Christian-Albrechts-Universität zu Kiel

Institut für Informatik

Seminar UML

Sommersemester 2001

Thema: Fortgeschrittene strukturelle Modellierung

Christian Buck

Matrikelnummer: 539623

Betreuer: Dr. Martin Steffen

Literatur

[1] Booch, Rumbauch, Jacobsen. The Unified Modeling Language User Guide, Verlag Addison-Wesley.

[2] Jens Schönborn. Seminarvortrag UML: Grundlagen struktureller Modellierung

Inhaltsverzeichnis

1 Einleitung

Ein Kernaspekt einer jeden Modellierungssprache wie auch der *Unified Modeling Language* `UML` ist die Forderung nach ausgereiften Mechanismen der Strukturierung. Diese sollen den Anforderungen aller Beteiligten genügen, also sowohl allgemeine Übersichten als auch detailierte implementierungsspezifische Infomationen für die Programmierer liefern können. Dazu sind starke Abstraktionen und fein unterschiedene Eigenschaften notwendig.
Diese Seminararbeit behandelt diese beiden Aspekte bezüglich der statischen Bereiche eines Modells. In Kapitel 2 werden die von `UML` bereitgestellten Abstraktionen vorgestellt. Kapitel 3 beschreibt die Beziehungen dieser abstrakten Objekte untereinander. Weiterhin werden in Kapitel 4 wichtige Details der Schnittstellen (`interfaces`) vertieft und in Kapitel 5 die `packages` als Gruppierungsmechanismus eingeführt. Kapitel 6 schließlich beschäftigt sich mit der Darstellung konkreter Zustände mittels Instanzen.
Diese Gliederung lehnt sich an `Section 2` von [1] an. Dort finden sich auch weiterführende Informationen zu den angesprochenen Themen.
Zur Lektüre dieser Arbeit werden Grundkentnisse in der objektorientierten Programmierung vorausgesetzt. Allerdings werden die meisten benötigten Begriffe im Anhang A nochmals kurz erklärt.
Ebenso sollten Basiskenntnisse der `UML`-Spezifikationen vorhanden sein, s. dazu auch [2] bzw. `Section 1` von [1].

2 Classifiers

2.1 Einführung

In jeder objektorientieren Umgebung bilden die sogenannten Klassen (`classes`) einen wichtigen Bestandteil. Sie dienen zur Abstraktion und Bündelung *logisch* zusammenhängender Elemente. Häufig möchte man aber auch andere Zusammenhänge abbilden, etwa die Kommunikation zwischen Objekten oder auch *physikalische* Zusammenhänge.
Dazu dienen in `UML` die `classifiers`. Sie vermögen mehr als nur die Gruppierung von Attributen (i.e. Variablen) und Operationen (i.e. Methoden). Nachdem im folgenden Unterkapitel die verschiedenen Arten von `classifiers` vorgestellt wurden, werden diese weiterführenden Charakteristiken vorgestellt.
Anmerkung: `Classifiers` haben sowohl *Struktur-* als auch *Verhaltens-*Eigenschaften, d.h. sie können sowohl Elemente eines Datentyps als auch Operationen auf diesen Datentypen enthalten. Eine Ausnahme bilden die `interfaces`: Sie besitzen keine Attribute (s.a. Kapitel 4).
Außerdem sind `classifiers` insbesondere die Objekte in `UML`, die Instanzen bilden können.

2.2 Arten von classifiers

`UML` kennt acht verschiedene `classifiers`. Im folgenden werden sie mit kurzer Erklärung vorgestellt, in Abb. 1 graphisch dargestellt :

- `class`: Wie aus objektorientierten Sprachen bekannt; besitzt Attribute und Operationen.
- `component`: Ein physikalischer, ersetzbarer Teil eines Systems. Erfüllt die Anforderungen von `interfaces`.

- `interface`: Ist eine Sammlung von Operationen, die die Funktionalität einer `class` oder `component` spezifizieren.
- `datatype`: Umfasst selbstdefinierte Datentypen ebenso wie die elementaren und enumerierten.
- `signal`: Umschreibt eine asynchrone Kommunikation zwischen Instanzen.
- `node`: Eine für die Laufzeit relevante physikalische Resource.
- `use case`: Eine umgangssprachliche oder auch konkrete Vorgehensbeschreibung für auftretene Fallunterscheidungen.
- `subsystem`: Eine Bündelung von Elementen, die einander in Anspruch nehmen.

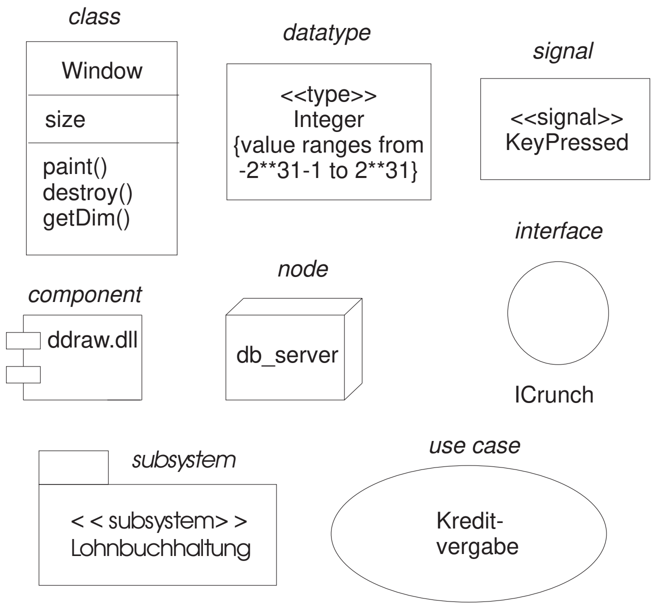

Abbildung 1: Classifier

2.3 Sichtbarkeiten

Es existieren drei Sichtbarkeitsattribute (`visibilities`) für Komponenten von `classifiers` in `UML`, die sich etwa an `Java` oder `C++` anlehnen:

- `public`: Alle `classifier`, die den adressierten `classifier` in ihrem Bindungsbereich haben, können auf die als `public` deklarierte Komponente zugreifen. Ein vorangestelltes "`+`" kennzeichnet dies.
- `protected`: Alle abgeleiteten `classifier` haben Zugriff; gekennzeichnet durch ein vorangestelltes "`#`".
- `private`: Nur der deklariende `classifier` selbst hat Zugriff; Kennzeichnung durch ein vorangestelltes "`-`".

