Objektorientierter Entwurf

Udo Kelter

04.10.2003

Zusammenfassung dieses Lehrmoduls

Dieses Lehrmodul stellt grundlegende Konzepte des objektorientierten Entwurfs vor. Wir gehen insb. auf die Darstellung von Entwurfsklassendiagrammen in der UML und auf den Übergang von der Analyse zum Entwurf und vom Entwurf zur Programmierung ein.

Vorausgesetzte Lehrmodule:

obligatorisch: – Objektorientierte Modellierung

Stoffumfang in Vorlesungsdoppelstunden: 0.5

Inhaltsverzeichnis

1 Einleitung

Dieses Lehrmodul gibt eine erste Einführung in den objektorientierten Entwurf (*object-oriented design; OOD*), genauer gesagt die Darstellung von Programm-Architekturen durch Entwurfs-Klassendiagramme. Es wird vorausgesetzt, daß die Konzepte und Notationen der objektorientierten Analyse schon bekannt sind.

Grundlegende Konzepte wie Klasse, Attribut, Operation, Paket u.a. sind bei der objektorientierten Analyse und beim objektorientierten Entwurf gleich, deshalb brauchen wir sie hier nicht erneut vorzustellen, sondern setzen entsprechende Kenntnisse voraus. Wir konzentrieren uns hier vor allem auf die Unterschiede. Generell stellen die Entwurfs-Klassendiagramme mehr Details dar als die Analyse-Klassendiagramme.

Wir werden die Konzepte und Notationen verwenden, die die UML [UML99] zur Spezifikation von Architekturen anbietet.

2 Klassen

Bei objektorientierten Sprachen sind Klassen die "Bausteine", aus denen Programme zusammengesetzt sind. Daher werden im Entwurf vor allem diese Bausteine repräsentiert, nicht unzufällig ebenfalls als **Klasse** bezeichnet.

Eine Entwurfsklasse repräsentiert normalerweise genau eine Programmklasse. Die Umkehrung gilt nicht. Es kann Programmklassen geben, die für das Entwerfen weniger wichtig sind und die, wenn man sie darstellen würde, von den wichtigen Dingen ablenken würden (vgl. die Diskussion über Bibliotheken und Frameworks in Lehrmodul [SAR]).

Graphisch dargestellt werden Entwurfsklassen in der UML im Prinzip genauso wie Analyseklassen; die enthaltenen Attribute und Operationen werden allerdings detailreicher dargestellt (s.u.).

Container-Klassen. Ein wichtiger semantischer Unterschied zwischen Analyse- und Entwurfsklassen besteht darin, daß Entwurfsklas-

sen keine implizite Objektverwaltung haben. Zur Erinnerung: bei Analyseklassen unterscheidet man nicht zwischen dem Typ und der Menge der Instanzen des Typs. Mit anderen Worten wird implizit eine Verwaltung der Instanzen des Typs unterstellt. Bei Entwurfsklassen ist dies nicht der Fall, es muß explizit eine Klasse vorgesehen werden, die die Instanzen des Basistyps verwaltet. Derartige Klassen nennt man Container-Klassen. Die Umsetzung von Analyseklassen in Entwurfsklassen wird in [TAE] ausführlich behandelt.

3 Attribute

Eine Attributspezifikation in der UML hat folgende Form[1]:

`Sichtbarkeit Attributname: Typ = Anfangswert { Merkmale }`

Da wir hier unterstellen, daß programmiersprachenabhängige Entwürfe erstellt werden, sind bei der Angabe von Attributnamen und Attributtyp die Gegebenheiten der Programmiersprache zu beachten. Dementsprechend gibt die UML nur Empfehlungen zur Wahl der Attributnamen (erster Buchstabe kleingeschrieben) und läßt es völlig offen, wie Attributtypen spezifiziert werden.

3.1 Sichtbarkeit

Bei Analysemodellen waren Attribute nur innerhalb der Klasse und in Subklassen sichtbar. In Entwürfen stehen die aus C++ und Java bekannten Sichtbarkeitsfestlegungen zur Verfügung:

public	für alle Klassen sichtbar
protected	für diese Klasse und ihre Unterklassen sichtbar
private	für diese Klasse, aber nicht für ihre Unterklassen sichtbar

In den Attributlisten wird die Sichtbarkeit eines Attributs durch die Kürzel `+`, `#` bzw. `-` notiert; Bild 1 zeigt Beispiele.

