Visuelle Spezifikation des Kerns objektorientierter Applikationen

Gerd Szwillus
Universität - GH - Paderborn
Fachbereich Mathematik/Informatik
Warburger Straße 100
D-4790 Paderborn
szwillus@uni-paderborn.de

Für die Entwicklung interaktiver, objektorientierter Applikationen hat sich das klassische Seeheim-Modell [Pfaff 85] zur Strukturierung von Benutzungsschnittstellen nicht bewährt. An die Stelle einer zentralen Dialogkontrolle tritt heute typischerweise die Idee einer gemeinsamen Objektwelt von Applikation und Benutzungsschnittstelle *("shared application model")* [Dance *et al* 87]. Allerdings hat dies die Konsequenz, daß die Codeanteile von Applikation und Benutzungsschnittstelle wieder stark durchmischt bzw. auf viele Objekte verteilt werden. Dies läuft grundsätzlich der Forderung nach Trennen der Aufgaben *Anwendungsprogrammierung* und *Schnittstellenprogrammierung* entgegen. Das Ziel dieser Arbeit ist es, eine visuelle Notation mit einer zugehörigen Methodik zu entwickeln, die es erlaubt, Applikationen mit komplexen Benutzungsschnittstellen auf der Basis der Architektur "gemeinsamer Objekte" zu entwerfen. Durch eine klare Spezifikation der Schnittstelle zwischen Applikation und Benutzungsschnittstelle, des hier so genannten **Objektkerns**, wird es möglich, die Entwicklung beider Anteile streng voneinander zu trennen.

1. Einführung

Die visuelle Darstellung des Objektkerns abstrahiert von technischen Eigenschaften einer konkreten objektorientierten Programmiersprache und besitzt andererseits genug Ausdruckskraft, um die - für den Zweck erforderlichen - Beziehungen festzulegen. Grundsätzlich ist dieser Spezifikationsschritt Teil einer Phase der objektorientierten Analyse des Anwendungsproblems - wie etwa mit OBA [Rubin und Goldberg 92] - konzentriert sich allerdings auf eine Oberflächensicht auf die Applikation: Die Teilobjektwelt, die hier festgehalten wird, beschränkt sich auf die für die **Benutzung** des Systems relevanten Objekte. In dieser Phase der Entwicklung arbeiten der Entwickler der Applikation (AP-Entwickler) und der Entwickler der Benutzungsschnittstelle (BS-Entwickler) zusammen, um diese "Vertragsgrundlage" ihrer dann weitgehend separaten Tätigkeiten festzulegen. Beide können anschließend die Teilobjektwelt unabhängig voneinander nur noch erweitern, den gemeinsamen Kern aber nicht mehr ohne explizite Absprache verändern. Erweiterungen geschehen auf beiden Seiten durch das Hinzufügen von Klassen, Methoden und Attributen.
Zwischen den beiden Entwicklern besteht Einigkeit darüber, daß es genau die Klassen, Attribute und Methoden des Objektkerns sind, die bei der Benutzung sichtbar, benutzbar und ggf. veränderbar sein müssen. Ausschließlich diese Komponenten werden dem Benutzer angeboten - alle anderen Anteile der Anwendung arbeiten intern, für den Benutzer nur mittelbar durch ihre Auswirkungen sichtbar. Etwas genauer gesagt heißt das, daß nur Objekte dieser Klassen in der Benutzungsschnittstelle repräsentiert werden, daß nur die erwähnten Attribute die Darstellung beeinflussen können und daß nur die angeführten Methoden dem Benutzer zur Ausführung zur Verfügung stehen.