S.a. Abb. 2 für ein Darstellungsbeispiel.

2.4 Bindungen

Ähnlich wie in `C++` lassen sich auch in `UML` Komponenten eines `classifiers` statisch binden (in `C++` mit dem Schlüsselwort `static`), d.h. jede Instanz des `classifiers` besitzt dieselbe Bindung an die Komponente. Dies wird als die `classifier`-Bindung bezeichnet; existiert für jede Instanz eine neue Komponente, so wird dies als `instance`-Bindung bezeichnet.
Graphisch wird eine `classifier`-Bindung durch Unterstreichen des Namens der Komponente dargestellt, s.a. Abb. 2 für ein Beispiel.

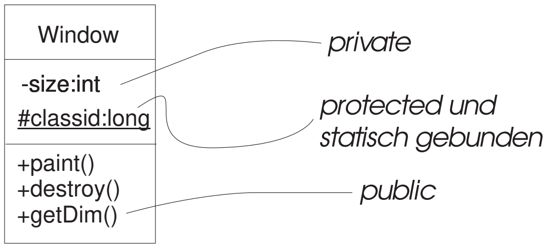

Abbildung 2: Sichtbarkeit und Bindungen

2.5 Abstraktion, Vererbung und Polymorphie

Ein weiterer wichtiger Mechanismus in objektorientierten Umgebungen sind die *abstrakten* Klassen. Sie haben keine Instanzen, sondern stellen nur Vorgaben zur Verfügung, die von ihnen abgeleitete Klassen erfüllen müssen, d.h. abstrakte Attribute und Operationen müssen deklariert werden. Abstrakte Klassen werden in `UML` dadurch gekennzeichnet, daß ihre Namen kursiv geschrieben werden.
Weiterhin existieren in `UML` noch Möglichkeiten, die Vererbung von Klassen einzuschränken. Wird eine Klasse als `leaf`-Klasse bezeichnet (indem das Schlüsselwort `leaf` in geschweiften Klammern unter den Klassennamen geschrieben wird), so kann von dieser Klasse nicht mehr geerbt werden. Analog können `root`-Klassen erstellt werden, die nicht erben (aber natürlich weiterhin *vererben*) können.
Die eben dargestellten Eigenschaften lassen sich auch auf Operationen übertragen. So kann eine Operation abstrakt sein, d.h. sie stellt an erbende Klassen die Anforderung, sie zu implementieren. Auch die Polymorphie von Operationen lässt sich in `UML` einschränken; wird eine Operation mit dem Schlüsselwort `leaf` (in geschweiften Klammern hinter dem Namen) bezeichnet, so ist sie nicht polymorph, darf also von erbenden Klassen nicht überschrieben werden.
Abb. 3 verdeutlich dies am Beispiel.

2.6 Multiplizität

Auch die *Anzahl* der erlaubten Instanzen einer Klasse lässt sich einschränken; dies ist die sogenannte Multiplizität (`multiplicity`). Sie kann eine konkrete Zahl (etwa `5`) oder auch ein Intervall (etwa `[2..*]`) sein. Graphisch wird sie durch Aufschreiben in der oberen rechten Ecke des Klassensymbols dargestellt. Die Voreinstellung ist eine beliebig große Anzahl; diese wird wie im obigen Beispiel mit dem Symbol “`*`“ angezeigt.
Multiplizität lässt sich auch auf Attribute (also Elemente eines Datentyps) von Klassen anwenden; dort schreibt man sie wie schon aus 2.5 bekannt hinter den Namen der Komponente.

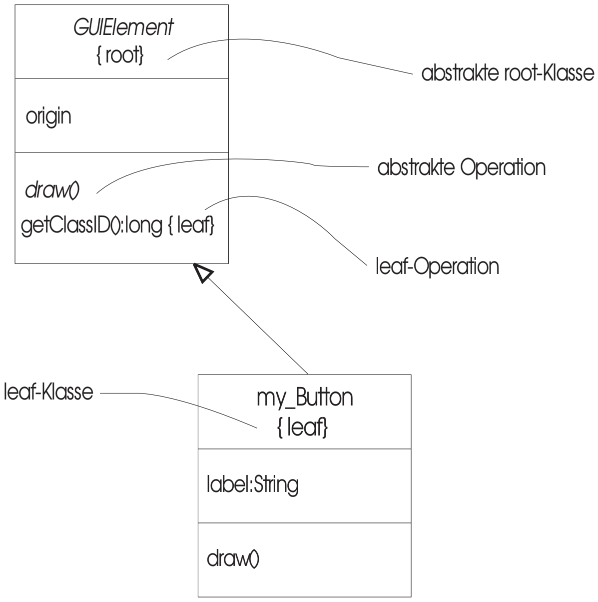

Abbildung 3: Abstraktion und Vererbung

2.7 Attribute: Weitere Eigenschaften

Neben den in 2.3-2.6 beschriebenen gibt es noch weitere Eigenschaften von Attributen. Die vollständige Syntax eines Attributes lautet:

```
[Sichtbarkeit] name [Multiplizitaet] [:Datentyp] [=Intialwert]
 [{weitere Eigenschaften}]
```

Die folgenden drei weiteren Eigenschaften können verwendet werden :

- `changeable`: Ist die Voreinstellung für Attribute. Es gibt keine Einschränkungungen, was das Verändern der Werte betrifft.
- `addOnly`: Besitzt das Attribut eine Multiplizität grösser als `1`, so können zusätzliche Werte zwar hinzugefügt, danach aber nicht mehr geändert oder gelöscht werden.
- `frozen`: Nachdem das Attribut initialisiert wurde, kann der Wert nicht mehr verändert werden (dies entspricht dem Schlüsselwort `const` in `C++`).

