Die Bedeutung der Objekt-Subobjekt-Beziehungen kann somit sowohl geometrisch als "liegt innerhalb" als auch logisch als "benutzt", "kontrolliert" oder "ist Teil von" interpretiert werden.

Ein weiteres Kriterium für den Entwurf von Klassen für UIO's ist deren Wiederverwendbarkeit. Man definiert Klassen, die immer wieder vorkommende Strukturen und Dialogabläufe beschreiben. Insbesondere setzen wir hierzu Mehrfachvererbung ein, damit verschiedene unabhängig voneinander formulierbare Strukturen und Abläufe isoliert und später in Unterklassen miteinander kombiniert werden können. Vererbung wird aber nicht nur dazu eingesetzt, daß durch Wie-

derverwenden existierender Beschreibungen die Erstellung neuer UI-Spezifikationen vereinfacht wird, sondern als bewußt eingesetztes Strukturmerkmal: Universelles Dialogverhalten soll so weit wie möglich extrahiert und damit sichtbar gemacht werden. Zur Unterstützung dieses Ansatzes führen wir Regeln ein, die die Bildung von Unterklassen reglementieren, so daß einmal ererbte Strukturen und Abläufe nur noch in engen Grenzen verändert werden können.

4.1 Die Basisarchitektur von DIWA

Wir wollen die Architektur hier nur soweit beschreiben, wie sie für die folgenden Abschnitte von Bedeutung ist. Weitere Details finden sich in [16],[18]. Ein einzelnes UIO besteht wieder aus drei Komponenten. Die folgende Abbildung zeigt den Aufbau eines UIO:

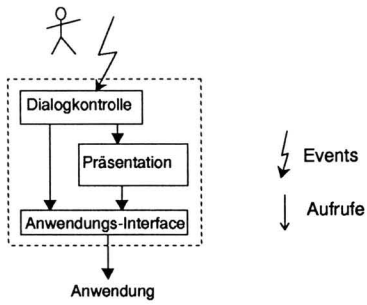

Abbildung 1

- Die Dialogkontrolle ist für die weiteren Betrachtungen die wichtigste der drei Komponenten. Sie empfängt und verarbeitet alle Events von außen, das sind Eingaben vom Benutzer und Signale von anderen Objekten, und entscheidet, wie darauf zu reagieren ist. Die Reaktionen bestehen aus Aufrufen an Präsentation und Anwendungs-Interface, sowie aus Signalen an andere Objekte. Durch die Einbindung in die Objekthierarchie ergibt sich für eine Dialogkontrolle die Aufgabe, untergeordnete Objekte zu kontrollieren, das heißt zu aktivieren oder zu deaktivieren, sowie zwischen verschiedenen Subobjekten zu vermitteln, da diese nicht direkt miteinander kommunizieren.
- Die Präsentation ist für die Bildschirmdarstellung des UIO verantwortlich. Dazu gehört auch die Berechnung, Veränderung und evtl. die lokale Speicherung der Objektdarstellung.
- Das Anwendungs-Interface sorgt für Datentransporte von und zur Außenwelt. Daten werden dabei direkt zwischen Präsentation und Anwendungs-Interface ausgetauscht. Die Dialogkontrolle ist nur als auslösende Instanz beteiligt. Dadurch ist die Dialogkontrolle nicht von programmiersprachenspezifischen Datenformaten abhängig.

4.2 UIO-Klassen

Die Architektur von DIWA könnte für sich stehen und als Modell für die Implementierung eines Toolkits dienen. Wir wollen aber einen Schritt weiter in Richtung Werkzeug gehen und UIO's so weit wie möglich abstrakt, das heißt programmiersprachenunabhängig beschreiben. Dabei soll die Flexibilität eines objektorientierten Toolkits erhalten bleiben.

Zu diesem Zweck wird ein programmiersprachenunabhängiges Schema zur Definition von UIO-Klassen eingeführt. Das Schema umfaßt neben der globalen Objektstruktur auch Dialogabläufe, also die Dialogkontrollkomponenten. Die Definition einer UIO-Klasse umfaßt Angaben über:

- Eine Reihe von Subobjekten, über die ein UIO dieser Klasse verfügt. Für jedes Subobjekt wird ein Name und eine Klasse spezifiziert.

