Entwicklung graphischer Benutzungsschnittstellen durch hierarchische Komposition

Michael Castner, Berlin

Der Gestaltungsspielraum für den Softwareentwickler bei der Entwicklung von graphischen Benutzungsschnittstellen hat sich im Vergleich zu den zeichenorientierten Benutzungsschnittstellen stark vergrößert und bietet ihm die Möglichkeit, neue und auf ein Anwendungsgebiet zugeschnittene Konzepte für den Benutzer angemessen zu verwirklichen. Allerdings wird die Kreativität der Entwickler einerseits durch die aufwendige Programmierung und andererseits durch die meist mangelnde Erweiter- und Änderbarkeit bestehender Repertoires von Standardbausteinen unnötig eingeschränkt. Im folgenden werden die Probleme bestehender Ansätze verdeutlicht und die Konzeption eines objektorientierten Fenstersystems (OFS) erläutert, das basierend auf einer hierarchischen Komposition beliebiger Bausteine eine flexible und kreative Entwicklung graphischer Benutzungsschnittstellen unterstützt.

1. Problempunkte bestehender Konzepte

Die Problempunkte bestehender Ansätze müssen vor dem Hintergrund verschiedener Anforderungen gesehen werden, die sowohl den Entwicklungsprozeß als auch die Qualität der Benutzungsschnittstelle betreffen. Eine wesentliche Erkenntnis der letzten Jahre ist, daß die Softwareentwicklung nicht als linearer Prozeß betrachtet werden darf, sondern vielmehr durch eine evolutionäre Entwicklungsstrategie geprägt ist, in der die Benutzer/Entwickler-Kommunikation in Verbindung mit prototyporientierten Vorgehensweisen im Vordergrund steht[1]. Keil-Slawik[2] führt dazu aus, daß die Qualität von Software sich erst im Einsatz erweist und ihre Angemessenheit nur in Beziehung zur Einbettung in den jeweiligen Arbeitskontext bewertet werden kann.

Die Forderungen nach einer prototypischen Entwicklung mit verstärkter Benutzer/Entwickler-Kommunikation besitzen gleichzeitig Auswirkungen auf den Entwicklungsprozeß von Benutzungsschnittstellen. Der Entwickler muß in der Lage sein, eine dem jeweiligen Arbeitskontext angepaßte Benutzungsschnittstelle ohne großen Aufwand zu entwickeln und sie dem Benutzer vorzuführen, um eine direkte und schnelle Evaluierung zu ermöglichen. Dabei ist wesentlich, daß die Benutzungsschnittstelle sowohl global als auch lokal einfach und schnell zu modifizie-

[1] Vgl. Floyd 89

[2] Vgl. Keil-Slawik 90 S. 37

ren ist, da sich einerseits im Verlauf des Entwicklungsprozesses die Anforderungen an die zu entwicklende Software ändern werden und andererseits eine evolutionäre Entwicklung eine schrittweise Erweiterung des Systems vorsieht. Desweiteren muß für eine schnelle Entwicklung auch der Aspekt die Wiederverwendbarkeit einzelner Komponenten der Benutzungsschnittstelle in Verbindung mit der Erweiterbarkeit der Gestaltungsrepertoires beachtet werden.

1.1 Entwicklung von Benutzungsschnittstellen mit Toolkits

Die Programmierung graphischer Benutzungsschnittstellen mithilfe graphischer Primitive hat sich als sehr komplex und zu aufwendig für den kommerziellen Einsatz erwiesen. Eine Erleichterung bei der Realisierung der Benutzungsschnittstelle stellen Fenstersysteme dar, zu deren Leistungen neben der Verwaltung von Flächen auf dem Bildschirm auch die Bereitstellung eines Toolkits gehört. Ein Toolkit kann als eine Bibliothek von Bausteinenn angesehen werden, die jeweils eine bestimmte Interaktionstechnik verbunden mit einer Präsentationsform festlegen[3]. Beispiele für Bausteine sind Menüs, Rollbalken oder Knöpfe. Es definiert also ein für das jeweilige Fenstersystem gültiges Repertoire an Standardbausteinen, aus denen eine Benutzungsschnittstelle zusammengesetzt bzw. komponiert werden kann. Dabei können die Bausteine in ihrer Funktionalität und Präsentationsform nur im geringen Maße geändert werden. Dieses hat zum Ziel, die Entwicklung konsistenter Benutzungsschnittstellen bzgl. der Präsentation und der Dialogabläufe zu unterstützen.