Angewandt sieht dies beispielhaft so aus:

```
i:Integer=42 {frozen}
```

2.8 Operationen: Weitere Eigenschaften

Wie es auch schon bei den Attributen einer Klasse der Fall war, gibt es auch für Operationen in `UML` weitere Eigenschaften. Die vollständige Syntax einer Operation lautet wiefolgt:

```
[Sichtbarkeit] name [(Parameterliste)] [:Rueckgabetyp][{weitere
 Eigenschaften}]
```

Dabei besitzt ein Parameter die Syntax

`[direction] name : Datentyp [=Default-Wert]`

`Direction` bezeichnet die Modifizierbarkeit eines Parameters und kann folgende Werte annehmen:

- `in`: Der Parameter ist ein Eingabe-Parameter und darf nicht verändert werden.
- `out`: Der Parameter ist ein Ausgabe-Parameter und darf verändert werden.
- `inout`: Der Parameter ist ein Eingabe-Parameter und darf verändert werden.

Bemerkung: Diese Notation der `direction` ist dem Leser vielleicht schon von der Sprache `Ada` bekannt.

Als weitere Eigenschaften einer Operation (neben der in 2.5 vorgestellten `leaf`-Eigenschaft) sind möglich:

- `isQuery`: Die so bezeichnete Operation verändert den Zustand des Systems nicht, ist also frei von Seiteneffekten.
- `sequential`: Die Operation ist nicht synchronisiert, es ist also die Konsitenz der Zustände bei (quasi-)parallelen Aufrufen nicht gewährleistet.
- `guarded`: Die Operation ist weiterhin nicht synchronisiert, stellt aber sicher, daß (quasi-)parallele Zugriffe *hintereinander* ausgeführt werden.
- `concurrent`: Die Operation ist synchronisiert, da sie als atomar behandelt wird.

2.9 Template-Klassen

`Template`-Klassen (Schablonen-Klassen) sind *parametrisierte* Klassen. Sie erlauben eine flexible Datentypbindung und sind mit Makros zu vergleichen, unterliegen aber im Gegensatz zu diesen den üblichen Syntaxregeln des Compilers. Ein Beispiel verdeutlicht ihre Funktion besser (`C++`-Quelltext):

```
template<class X>
 class Tools{
  public:
   void swap(X &a,X &b){
         X temp;
         temp=a;
         a=b;
         b=temp;
   }
 }
int x,y;
Tools<int> t; t.swap(x,y);
```

Das `<class X>` bezeichnet dabei, daß der der Funktion unterliegende Datentyp erst bei *Instanziierung* bekannt wird. Die Instanziierung von `Template`-Klassen lässt sich auf zwei Arten modellieren; Abb. 4 zeigt dieses.

2.10 Standardelemente für Klassen

`UML` kennt vier Standardstereotypen, die auf Klassen angewandt werden können:

- `metaclass`: Alle Komponenten der Klasse sind selbst Instanzen von Klassen.

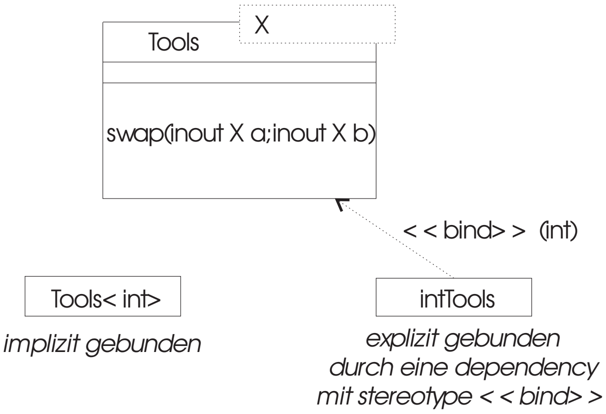

Abbildung 4: Template-Klassen

- `powertype`: Alle Komponenten der Klasse sind von einer Superklasse geeerbt.
- `stereotype`: Die Klasse ist selbst ein Stereotyp.
- `utility`: Die Komponenten der Klasse sind alle statisch gebunden.

3 Erweiterte Relationships

3.1 Einführung

Wann immer ein objektorientiertes System modelliert wird, müssen auch Zusammenhänge zwischen Elementen dieses Systems dargestellt werden. Diese Zusammenhänge können vielfältiger Natur sein: Eine Klasse erbt etwa von anderen Klassen, implementiert ein `interface` oder teilt sich Variablen mit anderen Instanzen von Klassen. Im folgenden werden die vier wichtigsten Zusammenhangsarten (`relationships`) in `UML` vorgestellt und ihre graphische Darstellung gezeigt.

3.2 Dependency

Eine `dependency` beschreibt einen Zusammenhang, bei dem eine Änderung in einer Komponente sich auch auf andere Komponenten auswirken kann. In `UML` sind 17 verschiedene spezielle `dependencies` definiert, die hier (kurz!) vorgestellt werden. Will man aber nur einen allgemeinen Zusammenhang aufzeigen, so muß natürlich keiner der folgenden Stereotypen benutzt werden.