- Eine Reihe von Attributen.
- Eine Reihe von Eventhandlern, die das Dialogverhalten von UIO's dieser Klasse beschreiben.

Jede Klasse wird als Unterklasse einer bereits definierten UIO-Klasse beschrieben. Die allgemeinste Klasse wird mit *DialogObjects* bezeichnet. Bei der Definition von Unterklassen werden lediglich die neu hinzukommenden Subobjekte, Attribute und Eventhandler angegeben. Geerbte Subobjektangaben können in Unterklassen modifiziert, das heißt spezialisiert werden, indem als neue Klasse des Subobjekts eine Unterklasse der alten angegeben wird. Einmal definierte Attribute und Eventhandler können in Unterklassen nicht mehr verändert werden.

Neben der Flexibilisierung der Typstruktur durch die Möglichkeit, neue, auch anwendungsspezifische UIO-Klassen zu definieren, ist eine Dynamisierung der Objektstruktur ein wesentliches Anliegen des DIWA-Ansatzes. Dazu müssen Anzahl und Anordnung von Subobjekten zur Laufzeit des Dialogs veränderbar sind. Zu diesem Zweck bietet DIWA die Möglichkeit, variabel große Mengen gleichartiger Subobjekte zu bilden. Die Elemente solcher Mengen müssen nicht individuell bezeichnet werden. Man kann daher auch von anonymen Objekten sprechen. Die Menge selbst bekommt einen Namen und wird weitgehend wie andere, einzelne Subobjekte behandelt.

Als Beispiel definieren wir eine Klasse für einfache Popup-Menüs. Objekte dieser Klasse besitzen als Subobjekte eine Menge *ItemSet* von Einträgen der Klasse *Items*, die schon definiert sei, sowie einen Titel, der ebenfalls von der Klasse *Items* ist. Oberklasse von *PopUpMenus* ist die Klasse *DialogObjects*.

```
DialogObjects class PopUpMenus
   subobjects:   [ Title:      Items,
                   ItemSet:    setOf Items ]
end PopUpMenus
```

In einer anwendungsspezifischen Unterklasse *EditMenus* werden schließlich individuelle Einträge als Elemente von *ItemSet* definiert.

```
PopUpMenus class EditMenus
   subobjects:   [ Insert:   InsertItems   elementOf ItemSet,
                   Delete:   DeleteItems   elementOf ItemSet,
                   Modify:   ModifyItems   elementOf ItemSet ]
end EditMenus
```

4.3 Die Integration von Ablaufbeschreibungen

Ein System von Eventhandlern realisiert die Dialogkontrollkomponente eines DIWA-UIO's. Die für Eventhandler entwickelte Beschreibungssprache erlaubt, Ereignisse und Aktionen, insbesondere solche mit Bezug zu anderen Objekten, zu beschreiben. Dazu werden auf die hierarchische Objektstruktur und auf die dynamisch veränderbaren Objektmengen zugeschnittene Objektausdrücke verwendet. Zum Beispiel bezeichnet der Ausdruck

ItemSet(cursor)

den gerade unter dem Mauscursor liegenden Menüeintrag.

Weiterhin werden Sprachmittel zur Kommunikation und Synchronisation mit anderen Eventhandlern und Objekten zur Verfügung gestellt. Bei der Auswahl der Sprachmittel, insbesondere für Objektausdrücke und Kommunikation, ist im Sinne einer besseren Transparenz von UI-Entwürfen als wichtiges Designkriterium die Beschränkung der Zugriffs- und Einflußmöglichkeiten eines Eventhandlers auf ihm untergeordnete Handler und Objekte verfolgt worden.