Individuelle aufgaben- oder firmenspezifische Lösungen sind durch diesen Ansatz nicht möglich, da durch Komposition der Standardbausteine nur komplexere aber keine wirklich neuen Elemente erzeugt werden können. Zusätzlich kommt durch die mangelnde Erweiterungsmöglichkeit eines Toolkits eine sehr eingeschränkte Wiederverwendbarkeit hinzu. Zwar besteht die Möglichkeit zur Komposition von Elementen, diese können aber nicht in die Bausteinbibliothek aufgenommen werden und stehen damit für weitere Entwicklungen nicht zur Verfügung. Betrachtet man die Vielfalt an verschiedenen Systemen, die konzeptuell unterschiedliche Benutzungsschnittstellen unterstützen (z.B. die Schreibtischmetapher, die Werkzeugmetapher, der Kartenstapel bei HyperCard oder die Raummetapher[4]), um aufgabenspezifisch angemessene Lösungen zu ermöglichen, so stellt sich die Frage nach dem Sinn dieser Beschränkung, vor allem unter dem Aspekt der bestehenden Inkompatibilität der meisten Fenstersysteme untereinander.

Vor dem Hintergrund, daß die Realisierung der Benutzungsschnittstelle zwischen 40 bis 60% des Programmcodes eines Anwendungsprogrammes ausmacht[5], muß davon ausgegangen werden, daß ein hoher Programmieraufwand zur Entwicklung einer Benutzungsschnittstelle notwendig ist. Durch die geringe Flexibilität und Wiederverwendbarkeit der Bausteine wird

[3] vgl. Meyers 89

[4] vgl. Henderson, Card 86

[5] vgl. Lane 89

sowohl eine prototypische Vorgehensweise in der Entwicklung behindert als auch die Kreativität der Entwickler, denen das einfache und schnelle Schaffen neuer Formen und Interaktionstechniken erschwert wird, stark eingeschränkt.

User Interface Development Systems (UIDS) basieren meist auf bestehenden Bausteinbibliotheken und bieten eine Sammlung von speziellen Werkzeugen, mit denen die Entwicklung von Benutzungsschnittstellen erleichtert werden soll. Ein wesentlicher Schwachpunkt besteht darin, daß die konzeptuellen Nachteile eines Toolkits durch ein UIDS nicht behoben werden. Insbesondere wird die Entwicklung applikationsspezifischer Interaktionstechniken durch die scharfe Trennung der Benutzungsschnittstelle von der Applikation nicht unterstützt.[6]

1.2 Der objektorientierte Ansatz

Demgegenüber stellen die objektorientierten **application frameworks** einen vielversprechenden Ansatz zur Entwicklung von graphischen Benutzungsschnittstellen dar. Dabei handelt es sich um eine Bibliothek abstrakter Klassen, die einen großen Teil der Standardbenutzungsschnittstelle eines Anwendungsprogrammes beschreiben bzw. implementieren. Schon aufgrund der Eigenschaften des objektorientierten Paradigmas, wie Einkapselung, Polymorphie und Vererbung, ist eine bessere Wiederverwendbarkeit sowie Änderbarkeit und daraus resultierend eine gute Grundlage für eine prototypische Vorgehensweise gegeben. Bekannte Beispiele für application frameworks sind das Smalltalk System[7] und MacApp[8].

Die Hauptkomponenten einer Benutzungsschnittstelle werden in Klassen beschrieben, wobei durch die Entwicklung von Unterklassen Anwendungen spezialisiert werden können. Die Klassenbibliothek eines application frameworks umfaßt sowohl Klassen zur Realisierung von Bildschirmobjekten (Fenster, Menüs) als auch Klassen zur Realisierung der Applikation selbst. In dieser Erweiterung besteht der große Unterschied zu den Klassenbibliotheken der objektorientierten Toolkits, die nur die Realisierung der Bildschirmobjekte beinhalten. Beiden ist aber gemeinsam, daß sie ein bestimmtes Repertoire an Standardbausteinen vorgeben. Zwar ist es dem Entwickler durch die Bildung von Unterklassen möglich, kleinere Änderungen an den vorgegebenen Standardbausteinen vorzunehmen, es ist ihm aber nahezu unmöglich andere Standards zu verwenden bzw. neu zu entwickeln. So werden z.B. Fenster meist als ein in sich geschlossener Standardbaustein angesehen, dessen interner Aufbau und festgelegte Interaktionstechnik von dem Entwickler nur in den vorgegebenen Grenzen beeinflußt werden kann. Neue Repertoires von Standardbausteinen für spezielle Anwendungsprobleme zu entwickeln wird dadurch sehr stark erschwert. Darin ist einer der wesentlichen Schwachpunkte bestehender application frameworks zu sehen, der auf fehlende Konzepte zur Konstruktion von Standardbausteinen zurückzuführen ist.