1. `bind`: Die angegebe Quelle der `dependency` instaziiert das Ziel-`template` mit den angegebenen Parametern.
2. `derive`: Die Quelle kann aus dem Ziel errechnet werden.
3. `friend`: Die Quelle hat spezielle Sichtbarkeit dem Ziel gegenüber.
4. `instanceOf`: Die Quelle ist eine Instanz des Ziels.
5. `instantiate`: Die Quelle erstellt Instanzen des Ziels.
6. `powertype`: Das Ziel ist ein `powertype` der Quelle, hat also all seine Komponenten von der Quelle geerbt.
7. `refine`: Die Quelle ist eine verfeinerte Abstraktion des Ziels.

8. `use`: Die Semantik der Quelle ist abhängig von der Semantik des Ziels.
9. `access`: Das Quell-`package` (s.a. Kap. 5) darf Elemente des Ziel-`package` referenzieren.
10. `import`: Die Elemente des Ziel-`packages` werden dem Bindungsbereich der Quelle hinzugefügt.
11. `extend`: Das Ziel-`use case` erweitert das Verhalten des Quell-`use case`.
12. `include`: Das Quell-`use case` enthält das Verhalten des Ziel-`use case`.
13. `become`: Das Ziel ist dasselbe Objekt wie die Quelle, allerdings zu einem späteren Zeitpunkt und wahrscheinlich anderem Zustand / mit andererer Verwendung.
14. `call`: Die Operation an der Quelle benutzt die Zieloperation.
15. `copy`: Das Zielobjekt ist eine *eigenständige* Kopie des Quellobjektes.
16. `send`: Die Operation an der Quelle löst das `event` am Ziel aus.
17. `trace`: Das Ziel ist irgendwann einmal aus der Quelle hervorgegangen.

3.3 Generalization

Mit `generalizations` werden in `UML` Vererbungsbeziehungen dargestellt. Diese Eigenschaft in objektorientierten Systemen sollte dem Leser hinlänglich bekannt sein und wird daher nicht weiter erläutert. Zu beachten ist allerdings, daß die `generalization` die *inverse* Vererbung bezeichnet, d.h. die Begriffe der Super- und Subklasse sind vertauscht. Im folgenden werden die Notationen wie gewohnt gewählt, also ist die Superklasse die vererbende und die Subklasse die erbende.
Auch die `generalizations` können in ihrer Bedeutung durch `constraints` verfeinert werden. `UML` kennt die folgenden:

- `complete`: In dem dargestellten Modell sind alle Superklassen verzeichnet und es werden keine weiteren hinzukommen.
- `incomplete`: Nicht alle Superklassen sind verzeichnet, es dürfen noch weitere hinzukommen.
- `disjoint`: Wird nur bei multipler Vererbung angewandt; mit `disjoint` werden Mengen von `generalizations` angezeigt, bei denen eine Subklasse nur genau einen Vater besitzt.
- `overlapping`: Ist das Gegenteil von `disjoint`; es werden Menge von Subklassen angezeigt, die mehr als einen Vater besitzen.

Weiterhin existiert noch ein Stereotyp:

- `implementation`: Die Subklasse erbt zwar Implementationen der Superklasse, macht sie aber nicht `public` und unterstützt nicht ihre `interfaces`.
  Damit wird angezeigt, daß die Subklasse andere Verhaltenseigenschaften besitzt.

3.4 Association

Eine `association` ist im Gegensatz zu einer `dependency` oder `generalization` ein rein struktureller Zusammenhang, sagt also nichts über die Ursachen von konkreten Zuständen aus. `UML` unterscheidet zwischen sieben Verfeinerungen von `associations`:

- `navigation`: Von Natur aus ist jede `association` bidirektional. Allerdings will man unter Umständen die Richtung der `association` einschränken; dies erreicht man durch Angabe eines geöffneten Pfeils am Ende der Richtung (s.a. Abb. 5).
  Achtung: Die Angabe einer `navigation` bedeutet *nicht*, daß es unmöglich ist, die `association` auch in anderer Richtung zu durchlaufen, sondern ist nur eine verständniserleichterne Maßnahme.

- `visibility`: Auch bei einer `association` lassen sich wieder die schon bekannten Sichtbarkeiten angeben. Sie werden wie in Abb. 5 benutzt. Die voreingestellte Sichtbarkeit ist `public`. Ist ein Objekt als `private` bezeichnet, so darf es von keinem fremden Objekt benutzt werden; ist es `protected`, so dürfen nur Objekte vom anderen Ende der `association` auf es zugreifen.

- `qualification`: Eine `qualification` dient dazu, zwei Objekte, die durch eine `association` verbunden sind, miteinander in Zusammenhang zu bringen. Sie ist dabei ein Attribut der `association` und nicht zwingend Attribut des Quell- oder Zielobjektes. In Abb. 5 ist ein Beispiel gegeben: `auftrags#` ist weder ein Attribut von `Werkstatt` noch von `KaputtesAuto`, spezifiziert aber die `association` zwischen den beiden.