[1] Diese Form ist leicht vereinfacht. Die komplette Angabe umfaßt zusätzlich eine Kardinalität des Attributs, auf die wir hier nicht eingehen.

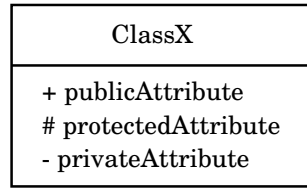

Abbildung 1: Sichtbarkeit von Attributen

public-Attribute sind aus softwaretechnischer Sicht bedenklich, da hier sozusagen Typen offen exportiert werden; dies widerspricht ganz eklatant dem Geheimnisprinzip, demzufolge das Wissen darüber, wie Datenstrukturen aufgebaut sind, an einer Stelle konzentriert wird und allen anderen Systemteilen somit verborgen bleibt, damit keine Abhängigkeiten entstehen. Von public-Attributen ist daher normalerweise abzuraten.

Für protected-Attribute gelten tendenziell die gleichen Bedenken. Derartige Attribute sind zwar in weniger Klassen sichtbar als public-Attribute, nichtsdestotrotz kann so das Wissen über die interne Struktur einer Klasse in beliebige Systemteile verschleppt werden, denn die Bildung von Unterklassen ist - neben dem Aufruf von Operationen - in objektorientierten Sprachen eine ganz normale Benutzung einer Klasse.

Letztlich sollten Attribute also möglichst als `private` angegeben werden. Wenn man dennoch von außen auf diese Attribute zugreifen möchte, sollte man entsprechende Operationen `liesX` bzw. `setzeX` anbieten.

Am Ende einer Attributspezifikation kann in gescheiften Klammern noch das Merkmal `frozen` angegeben werden. Ein eingefrorenes Attribut kann nach der Initialisierung nicht mehr verändert werden.

Aus Platzgründen sollte man in Entwurfsdiagrammen nur die Sichtbarkeit und den Namen der Attribute eintragen. Alle weiteren Angaben sollten in geeigneteren Darstellungen gemacht werden, wie sie typischerweise von Werkzeugen angeboten werden.

3.2 Abgeleitete Attribute

Abgeleitete Attribute sind auch im Entwurf möglich und werden wie in Analysemodellen durch einen vorgestellten `/` gekennzeichnet. Durch weitere UML-Sprachelemente, auf die wir hier aus Platzgründen nicht eingehen, kann dargestellt werden, von welchen anderen Größen das abgeleitete Attribut abhängt.

Abgeleitete Attribute, die im Analysemodell vorgegeben werden, können auf zwei Arten in den Entwurf umgesetzt werden:

1. durch eine Operation, die den Wert des Attributs berechnet (sozusagen als Ersatz für eine Operation `liesX`)
2. durch ein Attribut, dessen Wert allerdings immer konsistent gehalten werden muß mit den Größen, von denen der Wert abhängt. Ändert sich eine dieser Größen, muß der Wert des Attributs entweder komplett neu berechnet werden (was Fall 1 entspräche; in diesem Fall wäre das Attribut praktisch nur ein Puffer) oder inkrementell korrigiert werden.

Welche Alternative gewählt wird, ist vor allem eine Frage der Performance-Optimierung. Bei einer aufwendigen Berechnungsfunktion, häufigem Lesen des Attributs und seltenen Änderungen der "Original"-Größen liegt bspw. die zweite Alternative nahe.

3.3 Klassenattribute

Klassenattribute werden auch in Entwurfsdiagrammen durch Unterstreichung gekennzeichnet.

Klassenattribute können ebenfalls aus anderen Größen abgeleitet werden, d.h. prinzipiell hat man bei der Umsetzung von Klassenattributen, die im Analysemodell vorgegeben sind, die gleichen Alternativen wie bei abgeleiteten Attributen zur Auswahl. Bei der ersten Alternative, den Wert jedesmal zu berechnen, kann allerdings der Aufwand leicht zu hoch werden, denn definitionsgemäß muß ja über alle Instanzen der Klasse iteriert werden.