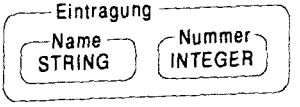
2. Die Objektkern-Notation

Objekte des objektorientierten Paradigmas werden als autonom handelnde Agenten aufgefaßt, die sich durch Identität, Gedächtnis und Verhalten [Booch 91] auszeichnen. Dieses Konzept legt eine Darstellung von Objekten bzw. Klassen als räumlich (optisch) zusammenhängende Strukturen nahe, wie bereits in ähnlichen derartigen Notationen ausgeführt - siehe etwa [Booch 91], [Wirfs-Brock, Wilkerson und Wiener 90], [Coad und Yourdon 91], [Felser 92]. Die Objektkern-Notation stellt Enthaltenseins-Beziehungen (wie etwa in Klassen definierte Methoden) durch Verwendung von graphischem "Ineinanderschachteln" und Verweis-Beziehungen (wie etwa die Vererbungsbeziehung) durch Linien bzw. Übereinstimmung von Namen dar. Damit rückt die Objektkern-Notation optisch und konzeptionell in die Nähe von Harel's *Statecharts* [Harel 88].

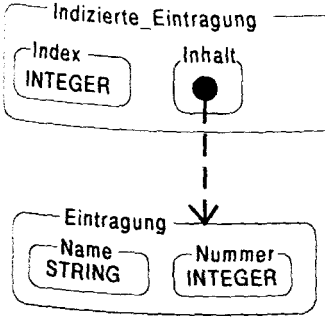
2.1 Klassen

Wesentliches Element des Objektkerns ist die Definition von Klassen. Die Darstellungen von Klassen bzw. Objekten sind flächig, damit sie die lokalen Komponenten (Attribute und Methoden) optisch umschließen können; wir verwenden Rechtecke mit abgerundeten Ecken. In der oberen Seite des Rechtecks integriert steht der Name der dargestellten Klasse. So repräsentiert das unten rechts gezeigte Rechteck einen Behälter für die Klassenstruktur der Klasse `Eintragung`. In dem Behälter kann man weitere Behälter unterbringen, welche die **Attribute** (siehe Abschnitt 2.2) der Klasse bezeichnen; außerdem werden die **Methoden** als in den Klassen enthalten dokumentiert (siehe Abschnitt 2.4). Wesentlich bei dieser graphischen Modellierung ist die optisch deutliche Unterscheidung zwischen als Teilen enthaltenen Datentypen und Datentypen, auf die nur verwiesen wird.

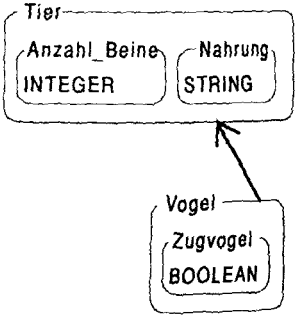
2.2 Attribute



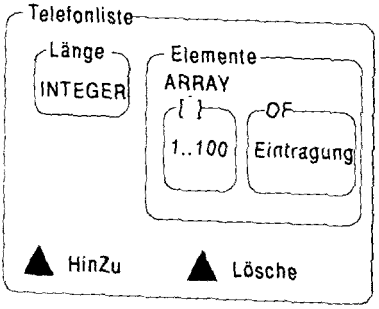
In der Abbildung rechts erkennt man, daß `Eintragung` zwei Attribute enthält: einen `Namen` vom Typ `STRING` und eine `Nummer` vom Typ `INTEGER`. Die Bezeichner von Typen, die in den Kästen stehen, sind entweder die Namen von Basistypen wie `STRING`, `INTEGER`, `REAL`, `BOOLEAN` oder aber die Namen anderer, selbstdefinierter Klassen.
Als speziellen Basistyp gibt es auch den **Pointer** auf andere Datentypen. Dieser wird durch einen gestrichelten Pfeil mit "dickem" Startpunkt dargestellt. In der Abbildung links wird der Typ `Indizierte_Eintragung` gezeigt, in dem ein Verweis auf `Eintragung` enthalten ist - `Eintragung` ist nicht Teil der Objekte vom Typ `Indizierte_Eintragung`. Außerdem können strukturierte Datentypen eingetragen werden, wie etwa Felder oder Mengen; Verbünde sind durch entsprechende Schachtelung der Behälter ohnehin ausdrückbar.