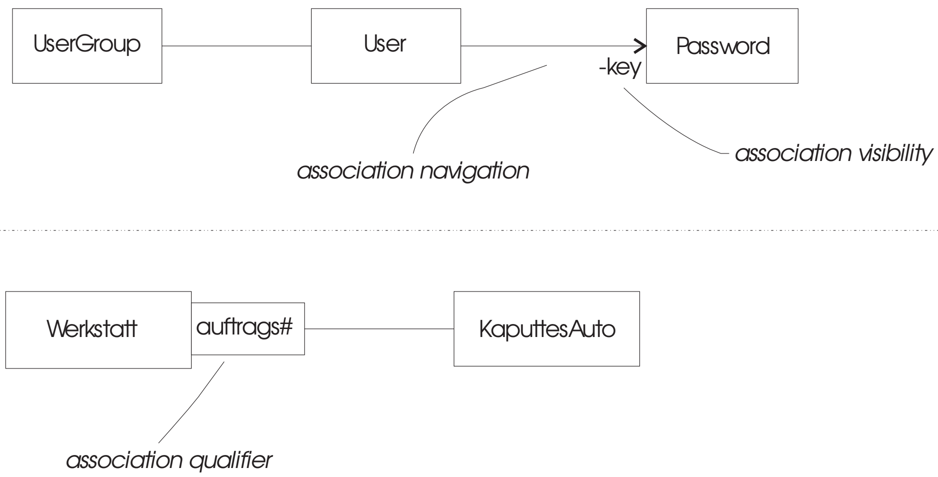

Abbildung 5: navigation, visibility, qualification

- `interface specifier`: Wie schon bekannt, kann eine `class` bzw. `component` mehrere `interfaces` implementieren. Soll jedoch nicht jedes `interface`, sondern nur etwa eines nach außen wirksam werden, so geschieht dies durch Angabe einer `role` (hier: eines aussagekräftigen Namens; s.a. Kap. 4) und des betreffenden `interfaces`. Abb. 6 stellt dies durch ein Beispiel dar.

- `composition`: Die `composition` stellt eine Erweiterung des Prinzips der `aggregation` dar. Im Gegensatz zu dieser, bei der ein Element auch von mehreren umfassenden Strukturen geteilt werden kann, ist bei der `composition`

das Element streng an ein anderes gebunden. Weiterhin ist dieses Oberelement verantwortlich für das Unterelement, etwa muss es die Destruktion veranlassen. S.a. Abb. 6 für ein Beispiel.
Bemerkung: `composition` lässt sich auch auf andere Form graphisch darstellen: Die Symbole der Unterelemente werden einfach in das Symbol des Oberelementes eingebettet.

- `association classes`: Wird eine `association` zwischen zwei `classes` modelliert, so kann diese Klassencharakter bekommen. Im Beispiel in Abb. 6 besteht zwischen der `Firma` und der `Person` die Beziehung der Arbeitsbeschäftigung, dort dargestellt durch die `association class Job`. Würde `Job` als eigenständige Klasse und als Zwischenglied der `association` modelliert werden, so würde nicht klar genug das spezielle Verhältnis zwischen den jeweiligen Instanzen `Firma` und `Person` dargelegt werden. Dies geschieht nun durch die `association classes`.

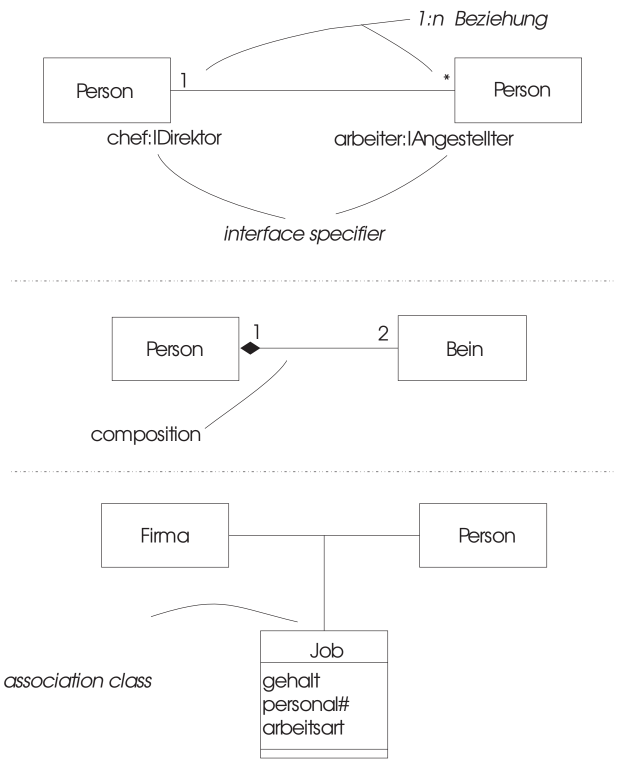

Abbildung 6: interface specifier,composition,association class

Weiterhin lassen sich noch weitere `constraints` auf die eben erwähnten `associations` anwenden. Von ihnen gibt es insgesamt fünf Stück:

- `implicit`: Zeigt an, ob ein Zusammenhang tatsächlich auftritt oder nur implizit ist (etwa: Zwei von derselben Vaterklasse abgeleiteten Klassen besitzen eine *implizite* Beziehung).

- `ordered`: Spezifiziert, daß die Menge von Objekten am einen Ende der `association` auf eine bestimmte Art geordnet ist (etwa: lexikalisch absteigend usw.).

- `changeable, addOnly, frozen`: Diese Schlüsselwörter beziehen sich auf die Veränderbarkeit der Instanzen am Ende der `association` und bezeichnen die schon bekannten Eigenschaften aus Kap.2.7.

3.5 Realization

In dem hier betrachteten Kontext ist eine `realization` eine semantische Verpflichtung einer `class` oder `component` einem `interface` gegenüber. Inhaltlich stellt sie die Anforderung der Implementierung des `interface` dar.
Sie ist als Kreuzung zwischen einer `dependency` und einer `generalization` zu sehen, und als solche wird sie auch graphisch dargestellt: Als gestrichelte Linie mit einem geschlossenen, nicht gefüllten Pfeil am Ende. Dies Darstellung wird die *kanonische* (`canonical`) Form genannt (s.a Abb. 7). Allerdings gibt es auch die vereinfachte Art der Darstellung, in der nur das aus Abb. 1 bekannte Symbol zum Ausweisen des `interface` benutzt wird. In ihr werden keine Operationen des `interface` bekanntgegeben. Sie wird die *elidierte* (`elided`) Form genannt.