Bei der zweiten Alternative können die Klassenattribute naheliegenderweise in der Container-Klasse, die i.d.R. zu einer Analyseklasse

gebildet wird (vgl. [TAE]), angeordnet werden. Ggf. kann auch eine eigene Klasse definiert werden, die nur dieses Attribut hat und von der nur eine Instanz existiert.

Alternativ kann - sofern die Programmiersprache dies unterstützt (in Java bspw. `static`-Attribute) - auch ein Klassenattribut verwendet werden.

4 Assoziationen

Entwurfsassoziationen sind im Gegensatz zu Analyseassoziationen *gerichtet*; die Richtung wird durch einen Pfeil angezeigt (s. Bild 2).

Abbildung 2: Beispiel für eine Entwurfsassoziation

Eine Assoziation zwischen zwei Entwurfsklassen drückt im einfachsten Fall aus, daß die Klasse, von der der Pfeil ausgeht, Referenzen auf Objekte der zweiten Klasse (also letztlich entsprechende Zeiger) enthält. Insofern sind Assoziationen vergleichbar mit Attributen, und konsequenterweise kann man die Sichtbarkeit (`+`, `#` bzw. `-`) wie bei Attributen festlegen. Angeordnet wird diese Angabe als Präfix des Rollennamens.

Der Pfeil drückt die Richtung aus, in der zwischen Instanzen der beiden Klassen navigiert werden kann. Je nach der Anwendung kann es notwendig sein, in beiden Richtungen navigieren zu können. In diesem Fall müssen in beiden Richtungen Pfeilspitzen angegeben werden[2]. Bei solchen bidirektionalen Assoziationen muß bei der Implementierung darauf geachtet werden, daß die beiden gegenläufigen Referenzen immer nur zusammen erzeugt und gelöscht werden.

[2]Für diesen Fall kann man die Konvention vereinbaren, daß dann gar keine Pfeilspitzen angegeben werden. In diesem Text werden wir allerdings immer alle Pfeilspitzen angeben.

Entwurfsassoziationen können ebenso wie Analyse-Assoziationen **Kardinalitäten** haben und **attributiert** sein. Die Kardinalitäten werden wie bei Analyse-Assoziationen angegeben. Attribute werden auch hier bei einer **assoziativen Klasse** angeordnet, die wie in Analysemodellen durch eine gestrichelte Linie mit der Assoziationslinie verbunden wird (vgl. Bild 5 in Lehrmodul [OOA]).

4.1 Umsetzung von Entwurfsassoziationen in Programme

Entwurfsassoziationen können in Programmen am einfachsten mit Hilfe von Zeigern zwischen Objekten realisiert werden. Für jede Richtung ist ein eigener Zeiger erforderlich. Die Programmklasse, von der die Assoziation ausgeht, wird um entsprechende Zeigervariablen erweitert. Die Details hängen von der Kardinalität der Rolle bzw. Richtung ab:

Kardinalität 0:1 oder 1: Die Assoziation kann hier durch einen Zeiger in der Klasse realisiert werden. Bei der Kardinalität 1 muß dieser Zeiger schon beim Anlegen eines Objekts dieser Klasse initialisiert werden, d.h. die Konstruktoroperationen müssen ggf. einen Parameter haben, der das Zielobjekt angibt.

Sofern die Entwurfsassoziation attributiert ist, also eine zugehörige assoziative Klasse vorhanden ist, können die Attribute der assoziativen Klasse direkt in der Programmklasse "neben" der Zeigervariablen realisiert werden.

andere Kardinalitäten: Hier muß eine Menge von Zeigern verwaltet werden. Ist eine Maximalzahl der Zeiger bekannt (Kardinalität 0:n), kann ein Array von Zeigern verwendet werden, andernfalls muß eine dynamische Datenstruktur eingesetzt werden.

Ist die Entwurfsassoziation attributiert, muß statt eines Zeigers ein Objekt verwendet werden, das den Zeiger und die Attribute enthält.

Die vorstehenden Realisierungsmöglichkeiten gelten auch für Aggregationen und Kompositionen. Damit sich das "Ganze" um seine

"Teile" kümmern kann, muß hier mindestens in diese Richtung navigiert werden können.