---

[6] vgl. Knolle 89 und Hartson 89

[7] vgl. Goldberg, Robson 83

[8] vgl. Schmucker 86

2. OFS - ein objektorientiertes Fenstersystem

Im folgenden wird ein objektorientiertes Fenstersystem (OFS) zur Unterstützung in der Entwicklung interaktiver Systeme vorgestellt, das wir zur Zeit am Institut für Angewandte Informatik an der TU-Berlin entwerfen und implementieren. OFS basiert auf dem von der Firma Sun entwickelten objektorientierten Fenstersystem NeWS™, ohne allerdings die vorhandene Klassenbibliothek zu nutzen. Wir haben eine vollständig neue Konzeption der Klassenbibliothek und darauf arbeitender Werkzeuge mit dem Ziel entworfen, dem Softwareentwickler eine möglichst änderbare, wiederverwendbare und flexible Arbeitsumgebung zur Verfügung zu stellen. Die Implementierung erfolgt in der Seitenbeschreibungssprache PostScript, die im NeWS-System um einen objektorientierten Ansatz der Programmierung erweitert worden ist.

Bei der Entwicklung dieser Umgebung haben wir zunächst das Hauptaugenmerk auf die Klassenbibliothek gelegt und die Entwicklung von Werkzeugen hintenangestellt. Zur Zeit existieren neben der Klassenbibliothek mehrere kleine Werkzeuge zur interaktiven Entwicklung und Gestaltung sowohl der Benutzungsschnittstelle als auch von Teilen der Applikation. Die Klassenbibliothek basiert auf mehreren Grundkonzepten.

- ◇ Die Benutzungsschnittstelle ist vollständig durch eine hierarchische Komposition von Bausteinen bzw. Objekten beschreibbar.
- ◇ Es stehen sehr einfache Basisbausteine zur Verfügung, die ohne Einschränkung zu komplexeren Strukturen zusammengesetzt werden können.
- ◇ Komplexe Strukturen werden genauso gehandhabt wie einfache Basiselemente.
- ◇ Jeder Baustein ist über seine Position in der Kompositionshierarchie adressierbar und dadurch direkt änderbar.
- ◇ Es besteht keine scharfe Trennung zwischen Benutzungsschnittstelle und Applikation.

Bevor auf die Prinzipien der hierarchischen Komposition näher eingegangen werden kann, wird zunächst die Grundstruktur der Klassenbibliothek kurz erläutert.

Die Klassenbibliothek basiert auf der einfachen Vererbungstechnik und kann in zwei große Bereiche unterteilt werden, die Klassen zur Generierung von sichtbaren und nicht-sichtbaren Objekten (vgl. Abb. 1). Die Klassen zur Erzeugung sichtbarer Objekte beschreiben Bildschirmobjekte bzgl. der Präsentation und ggf. das Verhalten der Objekte auf Eingaben des Benutzers. Nicht-sichtbare Objekte realisieren spezielle Funktionalität oder Beziehungen zwischen Objekten. Die Klasse **Base** bildet die Basisklasse der gesamten Klassenbibliothek und stellt die Grundfunktionalität für alle Klassen im OFS-System zur Verfügung. Die sichtbaren Klassen teilen sich weiterhin auf in sensitive also auf Eingaben des Benutzers reagierende Klassen bzw. Objekte (Area, Button, Menu) und nicht-sensitive Klassen (TextString, Image, Line). Das Dia-

logverhalten ist nicht von der Beschreibung der Präsentation getrennt und beide Aspekte der Benutzungsschnittstelle werden in der selben Klasse festgelegt. Dies hat den Vorteil, daß die Struktur der Benutzungsschnittstelle auf dem Bildschirm mit der der Programmierung weitgehend übereinstimmt und dadurch ein besseres Verständnis und eine leichtere Änderbarkeit erreicht wird.

