1. Einleitung und Überblick

Dieser Aufsatz beschäftigt sich mit dem Problem der Spezifikation von Dialogen auf graphischen Benutzeroberflächen. Für den Entwurf und die Abstimmung solcher Dialoge mit den Anwendern wird heute oft Prototyping statt einer Spezifikation verwendet. Prototyping stößt allgemein auf hohe Akzeptanz bei Anwendern, wirft jedoch in technischer Hinsicht, besonders für die Konstruktion großer Softwaresysteme, Probleme auf.

Graphische Benutzeroberflächen sind geradezu "natürlich" objektorientiert. Es liegt daher nahe, sie genau wie Anwendungskernobjekte zu spezifizieren. In den meisten Werken zur objektorientierten Spezifikation und Konstruktion von Informationssystemen sucht man jedoch vergeblich nach Methoden zur Spezifikation von Benutzeroberflächen.

Ausgangspunkt für diesen Aufsatz war ein Projekt, in dem unter anderem eine graphische Benutzeroberfläche realisiert werden muß. Für die Anwendungskernobjekte ist eine eingeführte, objektorientierte Spezifikationsmethode [BarDen 93] mit zugehörigen Werkzeugen und Generatoren vorhanden. Es liegt also nahe, auch die graphische Benutzerschnittstelle zu spezifizieren, um einen Methodenbruch zu vermeiden. Bei sd&m existieren reichhaltige praktische Erfahrungen mit der dokumenten- und objektorientierten Spezifikation von betrieblichen Informationssystemen[1] [Denert 93a, Denert 93b, Denert 91]. Diese Methoden beschränken sich nicht nur auf Anwendungskernobjekte, sondern decken auch Dialoge für blockorientierte HOST-Terminals ab. Dieser Aufsatz beschreibt, wie diese Methoden für die Spezifikation von graphischen Benutzeroberflächen erweitert wurden.

In diesem Aufsatz werden zunächst bekannte Modelle vorgestellt, die für Anwendungen mit Dialogen auf graphischen Benutzeroberflächen relevant sind. Es werden dann einige Beobachtungen diskutiert, die die Erweiterung der vorhandenen Spezifikationsmethode erleichtert haben. Im Abschnitt 4 wird die verwendete Spezifikationssprache für Dialog-Objekttypen an Ausschnitten aus Spezifikationen vorgestellt.

Die abschließende Diskussion beschäftigt sich mit der Frage, ob und wann die Spezifikation einer graphischen Benutzeroberfläche eine Ergänzung zum Prototyping oder ein Ersatz für das Prototyping ist. Eine Spezifikation soll das

[1] Wir verstehen darunter große Informationssysteme, die Unternehmen auf der Basis von Unternehmensdaten- und Vorgangsmodellen bei der Informationsverarbeitung unterstützen. Solche Systeme sind meist sehr umfangreich, aber im Vergleich zum Beispiel zu einem Textverarbeitungssystem wenig komplex.

Verhalten des mit ihr beschriebenen Systems *exakt*, und für den Anwender *verständlich* beschreiben. Dies führt zu einem Zielkonflikt zwischen Genauigkeit und Verständlichkeit. Bei der Frage, wann schwerpunktmäßig Prototyping, und wann besser Spezifikationen eingesetzt werden, hilft die Betrachtung der Komplexität des zu implementierenden Systems.

2. Modelle und Spezifikationsmethoden

Sucht man in der Literatur zur objektorientierten Spezifikation und Konstruktion von Informationssystemen nach Methoden zur Spezifikation von Dialogen auf graphischen Benutzeroberflächen, so wird man selten fündig werden. Selbst Werke, deren Titel erwartungsfroh stimmen[2], bieten keinen "cookbook-approach". Kochbücher für die Spezifikation von Anwendungskernobjekten sind leichter zu finden.

Den meisten Spezifikationsmethoden liegen Modelle zugrunde. Wenn es also kein fertiges Kochbuch gibt, dann gibt es vielleicht wenigstens Zutaten, aus denen man seine Rezepte zusammenstellen kann. Dieser Abschnitt stellt kurz gebräuchliche Modelle zum Thema vor. Ferner wird stichpunktartig die TOLEDO-Methode zur objektorientierten Spezifikation vorgestellt.

2.1. Seeheim-Modell und Anwendungsarchitektur

Das Seeheim-Modell[3] beschreibt eine Software-Architektur, bei der die Benutzerschnittstelle von den Objekten des Anwendungskerns getrennt ist (siehe Abbildung 1). Die Benutzerschnittstelle selbst gliedert sich nach dem Seeheim-Modell in

- *Präsentationsobjekte*, die für die physische Repräsentation der Daten an der Benutzerschnittstelle verantwortlich sind.

- *Dialogkontrollobjekte*, die für die Kontrolle der Interaktion zwischen dem Benutzer und der Anwendung verantwortlich sind.

Eine solche oder verwandte Schichtenarchitektur ist allgemein für die objektorientierte Entwicklung betrieblicher Informationssysteme akzeptiert. Auf dieser Architektur kann eine umfassende Methodik zur Spezifikation aufgesetzt werden, die auch die Spezifikation von Dialogen beinhaltet.

---

[2] Zum Beispiel [AndGre 92],[Larson 92].

[3] Siehe [Pfaff 83]

Die Stärke des Modells ist die Trennung und damit die Unabhängigkeit von Benutzerschnittstelle und Anwendungskern. Dies führt in der Praxis zu besser modularisierten und besser wartbaren Softwaresystemen.