Bei Kompositionen kann, da die Komponenten exklusiv im Ganzen enthalten sind, eine alternative Realisierungsform gewählt werden (vor allem bei Kardinalität 0:1 oder einer bekannten Maximalzahl der Komponenten): Die Komponenten werden direkt in das Ganze eingebettet. Dann werden die Komponenten immer automatisch mit dem Ganzen angelegt bzw. gelöscht.

4.2 Umsetzung von Analyse-Assoziationen in Entwurfsassoziationen

Wir betrachten hier nur binäre Assoziationen.

Für die Umsetzung einer Analyse-Assoziation in eine Entwurfsassoziation stehen prinzipiell zwei Alternativen offen:

1. Der einfachste Fall ist eine 1:1-Umsetzung, d.h. eine Analyse-Assoziation wird umgesetzt in genau eine Entwurfsassoziation.

   Ob bei letzterer nur eine der Navigationsrichtungen vorgesehen wird oder beide, muß abhängig davon entschieden werden, wie durch die Anwendung (insb. die Implementierungen der Operationen der beiden involvierten Klassen) auf über die Beziehungen navigiert wird.

   Sofern die Analyseassoziation attributiert ist, also eine zugehörige assoziative Analyseklasse vorhanden ist, und beide Navigationsrichtungen vorhanden sind, muß abhängig vom Zugriffsverhalten entschieden werden, welcher der beiden Navigationsrichtungen die Attribute zugeordnet werden. Die Attribute können auch auf die beiden Richtungen aufgeteilt werden.

   Sofern auf bestimmte Attribute von beiden Navigationsrichtungen aus zugegriffen werden muß, benötigt man zusätzlich eine Operation, die zu einer Beziehung die zugehörige Umkehrbeziehung liefert.

2. Einsatz von Assoziationsobjekten: eine Analyse-Assoziation wird umgesetzt in eine Klasse, von der i.w. zwei Assoziationen ausgehen, zu den Entwurfsklassen K1 und K2 führen, wobei K1 und

K2 die Entwurfsklassen sind, die den umgesetzten Analyseklassen entsprechen[3]. Bild 3 zeigt ein Beispiel. Für die Assoziationsobjekte muß i.d.R. zusätzlich ein Containerobjekt bzw. eine entsprechende Klasse vorhanden sein.

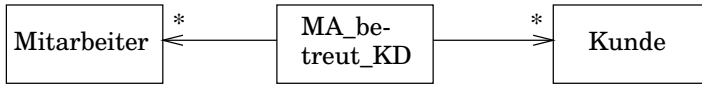

Abbildung 3: Einsatz eines Assoziationsobjekts

Für jede zu unterstützende Navigationsrichtung, z.B. von K1 nach K2, enthält diese Containerklasse typischerweise eine Operation, die zu einem Objekt des Typs K1 alle Assoziationsobjekte des Typs K2 liefert, die mit dem K1-Objekt über ein Assoziationsobjekt verbunden sind.

Sofern die Analyseassoziation attributiert ist, können die Attribute direkt in den Assoziationsobjekten realisiert werden.

Assoziationsobjekte sind weniger effizient als Entwurfsassoziationen, haben aber den Vorteil, daß die involvierten Klassen nicht verändert werden müssen. Bei attributierten Assoziationen haben sie den Vorteil, die Attribute nicht u.U. willkürlich einer der beiden Navigationsrichtungen zuordnen zu müssen.

5 Operationen

Zur jeder Operation sind folgende Angaben zu machen:

1. die **Signatur**: diese besteht aus dem Namen der Operation, der Folge der Parametertypen und dem Typ des Rückgabewerts.

   Zu jedem Parameter ist zusätzlich anzugeben:
   - die Übergabeart `in`, `out` bzw. `inout`

[3]Diese entsprechen der Verbindungstabelle, die man benutzen muß, wenn man m:n-Beziehungstypen in ER-Diagrammen in tabellenartige Strukturen umsetzt (vgl. Abschnitt 2.2 in [TAE]).

- der Name
- ein Vorgabewert

In den Klassendiagrammen wird auf diese Angaben normalerweise verzichtet.