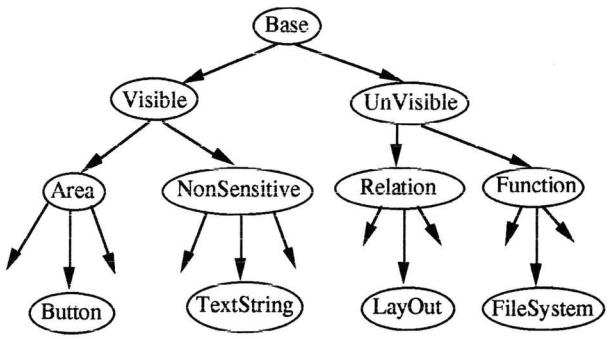

Abb.1: Auszug aus der Klassenhierarchie des OFS-Systems

Im folgenden wird die hierarchische Komposition unter dem Blickwinkel der Präsentation der Benutzungsschnittstelle erläutert. Da in dem System OFS nicht zwischen der reinen Benutzungsschnittstelle und der Applikation konzeptuell unterschieden wird, können die gleichen Techniken auch auf die applikativen Anteile angewendet werden.

2.1 Die hierarchische Komposition

Das Konzept der hierarchischen Komposition bildet die Grundlage der gesamten Klassenbibliothek. Dabei ist wesentlich, daß jedes Objekt eine beliebige Anzahl von weiteren Objekten anderer Klassen benutzen kann. Eine komplette Benutzungsschnittstelle besteht im Idealfall nur aus einer einzigen Klasse, die eine bestimmte Anzahl von Objekten benutzt, die wiederum Objekte anderer Klassen benutzen können. Die Erzeugung eines Objektes wird durch die jeweilige Klasse realisiert, wogegen die Methoden zum Entfernen und Einfügen von erzeugten Objekten zur Standardfunktionalität eines jeden Objektes im System gehören. Dadurch ist jedes Objekt ein potentieller Kandidat zur Verwaltung von weiteren Objekten, und eine explizite Unterscheidung zwischen atomaren und komplexen Klassen entfällt daher. Benutzt ein Objekt ein oder mehrere andere Objekte, so wird es **Vaterobjekt** dieser Objekte genannt. Wird ein Objekt von einem anderen Objekt benutzt, so ist es ein **Sohnobjekt** des Vaters.

In Abbildung 2 ist beispielhaft ein Objekt der Klasse **Area** abgebildet, in dem ein Objekt der Klasse **TextString** und ein Objekt der Klasse **TextButton** enthalten sind. Mithilfe der hierar-

chischen Komposition läßt sich dieser Sachverhalt einfach verdeutlichen bzw. als Unterklasse der Klasse **Area** beschreiben.

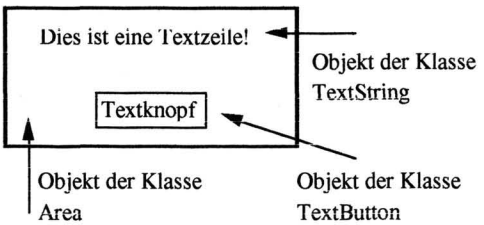

Abb.2: Ein komponiertes Bildschirmobjekt

In herkömmlichen objektorientierten Ansätzen erfolgt die Erzeugung der Sohnobjekte zur Laufzeit und geschieht genauso wie die Parametrisierung irgendwo im Programmcode der Klasse. Dieser Umstand macht es bei der Verwendung von Klassen sehr schwierig herauszufinden, welche Objekte erzeugt und in welcher Form sie parametrisiert werden. Die Möglichkeit ein Objekt zu parametrisieren, ist aber sehr wesentlich, da nicht zu erwarten ist, daß die Klasse des Objektes genau die Eigenschaften festgelegt hat, die für den speziellen Fall notwendig sind. Beispielsweise wird die Klasse **TextButton** (vgl. Abb. 1) nicht genau die Koordinaten für ihre Objekte festgelegt haben, die in diesem Beispiel erforderlich sind. Unterklassen erben die Beschreibungselemente zur Erzeugung der Sohnobjekte von ihren Oberklassen. Soll die Parametrisierung eines Sohnobjektes geändert werden, so muß die entsprechende Methode, die für die Erzeugung und Parametrisierung zuständig ist, überschrieben werden. Dies bedeutet unter Umständen einen immensen Aufwand, da auch die Teile der Methode, die gültig bleiben sollen, erneut beschrieben werden müssen.