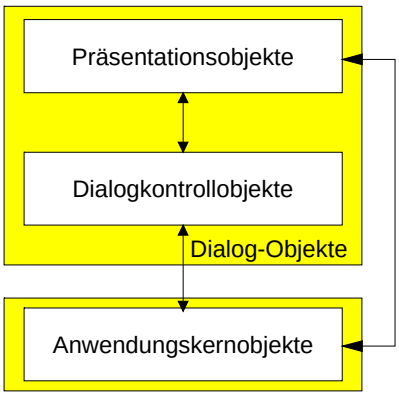

Bild 1: Anwendungsarchitektur nach dem Seeheim-Modell

Die Trennung von Präsentations- und Dialogkontrollobjekten ist jedoch für die Spezifikation graphischer Benutzeroberflächen nicht sehr zweckmäßig. Objekte, die für die Ein- und Ausgabe von Anwendungsdaten zuständig sind, haben meist selbst eine nichttriviale Zustandssteuerung. In dem noch folgenden Beispiel wurde diese Trennung deshalb aufgehoben.

2.2. Model-View-Controller Architektur

Aus der Smalltalk-Welt stammt die Model-View-Controller Architektur[4]. Die Objekte eines Systems lassen sich dabei in die drei in Abbildung 2 dargestellten Bereiche teilen:

- Das *Model* enthält die Objekte des Anwendungskerns.
- *Views* präsentieren die Objekte.
- *Controller* werden verwendet, um Daten einzugeben und um das Model zu Handlungen zu veranlassen.

Die Model-View-Controller Architektur erfordert den Austausch von Botschaften, um Model, Views und Controller konsistent zu halten.

Auf der Model-View-Controller Architektur basiert keine gebräuchliche Spezifikationsmethode. Die Stärken der Architektur werden dann ausgespielt,

[4] Siehe zum Beispiel [KraPop 88].

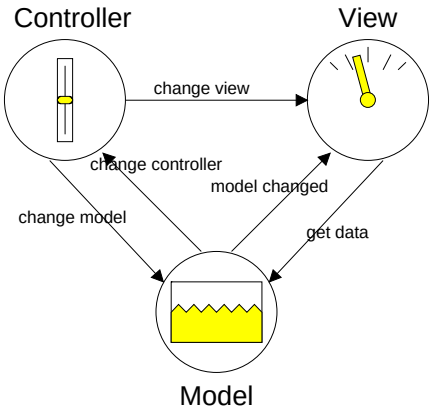

Bild 2: Model-View-Controller Paradigma

wenn es mehrere Views oder Controller gleichzeitig zu einem Anwendungskernobjekt (Model) gibt. Diese werden dann konsistent gehalten. Die Ideen der Model-View-Controller Architektur können also auch in Systeme eingebaut werden, die z.B. nach dem Seeheim-Modell strukturiert sind.

2.3. TOLEDO - Methode

Die TOLEDO-Methode [HöTe 93] ist eine objektorientierte Spezifikationsmethode. TOLEDO faßt Funktionenmodell und Datenmodell zu Anwendungskernobjekten zusammen. Dialogobjekte werden nach dem Seeheim-Modell in Interaktions[5]- und Präsentationstypen aufgegliedert.

Objekte werden in Form von Dokumenten[6] beschrieben. Diese Dokumente[7] sind halbformal und durch Schlüsselworte gegliedert. Objekte in TOLEDO haben Eigenschaften und Fähigkeiten. Sie unterliegen einem Zustandsmodell (Lebenslauf). Sie benutzen Vererbung und Polymorphismus.

Die zugehörige Methode zur Spezifikation der Dialog-Benutzerschnittstelle hat sich gegenüber dem von Denert [Denert 91] beschriebenen Vorgehen im Prinzip nicht geändert. Sie ist im wesentlichen für die Spezifikation von Dialogen auf HOST-Terminals, und nicht für GUI[8]-Oberflächen ausgelegt.

---

[5] Entspricht den Dialogkontrollobjekten des Seeheim-Modells.

[6] Die Spezifikation erfolgt nicht im starren Rahmen von CASE-Werzeugen. Statt dessen wird Softwareentwicklung als das Erstellen und die Transformation und Verfeinerung von Dokumenten begriffen. Zu diesem Vorgehen siehe [Denert 93a].

[7] Allgemein zugängliche Beispiele finden sich in [BarDen 93], [Denert 93b] und in [Scholz 93].

[8] Graphical User Interfaces.

TOLEDO mußte daher für die Spezifikation von GUI-Dialogen ergänzt werden. Die grundlegende Methodik der Beschreibung von Dialogen als Objekttypen in Form von Dokumenten war jedoch schon vorhanden. Der Änderungsbedarf leitet sich aus den im folgenden Abschnitt geschilderten Beobachtungen ab.

3. Eigenschaften von GUI-Dialogen

Benutzerschnittstellen, die mit graphischen Benutzerschnittstellen (GUIs) realisiert werden, müssen sich nicht wesentlich von Dialogen auf HOST-Terminals unterscheiden. Diese Dialoge kann man wie gehabt spezifizieren. Man kann aber mit GUIs wesentlich komplexere Dialoge realisieren. Dafür sind andere Spezifikationsmethoden erforderlich. Im folgenden werden zunächst GUI-Dialoge in Komplexitätsklassen eingeteilt. Dann werden die Implikationen diskutiert, die sich aus dem Vorhandensein von Klassenbibliotheken für graphische Benutzerschnittstellen ergeben. Schließlich werden Interaktionsdiagramme, eine Darstellungsform für Automaten, auf ihre Eignung für die Spezifikation von GUI-Dialogen untersucht.

3.1. Komplexitätsklassen von GUI-Dialogen

Beim Entwurf von GUI-Dialogen müssen, je nach Komplexität der Dialoge, Akzente in der Methodik gesetzt werden. Hier wird daher zwischen drei Komplexitätsklassen von GUI-Dialogen unterschieden.

Massendialoge auf graphischen Benutzeroberflächen

Unter Massendialogen verstehen wir einfache Dialoge mit einfachen Zustandsmodellen. Solche Dialoge sind mit einem anderen Aussehen, aber mit im Prinzip gleichem Verhalten, auch auf HOST-Terminals ohne Fenstertechnik realisierbar. Sie sind stark modal - es kann also immer nur in einer Maske zu einem Zeitpunkt gearbeitet werden. Die Zustandsübergangsmatrix ist schwach besetzt.