2.3 Vererbung



Vererbungsbeziehungen zwischen Klassen drückt man durch einen "erbt-von"-Pfeil aus, der von der erbenden zur vererbenden Klasse führt. Durch diese Richtung zeigt der Pfeil die Richtung an, in der man von einer angesprochenen Klasse aus ein ererbtes Attribut bzw. eine ererbte Methode suchen muß. Um den Mehrdeutigkeiten multipler Vererbung aus dem Weg zu gehen, sei hier nur einfache Vererbung erlaubt; das heißt, von jeder Klasse darf höchstens ein Vererbungspfeil ausgehen. Die Abbildung rechts zeigt die Vererbungsbeziehung zwischen den Klassen `Tier` und `Vogel`. Ein Objekt vom Typ `Vogel` hat das Attribut `Zugvogel`, aber auch `Anzahl_Beine` und `Nahrung`. Zur Vermeidung der Überfrachtung von Bildern werden die ererbten Attribute (und später auch Methoden) nicht wiederholt. Nur wenn Methoden oder Attribute einer vererbenden Klasse auf tieferer Ebene überschrieben werden, wiederholt man die Symbole in den spezielleren Klassen.



2.4 Methoden

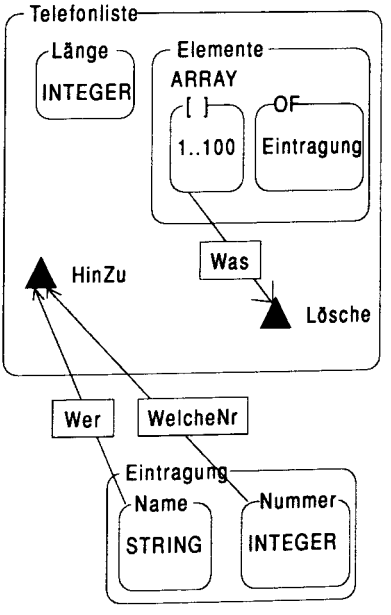
Die Methoden werden - wie die Attribute - als graphische Objekte innerhalb des die Klasse darstellenden Rechtecks gezeichnet. Wir verwenden - in Anklang an den Buchstaben "A" - ein Dreieck als Symbol für Methoden (Aktionen). Die Abbildung links zeigt als Beispiel die Methoden `HinZu` und `Lösche` der Klasse `Telefonliste`. Die Anbindung der Parameterbeschreibung folgt der Idee, daß die Parameter einer Methode in praktisch allen Fällen als Werte für Attribute von Objekten verwendet werden. Dies ist zunächst eine rein empirische Beobachtung, die im Einzelfall auch unzutreffend sein kann, erweist sich aber als grundlegend vernünftige Technik.

Wir definieren einen Parameter für eine Methode durch Verbindung des Methodensymbols (des Dreiecks) mit einem Typkonstrukt eines Attributs eines Objektes. Dies bewirkt einerseits die Deklaration des Parametertyps - nämlich mit dem Typ des angebundenen Attributs. Andererseits dokumentiert die Verbindung die vorgesehene **Verwendung** des Parameters. Die Richtung der Pfeile zeigt den Datenfluß in die Methode oder aus der Methode heraus an und unterscheidet auf diese Weise Eingabe- und Ausgabeparameter. Falls gewünscht, kann man gestrichelte und durchgezogene Pfeile unterscheiden, um - in Analogie zu den gestrichelten Pfeilen bei Zeigern - zwischen der Parameterübergabe über Referenzen (gestrichelt) und durch Wertkopie (durchgezogen) zu unterscheiden.