Aus diesem Grund wurde eine spezielle Form der Beschreibung für benutzte Objekte entwikkelt, die sowohl eine gute Änderbarkeit der Parameter von Sohnobjekten als auch eine einfache Erweiterung und Verminderung der Sohnobjekte erlaubt. Diese Beschreibung ermöglicht es, alle benutzten Objekte vollständig in der Klasse des Vaterobjektes zu spezifizieren und zentral zu verwalten. Zur Spezifikation eines Sohnobjektes muß der Entwickler mindestens den Namen und die Klasse des Objektes festlegen. Darüberhinaus besteht die Möglichkeit, alle zu ändernden Parameter aufzuführen und mit einem neuen Wert zu belegen. Abbildung 3 zeigt einen Auszug aus der Beschreibung der enthaltenen Objekte für das in Abbildung 2 aufgeführte Beispiel. Diese Beschreibung erfolgt in einer Unterklasse der Klasse **Area**. Bei der Erzeugung eines Objektes dieser neuen Klasse wird die Beschreibung interpretiert und die spezifizierten Sohnobjekte mit der jeweiligen Parametrisierung erzeugt.

Jede Klasse erbt die Objektspezifikation ihrer Oberklassen und kann die Parametrisierung der zu erzeugenden Objekte verändern, ohne die schon festgelegten und nicht zu ändernden Para-

meter erneut aufzuführen. Eine solche einstufige Kompositionshierarchie innerhalb einer Klasse ist jedoch für eine gute Änderbarkeit und eine prototypische Vorgehensweise nicht ausreichend. In vielen Fällen ist es wünschenswert, Parameter von Sohnobjekten in Sohnobjekten zu verändern, ohne dafür neue Klassen für die Sohnobjekte zu definieren. Aus diesem Grund kann auch die in einem Sohnobjekt enthaltene Objektspezifikation direkt geändert werden.

Diese direkte Änderung der enthaltenen Objektspezifikationen ist nicht auf die zweite Stufe der Kompositionshierarchie begrenzt, sondern kann auf beliebiger Hierarchiestufe erfolgen. Auf Grund der unvermeidlich zunehmenden Komplexität der Beschreibung mit zunehmender Hierarchietiefe steigt der Beschreibungsaufwand stark an. Gedacht ist diese direkte Änderbarkeit nur für die Entwicklungsphase, in der es wichtig ist, verschiedene Ausprägungen der Benutzungsschnittstelle möglichst schnell und einfach zu erzeugen und zu testen. Im Endprodukt sollten mehrstufige Objektspezifikationen nicht mehr verwendet werden, sondern die gewünschte Parametrisierung als Klasse festgeschrieben sein.

```
/NeueArea Area [ ]                          % Klassendefinition
classbegin
    /#ObjectSpec                            % Definition der Sohnobjekte
      { /#ObjectSpec super send begin % Geerbte Sohnobjekte
          /Textzeile lookupdictbegin
            /ClassName    /TextString def
            /Spec lookupdictbegin
                  /$Xpos   20 def
                  /$Ypos   150 def
                  /$Spec   (Dies ist eine Textzeile!) def
            lookupdictend
          lookupdictend
      end } def
classend def
```

Sohnobjekt!

Parametrisierung

Abb.3: Spezifikation benutzter Objekte in PostScript-Syntax

Auch die vorgegebenen Standardbausteine (Fenster, Menüs, Dialogboxen) sind nach dem Prinzip der hierarchischen Komposition aus einfachen Basiselementen aufgebaut. Dadurch ist es dem Entwickler möglich, diese in ihrer Präsentation und Funktionalität ohne großen Aufwand zu verändern und seinen speziellen Anforderungen flexibel anzupassen.

2.2 Beziehungen zwischen Objekten

Die Spezifikation benutzter Objekte deckt nur einen Teil der gesamten Benutzungsschnittstelle ab. Ein wichtiger Aspekt bei der Entwicklung von Benutzungsschnittstellen sind die Beziehun-