Betriebliche Informationssysteme[9] bestehen heute zum überwiegenden Teil aus solchen Dialogen. Solche Systeme mit hunderten von Dialogen lassen sich auf wenige, einfache Dialogtypen[10] reduzieren. Die komplette Ablauflogik kann dann in den Dialogtypen beschrieben werden.

[9] Beispiele für solche Informationssysteme sind die typischen betriebswirtschaftlichen Anwendungen, wie Buchhaltungssysteme, Materialwirtschaftssysteme, Auftragserfassungssysteme usw.

[10] Üblich sind ca. drei bis fünf Dialogtypen.

Will man solche Dialoge rationell erstellen, benötigt man Generierungswerkzeuge[11] und eine möglichst komprimierte Spezifikationsmethode[12], die es erlaubt, die Gemeinsamkeiten aller Dialoge eines Typs objektorientiert in einem Template zu spezifizieren und je Dialog ansonsten nur die wirklich spezifischen Eigenschaften dieses Dialoges zu beschreiben.

Abbildung 3 zeigt eine komprimierte Dialogspezifikation. Es wird ein Dialogschema _PflegeDialog mit einem Anwendungstyp _Kunde, seinem Schlüsseltyp _KdNummer, einem Schlüsselauswahldialog _KdAuswahlDialog und den benötigten Masken instantiiert. Das Dialogschema enthält die Ablaufsteuerung. Alle Objekttypen, mit denen er instantiiert wird, müssen gesondert spezifiziert werden, können aber auch generiert sein.

```
JEDER   _KundenPflegeDialog
IST_EIN         _PflegeDialog(
                        _Kunde,
                        _KdNummer,
                        _KdAuswahlDialog,
                        _KundeAnzeigenMaske,
                        _KundeAnlegenMaske,
                        _KundeÄndernMaske,
                        _KundeLöschenMaske)
```

Bild 3: Spezifikation eines Pflegedialoges

Solche Dialoge sind nicht der primäre Gegenstand dieses Aufsatzes. Sie können mit bekannten Methoden spezifiziert werden.

Individualdialoge auf graphischen Benutzeroberflächen

Unter diese Kategorie fallen solche GUI-Dialoge, die die Möglichkeiten der graphischen Benutzerschnittstellen ausschöpfen. Typisch sind Zustandsmodelle mit dicht besetzten Zustandsübergangsmatrizen. Die Dialoge sind typisch nicht modal. Es können mehrere Fenster gleichzeitig geöffnet sein. Beispiele für solche Oberflächen sind CASE-Tools, einfache Diagrammer etc.

Typisierung ist bei *Individualdialogen* oft weniger wichtig, da es sich eher um Einzel- und nicht um Massenentwicklungen handelt. Wichtig ist hingegen die Einbindung schon vorhandenen Verhaltens einer Klassenbibliothek über Vererbung. Das noch folgende Beispiel ist eine Anwendung dieser Art.

---

[11] Siehe dazu zum Beispiel [Scholz 91] und [Scholz 93].

[12] Siehe dazu zum Beispiel [Keller 93]

Hochkomplexe Dialoge auf graphischen Benutzeroberflächen

Hierbei handelt es sich um Textverarbeitungssysteme, Tabellenkalkulationssysteme und ähnliche Anwendungen. Die Zustandsmodelle haben eine enorme Anzahl von Zuständen. Gleichzeitig gibt es hunderte von Ereignissen und eine im allgemeinen dicht besetzte Zustandsübergangsmatrix.

Das Seeheim-Modell ist für solche Dialoge im Gegensatz zu den beiden oben dargestellten Dialogarten oft nicht mehr geeignet [Larson 92]. Solche Dialoge werden im folgenden nicht weiter untersucht, da wir über keine breiten Erfahrungen mit ihnen verfügen.

3.2. GUI-Klassenbibliotheken sind objektorientiert

Für die Programmierung graphischer Benutzeroberflächen werden heute vor allem objektorientierte Klassenbibliotheken verwendet. Diese Klassenbibliotheken stellen fertige Klassen zum Beispiel für Felder, Buttons, Dialogfenster, MDI-Interfaces usw. zur Verfügung. Will man spezifisches Verhalten erreichen, leitet man seine eigenen Klassen von einer dieser vordefinierten Klassen ab.

Abbildung 4 verdeutlicht diesen Ansatz der GUI-Bibliotheken. Objekte, wie ein Fenster, sind selbst wieder aus Objekten zusammengesetzt. Die komplette sichtbare Anwendung kann man sich auch als Baum von Objekten vorstellen. Es ist also nur natürlich, wenn man zur Spezifikation eine objektorientierte Methodik verwendet, und zumindest in späten Spezifikationsphasen auf vorgefertigte Oberflächenklassen Bezug nimmt.