Die Abbildung auf der nächsten Seite zeigt die Spezifikation der Methoden `HinZu` und `Lösche` mit ihren Eingabeparametern. Die Pfeilverbindungen treffen folgende Aussagen:

- Der Parameter `Was` von `Lösche` ist ein `INTEGER`-Wert im Unterbereich `1..100`. Da `Lösche` eine Eintragung der Telefonliste löschen soll, ist durch die Verbindung mit dem Index klar gestellt, daß der Wert `Was` den Index des zu löschenden Satz spezifiziert.

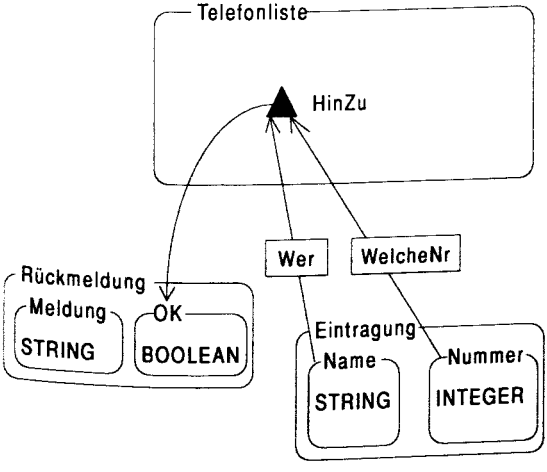
- Die Methode `HinZu` soll ein neues Paar `(Name, Nummer)`, also eine neue `Eintragung` erzeugen. Somit sind die übergebenen Parameter `Wer` und `WelcheNr` die entsprechenden Attributwerte der neuen `Eintragung`.



Diese Art der Spezifikation von Parametern kann man auch durch den Zweck der Spezifikation des Objektkerns rechtfertigen, der die für die Benutzung relevanten Objekte und Attribute festlegen soll: In der fertigen Applikation muß der Benutzer die Parameter angeben; daher müssen sie im Objektkern entweder direkt als Daten auftreten, oder aber es handelt sich um reine Steuerinformationen über den vom Benutzer zu steuernden Prozeß. Wie man im zweiten Fall verfahren kann, zeigt die folgende Erweiterung unseres Beispiels.

Wenn wir zum Beispiel die Methode `HinZu` dahingehend erweitern wollen, daß sie einen `BOOLEAN`-Wert zurückgibt, der das korrekte Ausführen widerspiegelt (z.B. **`true`** genau dann, wenn die Methode korrekt ausgeführt wurde), so stehen wir vor dem Problem, daß sich kein Attribut kanonisch anbietet, um diese Information aufzunehmen. Die Lösung dafür ist, ein zunächst künstlich erscheinendes Objekt mit geeigneten Attributen einzuführen.

Reflektiert man den Zweck der Spezifikation des Objektkerns als Festlegung der für den Benutzer relevanten Komponenten eines Systems, so ist die eingeführte Objektart `Rückmeldung` keineswegs so künstlich, wie es zuerst schien. Die Festlegung, daß eine Benutzerfunktion eine Ausgabe erzeugt, macht nur dann Sinn, wenn diese intern erzeugte Information dem Benutzer auch in irgendeiner Form vermittelt wird. Somit repräsentieren Objektarten wie die gezeigte `Rückmeldung` Objekte der Benutzungsschnittstelle, deren sich auch die Anwendung "bewußt" sein muß, da sie sie für Mitteilungen an den Benutzer benötigt.