2. eine Beschreibung der Wirkung der Operation; hierbei wird die Sprache bzw. konzeptuelle Basis offengelassen. Es können u.a. Vor- und Nachbedingungen und freier Text verwendet werden.
3. eine Sichtbarkeitsangabe wie bei Attributen
4. eine Kennzeichnung, ob die Operation abstrakt ist

Es kann in einer Klasse mehrere Operationen mit gleichen Namen geben. In diesem Fall ist der Operationsname **überladen**. Die Operationen mit dem gleichen Namen müssen sich aber in der Parameterliste, also der Sequenz der Parametertypen, unterscheiden.

Abstrakte Operationen. Der Zweck abstrakter Operationen besteht darin, für gleichlautende Operationen mehrerer Unterklassen eine gemeinsame Schnittstelle zu realisieren. Bei abstrakten Operationen wird analog wie bei abstrakten Klassen entweder der Name kursiv geschrieben oder das Merkmal `abstract` angegeben.

Abstrakte Operationen haben keine Implementierung in der Klasse, in der sie stehen, erst in den Unterklassen werden jeweils passende Implementierungen geliefert. Eine Klasse mit einer abstrakten Operation muß daher ebenfalls abstrakt sein, sie kann nicht instantiiert werden. Bildet man eine Unterklasse einer solchen abstrakten Klasse, so bleiben die geerbten abstrakten Operationen in der Unterklasse natürlich ebenfalls abstrakt, es sei denn, sie werden dort redefiniert, d.h. die Unterklasse enthält eine nichtabstrakte Operation mit gleicher Signatur.

6 Schnittstellen

Eine **Schnittstelle** (*interface*) ist eine Klasse, die nur abstrakte Operationen enthält, sonst nichts; sie enthält keine Attribute und keine

ausgehenden Assoziationen, kann aber das Ziel von Assoziationen sein, die von anderen Klassen ausgehen.

Eine Schnittstelle kann nicht instantiiert werden, sie kann aber in Typhierarchien enthalten sein, und es gibt normalerweise instantiierbare Subtypen. Dargestellt wird eine Schnittstelle in einem Klassendiagram entweder

- wie eine Klasse, aber mit dem Stereotyp «`interface`» , der Abschnitt für die Attribute entfällt.
- oder durch einen kleinen Kreis, neben dem der Name der Schnittstelle steht; die Operationen der Schnittstelle sind aus dieser Darstellung nicht erkennbar.

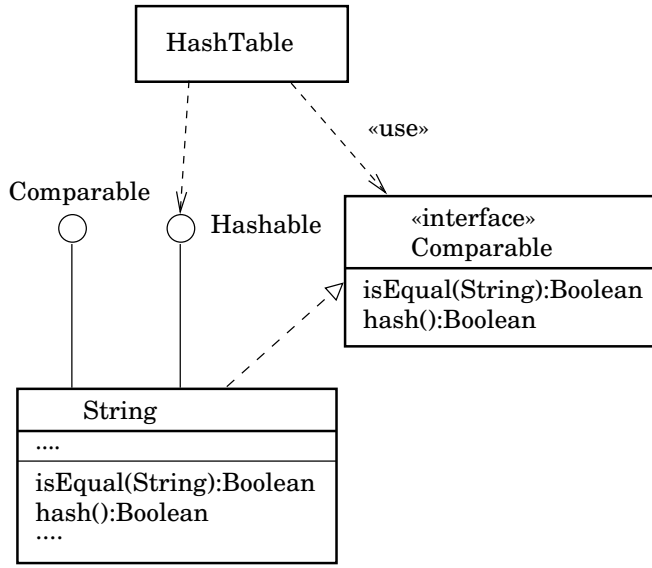

Abbildung 4: Schnittstellen (Interfaces)

Bild 4 zeigt als Beispiel die Schnittstelle `Comparable` in beiden Varianten. Man könnte sagen, eine Schnittstelle sei nur eine spezielle abstrakte Klasse, deswegen sei ein eigenes Konzept überflüssig.