gen zwischen Sohnobjekten im Kontext eines Vaterobjektes. Dabei gilt es zunächst **graphische** und **semantische** Beziehungen zu unterscheiden. Graphische Beziehungen betreffen alle Aspekte der relativen Positionierung, d.h. der räumlichen Abhängigkeiten zwischen Sohnobjekten; semantische Beziehungen beschreiben dagegen funktionale Abhängigkeiten, wie z.B. das Schaltverhalten einer Gruppe von Stationstasten. Desweiteren muß eine Unterscheidung in **allgemeine** und **spezielle** Beziehungen getroffen werden. Spezielle Beziehungen betreffen nur ausgewählte Sohnobjekte eines Vaterobjektes, wogegen allgemeine Beziehungen zwischen allen Objekten einer Klasse und deren Unterklassen bestehen können. Diese vier Arten von Beziehungen treten immer in Kombination auf; die anzutreffenden Beziehungen sind entweder **allgemein-graphisch, speziell-graphisch, allgemein-semantisch** oder **speziell-semantisch**. Die folgende Tabelle zeigt für jede Art von Beziehung ein mögliches Beispiel.

Beziehungen	allgemein	speziell
graphisch	Alle Sohnobjekte der Klasse **TextString** werden horizontal angeordnet.	Das Sohnobjekt **Textzeile** liegt 10 Einheiten über dem Sohnobjekt **Textknopf**.
semantisch	Wenn ein Sohnobjekt der Klasse **TextButton** aktiviert wird, so werden alle anderen Sohnobjekte der Klasse **TextButton** deaktiviert.	Wenn das Sohnobjekt **Textknopf** aktiviert wird, wird das Sohnobjekt **Textzeile** kursiv dargestellt.

Tabelle 1: Beispiele für Beziehungen zwischen Sohnobjekten

In den meisten application frameworks sind diese Beziehungen ein fester Bestandteil der jeweiligen Klasse des Vaterobjektes und nur mit Schwierigkeiten an die speziellen Anforderungen des Entwicklers anzupassen, da sie nicht explizit beschrieben werden. Unter dem Blickwinkel von Wiederverwendbarkeit, Änderbarkeit und Flexibilität ist diese Vorgehensweise als ein Nachteil in der Struktur einer Klassenbibliothek zu bewerten.

Um dieses Problem zu lösen, werden im OFS-System alle Beziehungen, die zwischen Sohnobjekten auftreten können, durch spezielle sogenannte **Beziehungsklassen** realisiert. Diese Klassen erlauben es dem Entwickler sowohl allgemeine als auch spezielle Beziehungen zwischen Objekten zu beschreiben. Objekte dieser Beziehungsklassen werden genauso wie andere sichtbare Objekte als Sohnobjekte verwaltet. Dabei ist es dem Entwickler möglich, durch eine geeignete Parametrisierung sowohl konkrete Sohnobjekte als auch die Klassen der Sohnobjekte anzugeben, die verwaltet werden sollen. Die Anzahl der Sohnobjekte, die Beziehungen realisieren, ist dabei nicht beschränkt. Durch diese Strategie werden Beziehungen zwischen den

Objekten explizit beschrieben und ermöglichen eine einfache Anpassung der Benutzungsschnittstelle an neue oder geänderte Anforderungen.

2.3 Weitere Unterstützung in der Entwicklung

Die rein textuelle Beschreibung der gesamten Benutzungsschnittstelle mithilfe der hierarchischen Komposition ist in der Regel sehr aufwendig. Deswegen wurden mehrere kleine Werkzeuge entwickelt, mit denen Objekte interaktiv am Bildschirm komponiert und parametrisiert werden können. Das wichtigste Werkzeug ist der **ObjectComposer**, mit dem Objekte erzeugt, verändert oder gelöscht werden können. Dabei ist es nicht wesentlich, ob es sich bei den Objekten um sichtbare oder nicht-sichtbare Objekte handelt. Neben der Komposition von Objekten bietet dieses Werkzeug dem Entwickler die Möglichkeit, die Funktionalität der Objekte zu aktivieren, um einzelne Komponenten oder die gesamte Benutzungsschnittstelle sofort zu testen. Neben dem ObjectComposer stehen dem Softwareentwickler noch eine Reihe weiterer Werkzeuge zur Verfügung, mit denen die Klassenhierarchie, die Parametrisierung sowohl von Klassen als auch von Objekten und das Dialogverhalten der Bildschirmobjekte interaktiv verändert werden können.