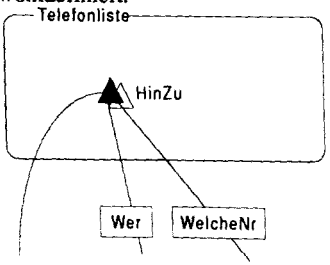
3. Methodischer Einsatz der Objektkern-Notation

Die Notation, wie im vorigen Kapitel eingeführt, bildet den Kern einer Entwicklungsmethodik von Benutzungsschnittstellen objektorientierter Applikationen. Dabei liegt die Betonung für den Einsatz dieser Notation bei der Definition der

Schnittstelle zwischen Anwendung und Benutzungsschnittstelle unter dem Konzept der "gemeinsamen Objekte". Die Ermittlung der Objektkern-Struktur ist, wie die Ermittlung der Objektstruktur einer Anwendung überhaupt, ein im allgemeinen nicht-triviales Problem, dessen Behandlung unter dem Begriff *objektorientierte Analyse* zusammengefaßt wird. Unter der Überschrift "Methodischer Einsatz" soll es aber hier nicht um eine analytische Methode gehen, sondern darum, inwieweit eine Objektkern-Spezifikation als technisches Dokument Ausgangspunkt der anschließenden Software-Entwicklung sein kann.

3.1 Entwicklung der Applikation: Der Benutzer als Objekt

Ziel dieser Methodik in Bezug auf die Anwendungsentwicklung ist eine hochgradige, wenn nicht völlige Entlastung dieses Prozesses von Betrachtung von Aspekten der Benutzungsschnittstelle. Das objektorientierte Paradigma bietet hier einen naheliegenden Ansatz: Für die Anwendung ist es praktisch, den Benutzer als Objekt im Software-Sinne zu betrachten[1]. Unter diesem Denkansatz ist der Benutzer aus Sicht der Anwendung ein Objekt mit Attributen und Methoden. Daher ist dieses Objekt in der Lage, genau wie "echte" Software-Objekte, Methoden aufzurufen und Methoden zur Verfügung zu stellen. Die Art der Einbindung von Methoden und ihre Versorgung mit Parametern in der Objektkern-Notation erlaubt schon sehr früh und auf abstraktem Niveau eine Kontrolle darüber, welche Informationen der Benutzer später sehen wird und muß, um die ihm angebotenen Methoden korrekt einsetzen zu können.
Dadurch ist nach Festlegen des Objektkerns die Vorgehensweise für den Anwendungsentwickler in Bezug auf die Benutzungsschnittstelle sehr klar vorgezeichnet: Seine Aufgabe besteht in der korrekten Erstellung und Verwaltung der spezifizierten Objektwelt mit den angegebenen Attributen und Methoden. Aspekte der optischen Darstellung der Objektwelt, Interaktionstechniken zum Aufruf der Methoden und genauere Festlegung der Ausgaben an den Benutzer sind nicht seine Aufgabe. Gleichzeitig ist der Aufsatzpunkt für den Entwickler der Schnittstelle, der dem Benutzer all diese "Dienste" anbieten muß, wohldefiniert.



Da der Benutzer keine Software-Komponente darstellt, die der Entwickler der Anwendung kontrolliert, muß es ihm möglich sein, den Aufruf von Methoden durch den Benutzer einzuschränken. Der Entwickler weiß - aufgrund der Struktur der Anwendung - daß es nicht für jede Methode zu jeder Zeit Sinn macht, dem Benutzer zur Verfügung zu stehen. Ähnliche Abhängigkeiten existieren für alle Methoden in dem objektorientierten System, werden aber - durch "korrekte" Verwendung von Methoden durch andere Systemteile - implizit beachtet. Es ist daher sinnvoll, den dem Benutzer zur Verfügung stehenden Methoden, von seiten der Anwendung eine APPLICABLE-Funktion zuzuordnen: Diese liefert einen Booleschen Wert **`true`** genau dann, wenn die Methode anwendbar ist und **`false`** sonst. Die Entscheidung über die Anwendbarkeit sollte ein Objekt selbst steuern, daher benötigt eine APPLICABLE-Funktion keine Eingabeparameter. Da auch das Format und die Verwendung der Ausgabe implizit vordefiniert sind, ist auch keine optische Verbindung dafür im eigentlichen Objektkern erforderlich. Optisch kennzeichnen wir das Vorhandensein einer APPLICABLE-

---

[1]) Interessanterweise erscheint der Benutzer im Kontext von objektorientierter Analyse durchaus noch als "Objekt"; im endgültigen Software-System jedoch tritt an die Stelle des Benutzer-Objektes eine Sammlung von den Benutzer bedienenden Objekten - die Benutzungsschnittstelle.