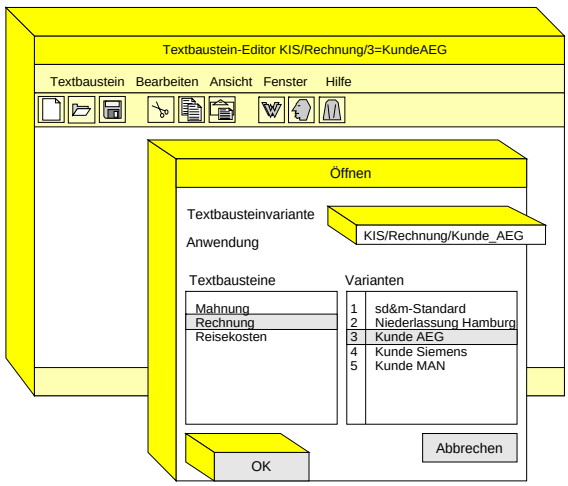

Bild 4: Objekte in GUI-Oberflächen

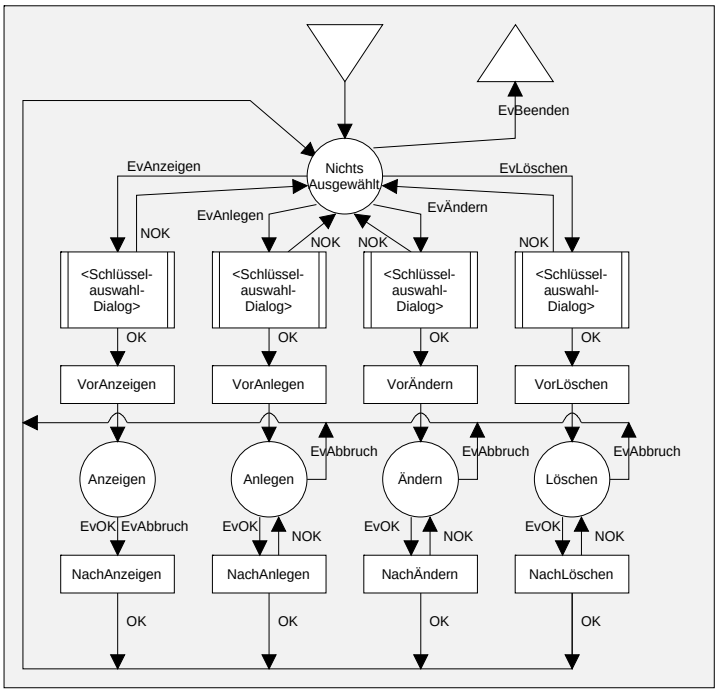

Bild 5: Interaktionsdiagramm

3.3. Dialoge als Automaten - Interaktionsdiagramme

Dialoge können als Automaten[13] beschrieben werden. Die graphische Darstellung solcher Automaten wird Interaktionsdiagramm genannt. Abbildung 5 zeigt ein solches Interaktionsdiagramm für einen einfachen Einsatzpflegedialog. Ein solcher Dialog ist der Gruppe der *Massendialoge* zuzuordnen.

Der Dialog hat mehrere Zustände (dargestellt durch Kreise). In jedem dieser Zustände wird genau eine Maske gezeigt. Die Zustandsübergänge erfolgen durch Ereignisse (virtuelle Tasten), die in der Regel vom Benutzer ausgelöst werden. Dabei werden Aktionen (dargestellt als Rechtecke) ausgeführt. Unterdialoge sind als Rechtecke mit doppeltem vertikalem Rand dargestellt.

Das Bild vermittelt gut, daß schon ein einfacher Dialog zu einem recht komplexen Diagramm führt. Nun verfügt dieser Dialog noch über wenige mögliche Ereignisse und eine schwach besetzte Zustandsübergangsmatrix.

[13] Zum Begriff siehe z.B. [HopUll 79], zur Anwendung für Dialoge siehe zum Beispiel [Larson 92] oder [Denert 91]. Komplexere Interaktionen lassen sich auch noch als Kellerautomaten (Kontextfreie Grammatiken) darstellen. Für unsere Zwecke sollen aber endliche deterministische Automaten genügen.

Ein Bild nach derselben Systematik für einen Individualdialog ist meist nicht mehr wartbar oder lesbar. Damit sind schon Individualdialoge nicht mehr vernünftig graphisch spezifizierbar. Es verbleiben Prototyping oder eine textuelle Repräsentation.

4. Spezifikation von GUI-Dialogen

Im folgenden werden möglichst wenige, aber mächtige Konstrukte eingeführt, die zusätzlich zu einer normalen Spezifikationssprache für Anwendungskernobjekte für die Spezifikation von Dialogen benötigt werden.

Um im Rest des Kapitels Ausschnitte aus einer Beispielspezifikation diskutieren zu können, wird ein kurzer Überblick über das vorgestellte Anwendungsbeispiel gegeben. Danach werden entsprechende Ausschnitte aus der Spezifikation diskutiert.

4.1. Sprachmittel

Die Konzepte, die man für eine objektorientierte Spezifikation von Massen- und Individualdialogen benötigt, sind schon erwähnt:

(1) Dialoge sind Objekte. Dabei muß in der Methodik nicht zwischen Fenstern, Buttons und anderem unterschieden werden (siehe Abbildung 4).

(2) Eine Unterscheidung zwischen Präsentations- und Dialogkontrollobjekten ist im allgemeinen verzichtbar, kann aber eingeführt werden, wo dies sinnvoll erscheint.

(3) Die Dialogobjekte können von Klassen aus GUI-Klassenbibliotheken abgeleitet werden.

(4) Dialogobjekte sind Automaten, die auf Ereignisse reagieren. Dabei können in einem Zustand beliebige andere Dialogobjekte sichtbar sein.

(5) Für die Spezifikation von Massendialogen sind generische Objekttypen[14] notwendig.

[14] Dabei handelt es sich um volle abstrakte Datentypen mit Typparametern. In C++ werden diese als Templates realisiert.

Mit unserer normalen Spezifikationsmethodik[15] waren die Punkte (1) bis (3) abdeckbar. Lediglich für die Punkte (4) und (5) mußten kleine Ergänzungen vorgenommen werden.