Um einen Nutzen aus der interaktiven Komposition zu ziehen, ist es allerdings notwendig, die erarbeiteten Ergebnisse abzulegen und damit wiederverwendbar zu machen. Dies geschieht nicht in einer bzgl. der Klassenbibliothek externen Datei, sondern in der Klassenbibliothek selbst. Jedes Objekt stellt eine Methode zur Verfügung, mit deren Hilfe eine neue Klasse aus dem momentanen Zustand des jeweiligen Objektes generiert werden kann. Diese Klasse wird automatisch in die Klassenbibliothek aufgenommmen und steht zur weiteren Verwendung sofort zur Verfügung. Dadurch wird gewährleistet, daß jeder neu entwickelte Baustein auf dieselbe Art und Weise wie die vorgegebenen Basisbausteine verwendet werden kann. Die neu erzeugte Klasse besitzt die Klasse des bearbeiteten Objektes als Oberklasse und umfaßt nur die Information, in der sich das veränderte Objekt von seiner eigenen Klasse unterscheidet. Dabei werden nicht nur die veränderten Parameter berücksichtigt, sondern auch die Spezifikation der Sohnobjekte bzgl. der gesamten Hierarchietiefe. Darüberhinaus ist es auch möglich, schon bestehende Klassen zu überschreiben oder zu duplizieren.

2.4 Zusammenfassung und Ausblick

Das Konzept der hierarchischen Komposition ermöglicht sowohl eine flexible und kreative Entwicklung von Benutzungsschnittstellen als auch eine einfache Erweiter- und Änderbarkeit vorgegebener Standards. Die stufenweise Spezifikation von Sohnobjekten und die Möglichkeit, Beziehungen zwischen Sohnobjekten im Kontext eines Vaterobjektes explizit zu beschreiben, unterstützen eine prototypische Vorgehensweise im Softwareentwicklungsprozeß. Vor allem die Möglichkeit allgemeine Beziehungen zwischen Objekten auszudrücken, versetzt den Entwickler in die Lage, allgemeingültige Komponenten zu entwerfen, die die Grundlage eines flexiblen application frameworks bilden. Desweiteren werden die Klassen zur Beschreibung

komplexer Bildschirmobjekte übersichtlicher und damit einfacher zu beherrschen. Die Erweiterbarkeit der Klassenbibliothek sorgt darüberhinaus für eine gute Wiederverwendbarkeit der neu entwickelten Bausteine.

Die Arbeit an dem objektorientierten Fenstersystem ist noch nicht abgeschlossen. Zur Zeit werden die prototypisch entwickelten Werkzeuge bzgl. ihrer Funktionalität und ihrer Benutzungsschnittstelle überarbeitet. Desweiteren wird eine Schnittstelle zu nicht objektorientierten Sprachen geschaffen, die sowohl die Einbindung externer Programme ermöglicht als auch die uneingeschränkte Nutzung der Klassenbibliothek in Form eines erweiterbaren Toolkits erlaubt.

Literatur :

Floyd 89 : Floyd,C. : Softwareentwicklung als Realitätskonstruktion. In: Lippe,W.-M. (Hrsg) : "Softwareentwicklung. Konzepte Erfahrungen, Perspektiven - Proc. der GI-Fachtagung", Springer Verlag, 1989

Goldberg, Robson 83 : Goldberg,A., Robson,D. : SMALLTALK-80, The Language and its Implementation. Addison Wesley, Reading, Mass. 1983

Hartson 89 : Hartson,R. : User-Interface Management Control and Communication. IEEE Software, Januar 89, pp.62 - 70

Henderson, Card 86 : Henderson,D.A., Card,S.K. : Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface. In: ACM Transactions on Graphics, Vol.5, No.3, July 1986, pp.211 - 243

Keil-Slawik 90 : Keil-Slawik,R. : Konstruktives Design Ein ökologischer Ansatz zur Gestaltung interaktiver Systeme. Habilitationsschrift an der TU Berlin 1990

Knolle 89 : Knolle,N.T. : Why Object-Oriented User Interface Toolkits Are Better. In: Journal of Object-Oriented Programming, November/December 1989, pp.63 -67

Lane 89 : Lane,A. : Domesticating Microsoft Windows. In: Byte, Juni 89, pp.205-207

Myers 89 : Myers,B.A. : User-Interface Tools : Introduction and Survey. In: IEEE Software, Januar 89, pp.15-23

Schmucker 86: Schmucker,K.J. : Object-Oriented Programming for the Macintosh. Hayden Book Company 1986

Michael Castner
Technische Universität Berlin
Franklinstr. 28/29
Institut für Angewandte Informatik
Sekr. FR 5-6
1000 Berlin 10