Wir wollen das Zusammenspiel verschiedener Eventhandler eines Objekts am Beispiel der Pop-up-Menüs demonstrieren. Der Besitzer des Menüs entscheidet, wann das Menü aktiviert und wo es dann dargestellt wird. Das Menüobjekt selbst sorgt für seine Darstellung, das Feedback (Invertieren von Einträgen, wenn die Maus hineinbewegt wird) und meldet schließlich ein Signal an den Besitzer, wenn der (rechte) Mausknopf über einem der Einträge losgelassen wird. Die gerade beschriebenen Aufgaben werden von den zwei Eventhandlern *FeedbackHandler* (der auch für die Darstellung bei der Aktivierung des Menüs verantwortlich ist) und *SendSignalHandler*, der den Besitzer über den angewählten Eintrag unterrichtet. Der *SendSignalHandler* kann allerdings erst in einer Unterklasse (z.B. *EditMenu*) definiert werden, wenn die konkreten Einträge und die damit verbundenen Signale festgelegt sind. Die Beschreibung der Klasse PopUpMenus einschl. der Eventhandler hat jetzt die Form:

```
DialogObjects class PopUpMenus
   subobjects:          [ Title:          Items,
                          ItemSet:        setOf Items ]
   attributes:-
   eventhandlers:       [ FeedbackHandler ]
   FeedbackHandler: eventHandler
        type: simpleHandler
        stateDesc:      ( Enter (ItemSet) → setAttr (Invers) to eventob → keepState
                        | Exit (ItemSet) → unsetAttr (Invers) to eventob → keepState )
        startActions:   setAttr (mapped);
                        setAttr (Invers) to ItemSet (cursor)
        stopActions:    unsetAttr (Invers) to ItemSet (Invers);
                        unsetAttr (mapped)
   end FeedbackHandler
end PopUpMenus
```

Als Erweiterung der Popup-Menüs, die die Bedeutung von Vererbung für die Beschreibung von Dialogverhalten illustriert, führen wir nun Submenüs ein. Dazu wird zunächst die Klasse *Items* zu *ItemsWithSubmenu* modifiziert. Ein solcher Eintrag besitzt als Subobjekt ein Feld *FollowBox* (mit einem Pfeil nach rechts darin). Das Submenü wird gezeigt, wenn die Maus in dieses Feld hineinbewegt wird. Die folgende Abbildung zeigt das Layout der Menü- und Itemobjekte:

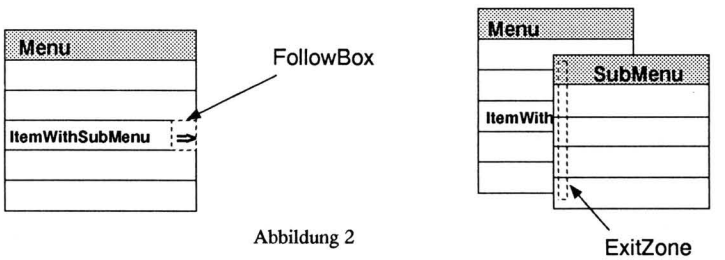

Abbildung 2

Das Submenü selbst wird als weiteres Subobjekt des modifizierten Items aufgefaßt. Das Attribut *Up* dient zum Markieren des Eintrags mit dem gerade aktivierten Submenü. Das Submenü bekommt ebenfalls eine eigene Klasse. Als Erweiterung gegenüber den normalen Popup-Menüs enthält es ein Subobjekt *ExitZone*, ein schmales unsichtbares Rechteck am linken Rand des Submenüs. Wenn die Maus über dieses Feld, das heißt nach links aus dem Submenü heraus, bewegt wird, verschwindet das Submenü wieder. Es verschwindet natürlich auch, wenn der rechte Mausknopf losgelassen wird. Über Aktivierung und Deaktivierung des Submenüs wacht ein neuer

Eventhandler *SubMenuWatchHandler* des übergeordneten Menüs:

```
Items class ItemsWithSubMenu
   subobjects:    [ FollowBox: TextButtons
                    SubMenu: SubMenus ]
   attributes:    [ Up ]
   eventhandlers:-
end ItemsWithSubMenus

PopUpMenus class SubMenus
   subobjects:    [ ExitZone: InvisibleRects ]
   attributes:    -
   eventhandlers:-
end SubMenus

PopUpMenus class PopUpsWithSubMenu
   subobjects:-
   attributes:-
   eventhandlers: [ SubMenuWatchHandler ]
   SubMenuWatchHandler: eventhandler
        stateDesc:Watching = (Enter (ItemSet . FollowBox ) →
                                    deactivateOtherHandlers to parenthdl;
                                    setAttr (Up) to eventob . parentob;
                                    start to eventob . parentob . SubMenu → ChildUp )
                    ChildUp =   (RightUp,
                                 Enter (ItemSet . SubMenu . ExitZone) →
                                    stop to ItemSet (Up) . SubMenu;
                                    unsetAttr (Up) to ItemSet (Up);
                                    reactivateHandlers to parenthdl → Watching )
   end SubMenuWatchHandler
end PopUpsWithSubMenu
```