4.2. Beispiel: Editor für Textbausteine

Bei der Anwendung, aus deren Spezifikation im folgenden Ausschnitte gezeigt werden, handelt es sich um eine Editor für Textbausteine eines Dokumentenerzeugungssystems. Abbildung 4 zeigt das Anwendungsfenster zusammen mit dem Dialog zum Öffnen eines Textbausteins. Die Textbausteine sind in einer Datenbank abgelegt. Sie können mit einem Textsystem (Standardsoftware) editiert werden, verfügen aber auch über Eigenschaften, die nicht im Textbaustein selbst, sondern nur in der Datenbank abgelegt sind. Rund um die Textbausteine gibt es weitere Begriffe und Objekte, die ebenfalls über die Benutzerschnittstelle manipuliert werden müssen. Auf die Textbausteine kann über hierarchische Ordnungsbegriffe zugegriffen werden. Dies sind die "Anwendungen", für die der Textbaustein verwendet wird, und der Name des Textbausteins. Ferner sind zu Textbausteinen noch Varianten zugelassen.

Im Bausteineditor kann jeweils ein Textbaustein bearbeitet werden. Auf die Textbausteine gibt es verschiedene Sichten:

- *Layout-Sicht*: In ihr kann der Textbaustein mit dem Textsystem bearbeitet werden.
- *Baustein-Varianten-Sicht*: In ihr können Datenbank-Attribute einer Textbausteinvariante bearbeitet werden.
- *Baustein-Sicht*: In ihr können Datenbank-Attribute bearbeitet werden, die allen Varianten des Bausteins gemeinsam sind.

Je nach Zustand der Anwendung können alle drei Sichten gleichzeitig geöffnet sein und aufeinander wirken.

4.2.1. Spezifikation der Anwendungsklasse

Die komplette Anwendung wird in Form von Objekttypen spezifiziert. Der Textbaustein-Editor ist ein Objekttyp. Dieser Objekttyp hat Referenzen auf alle Sichten des Editors und vor allem auch auf das bearbeitete Anwendungsobjekt.

[15] Siehe [Denert 91] und [BarDen 93]

Daneben besitzt der Objekttyp alle Kontrollelemente, über die Ereignisse ausgelöst werden können. Abbildung 6 zeigt Auszüge.

Dabei wird ein Sekundärfenster genauso als Eigenschaft (Attribut) betrachtet, wie ein Button oder das bearbeitete Anwendungsobjekt.

JEDER	_TextbausteinEditor		
IST_EIN	MDIPrimaryWindow		
BESCHREIBUNG	Der ^_TextbausteinEditor ist das primäre Anwendungsfenster für eine Benutzerschnittstelle für Textbausteine. Er erlaubt es, jeweils einen Textbaustein und die mit ihm zusammenhängen Objekte zu bearbeiten. Dazu verfügt er über die folgenden Sichten auf ein Objekt:		
HAT	EditierterBaustein	--->	_TextbausteinVar
	Die jeweils bearbeitete Textbausteinvariante, also das Anwendungsobjekt, das im ^_TextbausteinEditor zu bearbeiten ist.		
HAT	VariantenSicht	--->	_TBVariantWindow
	Sekundärfenster, das die für eine Textbausteinvarianten spezifischen Informationen anzeigt.		
HAT	StoreButton	--->	_PushButton
	Button zum Speichern des Textbausteines - untergebracht in der Funktionsleiste Icon: 		

Bild 6: Auszüge aus einer Objekttyp-Beschreibung

Im Abschnitt *Beschreibung* kann ein Überblick über das Aussehen der Anwendung mit Screen-Dumps oder ähnlichem gegeben werden. Graphiken, die dabei verwendet werden, müssen nicht formal sein, wenn sie nicht maschinell ausgewertet werden.

4.2.2. Die Anwendung als Automat

Wie schon erwähnt, werden Dialogobjekte als Automaten[16] spezifiziert. Ein Automat ist definiert als ein Tripel. Dieses besteht aus:

- einer Menge von Zuständen.
- einem Eingabezeichenvorrat. Dies sind hier die möglichen Ereignisse, auf die der Dialogautomat reagieren muß.
- einer Menge von Zustandsübergängen, die angeben, wie der Automat in welchem Zustand auf welches Ereignis reagiert. Dabei können Aktionen ausgelöst werden.

Durch Zustände, Zustandsübergänge und Aktionen ist das Verhalten der Anwendung vollständig spezifiziert. Oft ist es auch sinnvoll, das Zustandsübergangsdiagramm als Bild in das Dokument einzufügen.

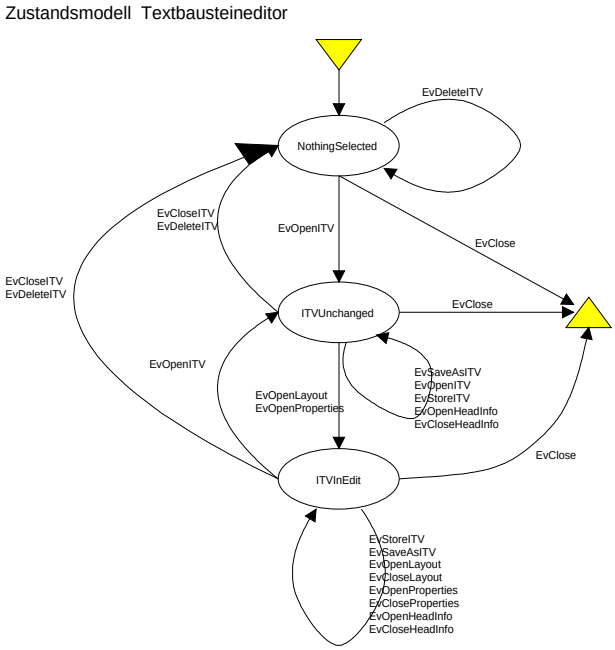

Bild 7: Zustandsübergangsdiagramm

Dabei werden im Bild 7 nicht alle Zustandsübergänge des Primärfensters, sondern nur die wesentlichen Zustandsübergänge angegeben. Solche Bilder sollen der Klarheit dienen. Die eigentlich präzise Beschreibung ist die textuelle

16 Zum Begriff siehe [HopUll 79].

Repräsentation des Automaten. Würde man hier mit einem CASE-Tool arbeiten, hätte man meist nicht die Freiheit, das Wesentliche darzustellen. Man erhält dann oft vollständige Bilder, deren Aussagekraft aber schwindet.