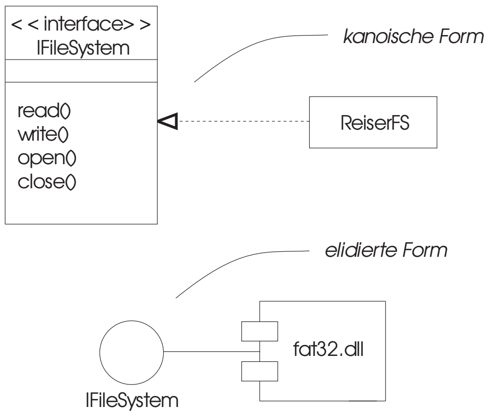

Abbildung 7: realization

4 Interfaces, Types und Roles

4.1 Einführung

Schnittstellen, also `interfaces`, stellen einen wichtigen Bestandteil eines jeden Systems dar. Sie dienen zum einen dazu, Informationen über die *Funktionalität* ohne direkte Kenntnis über ihre Semantik (bzw. ihre Implementierung) bereitzustellen. Zum anderen stellen sie einen wichtigen Mechanismus zur Integritätssicherung bei der Erweiterung oder Veränderung eines bestehenden Systems dar, indem (allgemein ausgedrückt) eine Kanalisierung des Informationsflußes stattfindet.
Die graphische Darstellung fand sich bereits in Abb. 7, nun folgen noch weitere Details zu `interfaces`.
Bemerkung: Wie bereits erwähnt, können `interfaces` weder Attribute enthalten noch instanziiert werden.

4.2 Namen von interfaces

Per Konvention wird in `UML` zur Kennzeichnung eines `interface` dem Namen ein '`I`' vorangestellt (etwa `IDraw`). Er allein wird in `UML` als *simple name* bezeichnet;

bei Angabe des umfassenden `package` als *path name* (etwa `Graphic::IDraw`).
Als verwendbare Zeichen für den Bezeichner stehen die alphanumerischen Ziffern sowie alle Interpunktionssymbole bis auf den Doppelpunkt '`:`' und den Punkt '`.`' zur Verfügung.

4.3 Types und Roles

Eine Klasse darf natürlich mehrere `interfaces` implementieren. Betrachtet man aber eine Instanz einer solchen Klasse in einem speziellen Kontext, so möchte man eventuell nur einen Teil der von ihr realisierten `interfaces` beachten. Ein solches `interface` stellt dann die *Rolle*, die die Klasse spielt, dar. Diese `role` ist dann also das Verhalten in einer bestimmten Situation. Die Rollen habe wir bereits im Kap. 3.4 bei der Einführung der `interface specifier` kennengelernt; in Abb.6 ist ein Beispiel gegeben:
Die `Person` kann in verschiedenen Situationen auftreten, z.B. als Angestellter, Kunde, Lieferant usw. Im Beispiel spielt sie zum einen die Rolle `chef` vom Typ `IDirektor`,zum anderen die Rolle `arbeiter` vom Typ `IAngestellter`, d.h. im betrachteten Kontext ist nur das Verhalten der Klasse `Person` als `arbeiter` und `chef` interessant.

Anmerkung: `UML` stellt eine stereotype Klasse `type` zur Verfügung, die es erlaubt (im Gegensatz zur Abstraktion mittels `interfaces`) auch Attribute in die Definition mit aufzunehmen. Bei gewissen Modellierungsaufgaben sollte sich das als hilfreich erweisen.

5 Packages

5.1 Einführung

`Packages` sind dem Leser vielleicht schon aus der Sprache `Java` o.ä. bekannt. Sie dienen dazu, Klassen zu einer Gruppe zusammenzufassen, so daß etwa eine zentrale Manipulation oder Sichtbarkeits- und Zugriffsbeschränkungen möglich sind. Selbstverständlich sollten semantisch dicht beieinanderliegende Elemente in ein `package` zusammengeführt werden und klare Zugriffsmethoden auf diese Elemente definiert sein.
In der Softwareentwicklung haben gerade diese Eigenschaften die `packages` zu einem wichtigen Instrument gemacht; verschiedene Entwicklerteams können (nachdem gemeinsam eine Interfacestruktur definiert wurde) ohne Kenntnis voneinander ihren strukturell unabhängigen Bereich bearbeiten. Die Vorteile liegen auf der Hand: Durch die Partionierung müssen die Entwickler nur über ihren eigenen Bereich den Überblick bewahren, die Integration verschiedener Programmfragmente fällt leichter und durch den modularen Aufbau wird die Fehlerbeseitigung bzw. das komplette Ersetzen einer Einheit wesentlich einfacher.

5.2 Namen und Arten von packages

Die Namenskonventionen für `packages` sind ähnlich denen für `interfaces`, s.a. Kap. 4.2: Der Name eines `package` allein wird als *simple name* bezeichnet (etwa `Buttons`). Wird es von einem anderen `package` umfasst und gibt man dieses auch an, so ist dies der *path name* (etwa `Windows::Buttons`). Als verwendbare Zeichen stehen wieder die alphanumerischen sowie die Interpunktionssymbole bis auf den Doppelpunkt '`:`' und den Punkt '`.`' zur Verfügung.
In `UML` gibt es zwei Arten der graphischen Darstellung von `packages`, die normale

und die erweiterte (`extended`) Form, welche Teile ihrer oder alle Komponenten ausweist. S. Abb. 8 für Details.