Funktion durch Unterlegen des "Methoden-Dreiecks" mit einem weißen "Schatten", wie oben links gezeigt. Nicht jede Methode muß eine zugeordnete APPLICABLE-Funktion besitzen.

3.2 Entwicklung der Benutzungsschnittstelle

Prinzipiell geht der BS-Entwickler genauso vor, wie der Entwickler der eigentlichen Anwendung: Er erweitert den Objektkern um weitere Klassen, Attribute und Methoden. Ziel dieser Ergänzungen ist die Gestaltung der Benutzungsschnittstelle mit den Aspekten *graphische Darstellung* der Objektwelt und *Erkennen und Bearbeiten von Eingaben* des Benutzers. Dieser Gestaltungsprozeß ist durch eine Dualität der Betrachtungsweise gekennzeichnet: Auf der einen Seite müssen die bildliche Ausgabe und die Interaktionstechniken in ihrer Erscheinungsform "nach außen", also wie für den Benutzer erkennbar, beschrieben werden; andererseits müssen die Ableitung der graphischen Darstellung aus den Objekten bzw. die Wirkung der Interaktion "nach innen", auf die internen Objekte, definiert werden. In der Folge sprechen wie jeweils von der externen und der internen Sicht auf die Darstellung bzw. die Interaktionen. Technisch geht der BS-Entwickler so vor, daß er die graphische Ausgabe durch Hinzufügen graphischer Attribute und die Interaktion durch Hinzufügen von Interaktionsmethoden definiert.

4. Das Werkzeug

Eine experimentelle Version eines die Objektkern-Notation unterstützenden Werkzeugs wird zur Zeit implementiert. Dabei steht zunächst die reine Editorfunktion im Vordergrund. Für die beiden folgenden getrennten Entwicklungen von Anwendung und Benutzungsschnittstelle sind eine Reihe sehr mächtiger Anschlüsse denkbar. Zentral ist für beide Bereiche, daß das Werkzeug Mechanismen anbieten muß, um Teile des Diagramms "einzufrieren", um die Konsistenz des Objektkerns zu garantieren. Endziel wäre eine netzwerkgestützte Mehrbenutzer-Version eines solchen Editors .

5. Literatur

Booch, G
Object Oriented Design with Applications, Benjamin/Cummings, Redwood City, CA, 1991

Coad, P; Yourdon, E
Object-Oriented Analysi,Prentice-Hall, Englewood Cliffs, 1991

Dance, J R; Granor, T E; Hill, R D; Hudson, S E; Meads, J; Myers, B A; Schulert, A
The Run-time Structure of UIMS-Supported Applications, Computer Graphics 21, 2, 1987

Felser, W
Kreativer objektorientierter Entwurf: Methodik, Diplomarbeit, Universität - GH - Paderborn, Fachbereich Mathematik/Informatik, August 1992

Harel, D
On Visual Formalisms,CACM, Vol. 31, No. 5, 1988.

Pfaff, G E
User Interface Management Systems: Proceedings of the Seeheim Workshop, Springer-Verlag, Berlin, 1985

Rubin, K S; Goldberg, A
Object Behavior Analysis, CACM Vol. 35, No. 9, S.48-62, 1992

Szekely, P; Myers, B A
A User Interface Toolkit Based on Graphical Objects and Constraints, OOPSLA'88, SIGPLAN Notices, November 1988

Wirfs-Brock, R; Wilkerson, B; Wiener, L
Designing Object-Oriented Software, Prentice-Hall, Englewood Cliffs, 1990