4.2.3. Beschreibung der Zustände

Die Beschreibung der Zustände erfolgt so, daß man sich eine möglichst gute intuitive Vorstellung davon machen kann, wozu der Zustand dient, welche Aktionen in ihm erlaubt sind, und wie der Dialog in dem Zustand aussieht.

ZUSTAND	ITVUnchanged
BESCHREIBUNG	Der Dialog befindet sich in einem Zustand, in dem ein Textbaustein ausgewählt wurde. Der Textbaustein wurde aber sicher noch nicht verändert. Das Fenster mit der Beschreibung der Textbausteinvariante ^VariantenSicht kann schon geöffnet sein.

Bild 8: Beschreibung eines Zustandes

Die textuelle Beschreibung kann durch Bilder und sonstige informelle Darstellungen angereichert werden.

4.2.4. Beschreibung der Ereignisse

Ereignisse (die Eingabezeichen des Automaten) fassen mehrere technische Ereignisse zusammen, die der Benutzer durch das Betätigen von Dialogkontrollelementen der Oberfläche auslösen kann:

DAS_EREIGNIS	EvOpenITV
WIRD_AUSGELÖST_DURCH	Auswahl von Menüpunkt Textbaustein/Öffnen oder ^OpenDOCButton. Beabsichtigte Aktion: Öffnen eines schon existenten Textbausteines

Bild 9: Beschreibung von Ereignissen

Es ist typisch, daß ein und dasselbe Ereignis durch verschiedene Aktionen an der Oberfläche ausgelöst werden kann. Im Beispiel wird das Ereignis durch einen Menüpunkt oder einen Button der Symbolleiste ausgelöst.

4.2.5. Beschreibung der Zustandsübergänge

Die Zustandsübergänge beschreiben das Verhalten des Systems. Hier wird allgemeinverständlich oder in Form von Pseudocode beschrieben, wie sich die Anwendung zu verhalten hat.

AUF	EvOpenITV
IN	ITVNothingSelected, ITVUnchanged
FOLGT	Auswahl einer existenten Textbaustein-Variante und Übergang in den Zustand ^ITVUnchanged. Nach Auswahl wird das Sekundärfenster ^EditierterBaustein gezeigt. **PSEUDOCODE:** WENN ^SelectVariantDialog("Öffnen",OpenIt) DANN ..... und so weiter

Bild 10: Spezifikation eines Zustandsüberganges

Dabei sollte die Syntax so gewählt sein, daß sich flexibel auch Mengen von Zustandsübergängen beschreiben lassen:

AUF	EvPileSubWindows, EvArrangeSubWindows, EvArrangeIcons, EvSubWindowsVertical
IN	**JEDEM_ZUSTAND**
FOLGT	Weiterreichen des Events an die Vaterklasse ^_MDIPrimaryWindow, da dieses Fenster schon über die Standardfunktionalität für das Arrangieren seiner Kindfenster verfügt.

Bild 11: Simultane Spezifikation mehrerer Zustandsübergänge

4.2.6. Modularisierung durch Fähigkeiten

Dialogobjekte können Fähigkeiten besitzen, die zur Modularisierung von Aktionen dienen, die in mehreren Zustandsübergängen gleichermaßen verwendet werden. Abbildung 12 zeigt die Spezifikation einer Fähigkeit.

KANN	**LoadWidgets**
EFFEKT	Die Dialogfelder werden mit der Information aus ^EditierterBaustein geladen. .....

Bild 12: Auszug aus der Spezifikation einer Fähigkeit

4.2.7. Diskussion

In den obigen Abschnitten wurde gezeigt, wie man Dialogobjekte als normale Objekttypen mit einem Zustandsmodell spezifizieren kann. Dabei können Primärfenster, Sekundärfenster, modale Unterdialoge, Widgets und Anwendungskernobjekte durchgängig mit derselben Methode spezifiziert werden. Die Dialogobjekte unterscheiden sich von normalen Anwendungsobjekten nur durch die Betonung des Zustandsmodells.

Damit können Anwendungskernobjekte und Dialogobjekte mit einer durchgängigen Methode beschrieben werden. Auswertungswerkzeuge und Generatoren haben eine gemeinsame Basis.

Zu den Klassenbibliotheken für Oberflächen existiert ebenfalls kein methodischer Bruch, da die spezifizierten Dialogklassen von Standard-Oberflächenklassen abgeleitet werden können.

Die Methode unterstützt objektorientiertes Vorgehen und bildet einen Ausgangspunkt für die Generierung von Klassenrahmen für die Realisierung.

5. Spezifikation als Alternative zum Prototyping

Nachdem eine Methode vorgestellt wurde, mit der man dokumenten- und objektorientiert Dialoge auf graphischen Benutzeroberflächen spezifizieren kann, bleibt noch der Untertitel dieses Aufsatzes zu diskutieren: Ist diese Methode *eine Alternative zum Prototyping*?

5.1. Prototyping contra Spezifikation

Wie eingangs erwähnt, ist Prototyping die am häufigsten angewandte Methode zur Spezifikation graphischer Benutzeroberflächen. Dies hat seinen Grund sicher in der **hohen Akzeptanz** beim Benutzer. Prototyping als alleiniges Spezifikationsmittel ist jedoch nicht unproblematisch:

Prototyping fördert Problemstrukturierung nicht: Viele Tools für die schnelle Erstellung von GUI-Oberflächen verfolgen einen User-Point-Ansatz: Zu einem Kontrollelement der Oberfläche kann man als sog. User-Point eine isolierte Aktion in einen Programmrahmen eintragen. Dieses Vorgehen zwingt nicht zum Denken in Zustandsmodellen, Klassen und Vererbungshierarchien. Wird ein solcher Prototyp unkritisch zur fertigen Anwendung weiterentwickelt, so können schlecht wartbare und schlecht strukturierte Programme entstehen.