Tatsächlich werden Schnittstellen intensiv dazu benutzt, sozusagen "Sichten" auf die Dienste einer Klasse zu definieren, also die Gesamtmenge der Dienste einer Klasse zu *filtern*. Wenn `K` eine derartige Klasse ist und `S` die Schnittstelle, könnte man `K` als normalen Subtyp von `S` betrachten. Hiergegen sprechen allerdings zwei Argumente:

1. Daß es sich hier um einen vom Normalfall stark abweichenden Sonderfall mit einer eigenen Bedeutung handelt, wäre optisch nicht erkennbar.
2. Häufig braucht man mehrere "Sichten" auf eine Klasse. Man müßte eine solche Klasse als Subtyp mehrerer Schnittstellen-Klassen definieren. Manche objektorientierte Sprachen (insb. Java) unterstützen aber kein mehrfaches Erben, sondern haben stattdessen ein Schnittstellenkonzept.

Daher werden in der UML Schnittstellen und die Beziehungen zwischen Schnittstellen und anderen Klassen speziell dargestellt.

In Bild 4 "realisiert" die Klasse `String` die Schnittstelle `Comparable`, sie ist also als Unterklasse von `Comparable` anzusehen. Dargestellt wird dies durch eine gestrichelte Vererbungsbeziehung.

Die Klasse `HashTable` **benutzt** die Schnittstellen `Comparable` und `Hashable`. Dargestellt wird dies durch einen gestrichelt gezeichneten Assoziationspfeil, an dem, wenn das Ziel als Klasse (und nicht als Kreis) dargestellt wird, das Stereotyp «`use`» angegeben ist. Eine normale Assoziation drückt ja aus, daß die Ausgangsklasse, von der die Assoziation ausgeht, Referenzen auf Objekte der Zielklasse hält; genau dies ist hier aber nicht möglich, denn die Zielklasse ist ja abstrakt; die Ausgangsklasse wird aber Referenzen auf Objekte von Unterklassen der Zielklasse halten.

Literatur

[UML99] OMG Unified Modeling Language Specification (draft, Version 1.3 alpha R5, March 1999); OMG; 1999

[OOA] Kelter, U.: Lehrmodul "Objektorientierte Modellierung"; 2001/10

[SAR] Kelter, U.: Lehrmodul "Software-Architekturen"; 2002/10

[TAE] Kelter, U.: Lehrmodul "Transformation von Analyse-Datenmodellen in Entwurfsdokumente"; 2002

Glossar

Assoziation (*Assoziation*): Beziehung (in Entwurfsklassendiagrammen der UML stets gerichtet; ein- oder beidseitig gerichtet)

Container: Objekt, das dazu dient, eine Menge von Objekten einer zug. Basisklasse zu verwalten

Containerklasse: Klasse, deren Instanzen Container sind

Klasse, abstrakte: Klasse, von der keine Instanzen existieren und die nur dazu dient, gemeinsame Attribute mehrerer konkreter(er) Subklassen zu tragen

Klasse, assoziative: Klasse, die die Attribute einer Assoziation trägt; wird in der UML mit einer gestrichelten Linie mit der Assoziationslinie verbunden

Klassenattribut: Eigenschaft der Menge der Instanzen einer Klasse

Klassendiagramm (*class diagram*): Diagramm, das die Klassen eines Systems und deren Beziehungen (Subtypen, Komponenten, einfach Beziehungen u.a.) anzeigt; vereinfachte Darstellung zum Einsatz in der Systemanalyse, detailreicher zum Einsatz beim Architekturentwurf

Operation, abstrakte: Operation in einer Superklasse, die dazu dient, für gleichlautende Operationen mehrerer Unterklassen eine gemeinsame Schnittstelle zu realisieren; in der UML wird entweder der Operationsname kursiv geschrieben oder das Merkmal `abstract` angegeben

Sichtbarkeit: Merkmal von Attributen und Operationen einer Klasse in der UML; wird nur in Entwurfsklassendiagrammen benutzt; Bedeutung wird i.d.R. anhand der verwendeten Programmiersprache definiert (z.B. C++ oder Java)

Signatur: relevante Angaben zu einer Operation; besteht aus dem Namen der Operation, der Folge der Parametertypen und dem Typ des Rückgabewerts

Schnittstelle (*interface*): Klasse, die nur abstrakte Operationen enthält; dargestellt als Klasse mit mit dem Stereotyp «`interface`» oder durch einen kleinen Kreis, neben dem der Name der Schnittstelle steht

Index