Der *SubMenuWatchHandler* muß dafür sorgen, daß *FeedbackHandler* und *SendSignalHandler* des Obermenüs deaktiviert werden, solange das Submenü aktiv ist; denn die Maus kann das Submenü verlassen, ohne daß es verschwindet, und dann ins Obermenü (bzw. den davon sichtbaren Teil) bewegt werden. Dieses soll dann nicht reagieren, natürlich auch nicht, wenn der Mausknopf über einem der Items losgelassen wird. Die folgende Abbildung zeigt die Objekt-Subobjekt-Beziehungen in einem Menü mit Submenü. Objekte mit einem oder mehreren Eventhandler sind durch Umrandungen hervorgehoben:

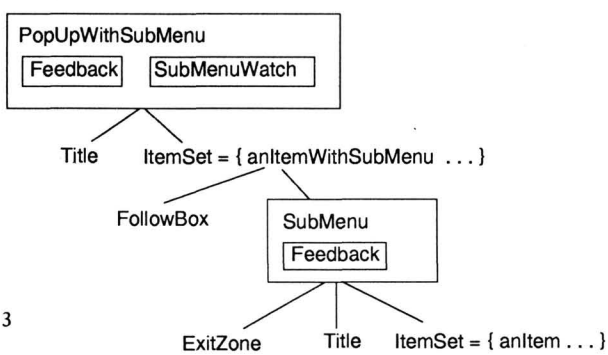

Abbildung 3

Man kann das Einsetzen von Submenüs in Submenüs iterieren, wenn man die Eigenschaften der Klassen *SubMenus* und *PopUpsWithSubMenu* kombiniert. Die neue Klasse *SubMenusWithSubMenu* benötigt außer den ererbten Eigenschaften keine weiteren Angaben und kann für alle Menüs genutzt werden, die sowohl ein Submenü als auch ein Obermenü haben.

```
[ SubMenus, PopUpsWithSubMenu ] class SubMenusWithSubMenu
   subobjects:   -
   attributes:   -
   eventhandlers: -
end SubMenusWithSubMenu
```

Schließlich hat die Klassenhierarchie für die vorgestellten Menü- und Itemklassen folgendes Aussehen:

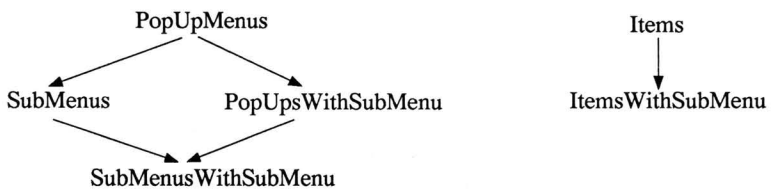

Abbildung 4

5 Zusammenfassung und Ausblick

Wir haben in dieser Arbeit zunächst in einem kurzen Abriß die Anforderungen an die Entwicklung anspruchsvoller graphischer Benutzeroberflächen sowie existierende Werkzeuge zur Unterstützung der User-Interface-Entwicklung beschrieben. Besonderes Augenmerk wurde dabei auf objektorientierte User-Interface-Architekturen gelegt. Als Quintessenz aus verschiedenen derartigen Architekturen wurde das Konzept der User-Interface-Objekte (UIO) vorgestellt, das einen zusammengehörigen Teil von Bildschirmlayout und Dialogablauf in einer weitgehend selbständig agierenden Softwarekomponente bündelt.