Prototyping fördert Modularisierung nicht: Gerade in GUI-Oberflächen gibt es meist mehrere Möglichkeiten, ein- und dasselbe Ereignis auszulösen. Das führt dazu, daß ein- und dieselbe Aktion in mehrere User-Points geschrieben werden muß. Diese User-Points müssen konsistent gewartet werden. Wird hier nicht mit großer Sorgfalt und Aufmerksamkeit gearbeitet, kann dies schnell zu schlecht strukturierter Software führen.

Wartbarkeit bei Massenänderungen: Werden Programme in großen Mengen mit GUI-Interface-Buildern erstellt, so birgt dies zwei Gefahren:

- Die Überprüfung von Styleguides wird aufwendig, weil die Dialoge nicht notwendig auf einem gemeinsamen von einer Vaterklasse ererbten Stil basieren.

- Konsistente Änderungen über mehrere Dialoge des gleichen Typs werden zur kostspieligen Tortur.

Medienbruch bei der Realisierung: Wenn nach dem Prototyping eine Implementierung ohne eine visuelle Programmierumgebung erfolgt, sind die Ergebnisse der Spezifikationsphase maschinell nicht mehr oder nur noch schlecht weiterzuverarbeiten.

Testbarkeit: Wird ein Prototyp zu einer fertigen Implementierung weiter verfeinert, so ist keine Dokumentation vorhanden, gegen die getestet werden kann.

Ist Prototyping aus dieser rein technischen Argumentation heraus also generell abzulehnen? Sicher nicht. Das Problem von relativ formalen Spezifikationen ist ihre mangelnde Akzeptanz bei den Anwendern.

Eine Spezifikation — egal ob als Prototyp, dokumentenorientierte Spezifikation oder Entwurf in einem CASE-Tool — ist Grundlage des Vertrages über den Leistungsumfang der zu erstellenden Software. Dieser Vertrag wird vor der Realisierung zwischen Softwareentwicklern und Anwendern geschlossen. Der Anwender muß also aus der Spezifikation erfahren, was er bekommt. Als Softwareentwickler sollten wir uns dabei immer daran erinnern, daß es dem Kunden Freude bereiten muß, mit uns zusammenzuarbeiten.

Wenn es den Softwareentwicklern nicht gelingt, eine Beschreibungsform zu finden, die der Anwender versteht und die sich zugleich in qualitativ hochwertige Software umsetzen läßt, die der Anwender erwartet, so ist dies vor allem das Problem der Softwareentwickler.

Man hat hier den bekannten Zielkonflikt zwischen Verständlichkeit und Genauigkeit vor sich. Das folgende kleine Modell, das auf den oben beschriebenen Komplexitätsklassen von GUI-Applikationen beruht, soll eine Richtung zeigen, wie dieses Dilemma gelöst werden kann.

5.2. Entscheidungshilfe: Spezifikation oder Prototyping

Angesichts der mangelnden Akzeptanz technischer Spezifikationen bei Anwendern von EDV-Systemen steht man vor dem Dilemma, daß man eigentlich eine Spezifikation **und** einen Prototyp benötigt. Dies ist nicht verwunderlich - schließlich erstellen auch Architekten für ihre Kunden Modelle, die sicher nicht als Grundlage für eine solide Baurealisierung dienen können. Umgekehrt würde niemand auf die Idee kommen, einem Bauherren technische Detailkonstruktionen zumuten zu wollen.

Wo der Schwerpunkt zu legen ist, beim Prototyp oder bei der Spezifikation, hängt nicht zuletzt von der Art der zu realisierenden Benutzeroberfläche ab. Abbildung 13 zeigt tendenzielle Gewichte.

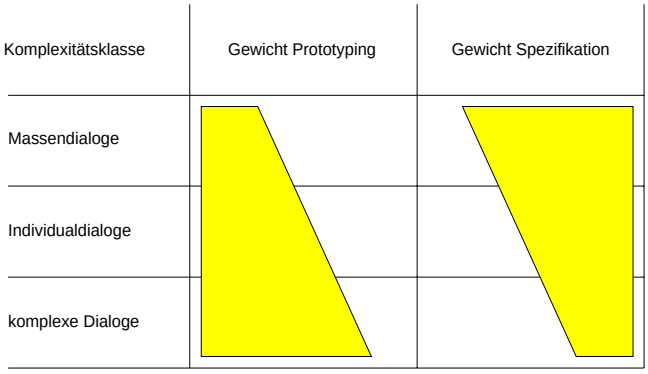

Bild 13: Prototyping oder Spezifikation

Bei *Massendialogen* ist es wichtig, je Dialogtyp einen Prototypen mit den Anwendern abzustimmen. Dieser Prototyp sollte nicht nur mit einem Rapid-Prototyping-Tool entwickelt werden. In Form eines Durchstichs bieten solche Prototypen zugleich die Möglichkeit, die Softwareentwicklungsumgebung einzurichten und zu erproben. Die Menge der Dialoge sollte dann aus einer Spezifikation generiert werden.

Bei *Individualdialogen* ist es meist zu aufwendig, einen voll funktionalen Durchstich zu realisieren. Der Prototyp sollte hier wie ein Architekturmodell die Ideen verdeutlichen. Die Spezifikation dient als Grundlage für die technische Realisierung und ist für qualitativ hochwertige Software unerläßlich.

Bei extrem *komplexen Dialogen* liegen uns wenig Erfahrungen vor. Die Tendenz wird aber stark zum Prototyping tendieren, da auch ein geschulter Softwareingenieur ab einer bestimmten Zahl von Zuständen und Ereignissen den Überblick verlieren wird. Hier hat die praktische Erprobung mit Prototypen Vorteile.