5.3 Komponenten und Sichtbarkeiten

Ein `package` darf alle uns bekannten `classifier` und wie schon erwähnt andere `packages` enthalten. Die Beziehung eines `package` zu seinen Komponenten ist die uns schon bekannte `composition association` (s. Kap.3.4), d.h. ein Element eines `package` gehört nur zu diesem und wird beim Auflösen des `package` ebenfalls aufgelöst. Ein `package` definiert einen Bindungsbereich, d.h. Elemente gleichen Typs müssen verschiedene Bezeichner erhalten.
Zur graphischen Modellierung von Komponenten stehen zum einen die textuelle Verschachtelung (die Bezeichner werden in das Symbol des `package` geschrieben) und zum anderen die graphische Verschachtelung (die Symbole der Komponenten werden in das Symbol des `package` eingefügt) zur Verfügung; Abb. 8 illustriert dies.
Die Sichtbarkeiten der Komponenten sind wie schon bekannt: Auf Elemente, die als `public` deklariert sind, darf immer zugegriffen werden, auf die, die als `protected` gekennzeichnet sind, nur von abgeleiteten Elementen und schliesslich auf die, die `private` sind, nur innerhalb des `package`.

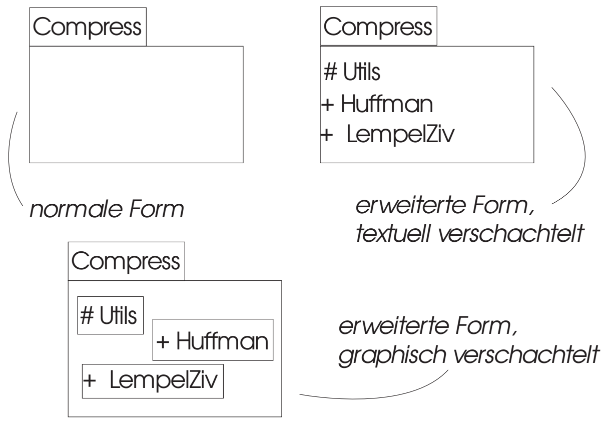

Abbildung 8: packages

5.4 Import und Export

Soll auf die Komponenten eines `package` zugegriffen werden, so muß dieses *importiert* werden. In `Java` geschieht dies beispielweise durch folgende Anweisung:

```
import java.awt.*;
```

Dadurch werden alle Komponenten des `package` '`awt`', welches sich im `package` '`java`' befindet, dem Bindungsbereich des aktuellen Programmes hinzugefügt. In `UML` wird eine `import`-Beziehung als eine `dependency` mit dem Stereotyp `import` dargestellt (s.a. Kap.3.2).

Bemerkung: Ebenfalls möglich ist die Angabe des Schlüsselwortes `access`. Dabei werden die Komponenten des `package` nicht dem Bindungsbereich hinzugefügt, sondern der Zugriff muß über den voll qualifizierten Namen der Komponente erfolgen, z.B. `okButton=new java.awt.Button();`. Dies verhindert Bezeichnerkollisionen, falls mehrere Komponenten verschiedener `packages` gleiche Bezeichner tragen.

Die Teile eines `package`, die als `public` deklariert wurden, nennt man den `export` eines `package`.
Zu beachten ist, daß `import dependencies` nicht transitiv sind; importiert also `a` die Komponenten von `b` und `b` importiert `c`, so sind die Komponenten von `c` nicht an `a` gebunden.

5.5 Vererbung

Neben der eben dargestellten `import`- bzw. `access`-Beziehung gibt es noch eine zweite mögliche `relationship` bei `packages`, nämlich die `generalization`, also die Vererbung. Sie funktioniert genauso wie bei der `classifier`-Vererbung (s.a. Kap.2.5). Abb. 9 zeigt ein Beispiel: die `WinNTGUI` erbt von der `WinGUI`, überschreibt aber die Klasse `Window` und stellt eine neue Klasse `NTWindow` zur Verfügung. Prizipiell hat `WinNTGUI` aber die selben Verhaltenseigenschaften wie `WinGUI`.

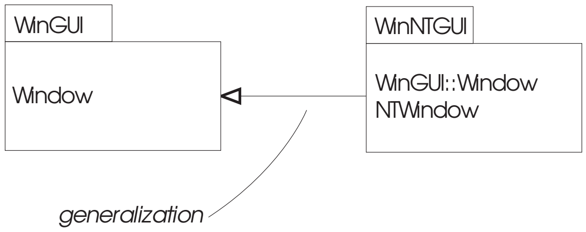

Abbildung 9: generalization bei packages

5.6 Standardelemente

`UML` kennt fünf Standardstereotypen, die auf `packages` angewandt werden können:

- `facade`: Das so bezeichnete `package` ist eine vereinfacht organisierte Sicht auf eine Teilmenge eines anderen `package`. Wird benutzt, falls nur bestimmte Teile eines sehr grossen und komplexen `package` betrachtet werden sollen.
- `framework`: Das `package` ist eine Musterimplementierung.
- `stub`: Der `export` des `packages` ist eine 'Dummyimplementierung', besteht also etwa nur aus Methoden mit leerem Rumpf und standardinitialisierten Variablen.
- `subsystem`: Zeigt auf, daß das `package` ein *unabhängiger* Teil des ganzen Systems ist.
- `system`: Das `package` repräsentiert das ganze betrachtete System.