Wir haben dann das Modell DIWA vorgestellt, das neben seiner auf UIO's beruhenden Softwarearchitektur programmiersprachenunabhängigen Dialogablaufbeschreibungen integriert. Dialogabläufe werden als System kooperierender Eventhandler beschrieben. An Beispielen wurde gezeigt, daß es damit möglich ist, komplexe Abläufe zu zerlegen, und insbesondere durch den Einsatz von Vererbung universelle von anwendungsspezifische Abläufen zu trennen.

Bisher haben wir basierend auf dem Windowsystem NeWS [5] ein Toolkit mit einem darin enthaltenen Interpreter für Ablaufbeschreibungen implementiert. Ein wichtiger Ansatzpunkte für weitere Entwicklungen ist die Erweiterung des Beschreibungsmechanismus auf das Bildschirmlayout (Dieses steckt bisher noch in der Implementierung der Präsentationskomponenten der UIO's). Erste Ansätze für high-level Beschreibungen von Layout in einem Toolkit kann man z.B. in [17] finden.

6 Literatur

[1] Alexander, H. *Executable Specifications as an Aid to Dialogue Design.* Proceedings Interact '87, Elsevier, 1987, 739-744.

[2] Cardelli, L. und Pike, R. *Squeak: A Language for Communicating with Mice.* Proceedings SIGGRAPH '85, ACM Computer Graphics 19,3 (1985) 199-204.

[3] Coutaz, J. *PAC, an Object Oriented Model for Dialog Design.* Proceedings Interact '87, Elsevier, 1987, 431-436.

[4] Flecchia, M. und Bergeron, R. *Specifying Complex Dialogs in ALGAE.* Proceedings CHI+GI '87, ACM New York, 1987, 229-234.

[5] Gosling, J., Rosenthal, D. und Arden, M. *The NeWS Book.* Springer Verlag, 1989.

[6] Green, M. A *Survey of Three Dialogue Models.* ACM Transactions on Graphics 5,3 (1986), 244-275.

[7] Herrmann, M. und Hill, R. *Abstraction and Declarativeness in User Interface Development. The Methodological Basis of the Composite Object Architecture.* Proceedings IFIP World Computer Conference, Elsevier, 1989, 253-258.

[8] Hill, R. *Supporting Concurrrency, Communcation and Synchronization in Human-Computer Interaction - The Sassafras UIMS.* ACM Transactions on Graphics 5,3 (1986), 179-210.

[9] Jacob, R. *A Specification Language for Direct-Manipulation User Interfaces.* ACM Transactions on Graphics 5,4 (1986), 283-317.

[10] Krasner, G. und Pope, S. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, Aug./Sept. 1988.

[11] Myers, B. *User-Interface Tools: Introduction and Survey.* IEEE Software, Jan. 1989 15-23.

[12] Olson, D. und Dempsey, E. Syngraph: *A Graphical User Interface Generator.* ACM Computer Graphics 17,3 (1983), 43-50.

[13] Pfaff, G. (Hrsg.) *User Interface Management Systems.* Springer Verlag, 1985.

[14] Rosenberg, J., Hill, R. Miller, J., Schulert, A. und Shewmake, D. *UIMSs: Threat or Menace?* Proceedings CHI '88, ACM New York, 1988, 197-200.

[15] Schmucker, K. *MacApp: An Application Framework.* Byte 11,8 (1986), 189-193.

[16] Six, H.-W. und Voss, J. *DIWA- A Hierarchical Object-Oriented Model for Dialog Design,* Proc. IFIP Working Conference on Engineering for Human-Computer Interaction, Napa Valley, USA, 1989.

[17] Szekely, P. und Myers, B. *A User Interface Toolkit Based on Graphical Objects and Constraints.* Proceedings OOPSLA '88, ACM SIGPlan Notices 23,11 (1988), 36-45.

[18] Voss, J. *Entwurf und Implementierung von graphischen Benutzeroberflächen - Ein integrierter, objektorientierter Ansatz,* Dissertation FernUniversität Hagen 1990.

[19] Wirfs-Brock, R. und Wilkerson, B. *Object-Oriented Design: A Responsibility-Driven Approach.* Proceedings OOPSLA '89, SIGPlan Notices 24,11 (1989), 71-75.