6. Zusammenfassung

Heute werden graphische Benutzeroberflächen im wesentlichen mit Hilfe von Prototypen entwickelt. Dies ist nicht in jedem Fall die optimale Lösung. Dieser Aufsatz hat gezeigt, daß man auch komplexere GUI-Anwendungen objekt- und dokumentenorientiert spezifizieren kann. Solche Spezifikationen haben jedoch eher den Charakter von technischen Zeichnungen, als den von Architekturmodellen.

Um dem Kunden qualitativ hochwertige Individualsoftware zur Verfügung zu stellen, kann derzeit weder auf Prototyping noch auf eine Spezifikation verzichtet werden. Die Gewichte hängen dabei von der Art der zu entwickelnden Software ab. Es bleibt eine Aufgabe der Softwaretechnik, die Transformierbarkeit von Spezifikationen in Prototypen und umgekehrt so zu verbessern, daß beides zu niedrigeren Kosten als heute möglich ist.

Literatur

[AndGre 92] Andleigh, P. K.; Gretzinger, M. R.: *Distributed Object Oriented Data-Systems Design*; Englewood Cliffs (Prentice Hall); 1992.

[BarDen 93] Bartsch, W.; Denert, E.: *Objektorientierte Spezifikation: Konzepte und eine Notation*; in Mayr, H. C.; Wagner, R. (Hrsg.): *Objektorientierte Methoden für Informationssysteme*; Proceedings, Fachtagung der GI-Fachgruppe EMISA, Klagenfurt, 7. - 9. Juni 1993 (Springer Verlag).

[Denert 91] Denert, E.: *Software-Engineering*; Berlin, Heidelberg, New York u.a. (Springer Verlag), 1991.

[Denert 93a] Denert, E.: *Dokumentenorientierte Software-Entwicklung*; Informatik Spektrum (1993) 16: S. 159 - 164.

[Denert 93b] Denert, E.: *Objektorientierte Spezifikation betrieblicher Informationssysteme*; Proceedings ONLINE 1993 Congress IV - C640.

[HopUll 79] Hopcroft, J. E.; Ullman, J. D.: *Introduction to Automata Theory, Languages and Computation*; Addison-Wesley 1979.

[HöTe 93] Höbel, N.; Tensi, T.: *Manual für die TOLEDO-Entwicklungsmethodik*; Interne Unterlagen, Siemens AG - CC Kordoba, 1993

[Keller 93] Keller, W.: *Spezifikation von Massendialogen mit TOLEDO*; Internes Diskussionspapier - über den Autor erhältlich; sd&mGmbH 1993.

[KraPop 88] Krasner, G. E.; Pope, S. T.: *A Cookbook for using the Model-View-Controller Interface Paradigm in Smalltalk-80*; Journal of Object Oriented Programming, 8/9-88, S. 26-49.

[Pfaff 83] Pfaff, G. E. (Hrsg.): *User Interface Management Systems*, Proceedings, Workshop on User Interface Management Systems, Seeheim, (1. - 3.11.1983); Springer Verlag 1983.

[Larson 92] Larson, J. A.: *Interactive Software*; Englewood Cliffs (Prentice Hall); 1992.

[Scholz 91] Scholz, G.: *Programmiersprachen und Makrotechnik* in: P.Schnupp (Hrsg.) *Moderne Programmiersprachen*; Oldenbourg 1991.

[Scholz 93] Scholz, G.: *Maßgeschneiderte Software-Generatoren*; Proceedings ONLINE 1993 Congress VI - C636.

Inhaltsverzeichnis

Dokumentenorientierte Spezifikation objektorientierter Benutzeroberflächen

Eine Alternative zum Prototyping

ONLINE ´94 - Congress VI - Innovative Softwaretechnologien: Neue Wege mit objektorientierten Methoden und Client/Server Architekturen

Wolfgang Keller

sd&m gmbh
München

Februar 1994

sd&m gmbh
Thomas-Dehler-Straße 18
81737 München
Telefon (089) 6 27 02 - 0
Telefax (089) 6 27 02 50

Abstract

In der Literatur zur objektorientierten Analyse liegt das Schwergewicht auf Methoden zur abstrakten Beschreibung von Anwendungsobjekten. Methoden zur objektorientierten Beschreibung von Benutzeroberflächen werden kaum angeboten.

Graphische Benutzeroberflächen werden heute meist in Form von Prototypen spezifiziert. Dieses Vorgehen bietet für kleinere Projekte mit einer geringen Anzahl von Dialogen den Vorteil schneller Abstimmung mit dem Anwender, führt jedoch bei großen Mengen von Dialogen zu schlechter Wartbarkeit der Spezifikationen, da vor allem strukturelle Änderungen an Dialogtypen hohen Aufwand verursachen.

Im HOST-Bereich haben sich für "dumme" Terminals formale Methoden zur Spezifikation von Dialogen (Seeheim-Modell, Interaktionsdiagramme) bewährt, die zu gut wartbaren Spezifikationen führen, aus denen teilweise fertige Dialoge generiert werden können.

Der Aufsatz stellt am Beispiel einer komplexeren GUI-Applikation eine dokumenten- und objektorientierte Methodik der Dialogspezifikation für graphische Benutzeroberflächen vor. Es wird gezeigt, daß sich GUI-Dialoge gut verständlich und relativ formal spezifizieren lassen, wobei Vorteile der Objektorientierung wie Klassenbildung und Vererbung zum Tragen kommen.

Zum Autor:

Wolfgang Keller studierte nach einer Siemens-Stammhauslehre Informatik mit Nebenfach Betriebswirtschaftslehre an der Technischen Universität München. Seine Schwerpunkte an der Hochschule waren Compilerbau und übersetzererzeugende Systeme. Seit 1991 arbeitet er bei der sd&m GmbH (München) als Softwareentwickler in verschiedenen Projekten. Sein Interesse gilt dabei vor allem Software-Produktionsumgebungen.