6 Instances

6.1 Einführung

Bisher haben wir nur abstrakte Elemente von `UML` betrachtet, etwa `classes` als eine Abstraktion und Kollektion von logisch zusammenhängenden Dingen. In tatsächlichen Programmen benötigt man aber konkrete Elemente, die sogenannten *Instanzen* oder *Objekte*. Sie sind eine Manifestation einer Abstraktion (hier: eines

`classifier`), sind also real existierend, benötigen daher Speicherplatz und können Zustände annehmen. Man unterscheidet zwischen *direkten* und *indirekten* Instanzen; ein Objekt einer Klasse, die ein `interface` implementiert, ist eine indirekte Instanz dieses `interface` und eine direkte Instanz ihrer zugrundeliegenden Klasse. Man rufe sich in Erinnerung, daß von abstrakten Klassen und von `interfaces` keine direkten Instanzen gebildet werden können.
Zum Einsatz kommen Instanzen zumeist bei der Modellierung von Interaktions- und Aktivitätsdiagrammen, bei denen konkrete Zustände den Programmfluß bestimmen.

6.2 Namen und Arten von Instanzen

Beim Vergebenen des Namens von Instanzen dürfen wieder alle alphanumerischen Zeichen und Interpunktionssymbole bis auf den Doppelpunkt '`:`' und den Punkt '`.`' verwendet werden. Per Konvention werden der erste Buchstabe klein und bei zusammengesetzten Wörtern die folgenden Anfangsbuchstaben groß geschrieben (etwa: `mainWindowListener`). Wie bereits bekannt, ist der Bezeichner allein der sogenannte *simple name* und der mit Angabe des umfassenden `package` der *path name*.
Ist die zugrundeliegende Abstraktion bei der Modellierung nicht bekannt, so kann ein sogenanntes *orphan object* eingesetzt werden (etwa: `irgendeineInstanz:`). Ist dagegen nicht der tatsächliche Name einer Instanz bekannt (etwa weil eine Methode modelliert wird, die mit verschiedenen Instanzen aufgerufen wird), so kann auch ein sogeanntes *anonymus object* benutzt werden (etwa: `:String`). Weiterhin existieren die *multiobjects* als Zusammenfassung von mehreren Instanzen selben Typs. Dies dient nur der Übersichtlichkeit, falls die konkreten Zustände der einzelnen Instanzen bei Betrachtung nicht von Belang sind.
Abb. 10 verdeutlicht dies am Beispiel.

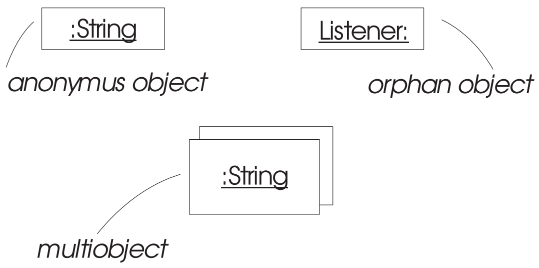

Abbildung 10: Arten von Instanzen

6.3 Operationen, Zustände

Wie bekannt ist, sind Instanzen etwas reales. Damit werden auch die Operationen, die etwa eine Klasse bereitstellt, zu etwas konkretem, den *Methoden*. Mit ihnen kann auf den Objekten gearbeitet werden, d.h. eine Methode ist an eine Instanz (und nicht etwa an eine Klasse) gebunden. Ihre Benutzung sollte hinlänglich bekannt sein: Ist `a` eine Instanz einer Klasse und `foo()` eine Operation dieser Klasse, so wird durch `a.foo();` die Methode `foo()` der Instanz `a` aufgerufen.
Bemerkung: Selbstverständlich kann `foo()` auch polymorph sein (s. Kap.2.5), der Einfachheit halber wird dies aber nicht gesondert betrachtet.

Nicht nur die Methoden manifestieren sich in einer Instanz, sondern auch

die Attribute, also die Variablen mit konkreten Zuständen (d.h. Werten). Sollen sie zusammen mit einer Instanz modelliert werden, so schreibt man ihren Bezeichner und ihren Wert in das Symbol der Instanz (s.a. Abb. 11). Wahlweise lässt sich auch ihr Typ angeben, obwohl der selbstverständlich durch die Klassendefiniton bekannt ist.
Außerdem kann anstelle von Instanzvariablen auch ein Zustand eines mit der zugehörigen Klasse assoziierten Automaten angeben. Dies kann sich bei der Modellierung der Lebenszeit einer Klasse als nützlich erweisen.
Weiterhin bietet `UML` die Möglichkeit, *aktive* Objekte (`active objects`) graphisch anzuzeigen. Dies ist z.B. bei der Betrachtung von nebenläufigen Prozessen von Interesse: Der jeweils aktive Prozess wird durch einen dicken schwarzen Rahmen um das Instanzsymbol angezeigt.

Achtung: Es darf nie vergessen werden, daß die Betrachtung von Instanzen immer einen zeitlichen 'Schnappschuß' voraussetzt, d.h. der Vergleich von zeitlich verschiedenen Instanzaufnamen darf nie davon ausgehen, daß etwa die konkreten Werte von Variablen unverändert geblieben sind.

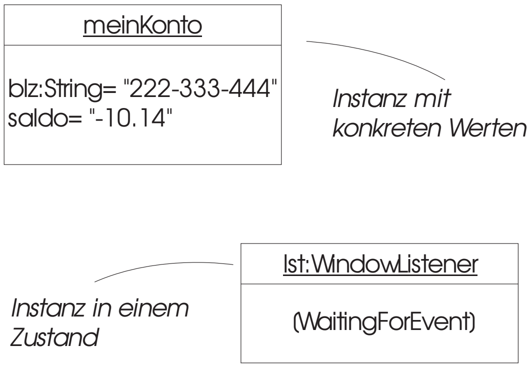

Abbildung 11: Instanzen mit Zuständen

6.4 Standardelemente

Alle bisher bekannten Standardelemente lassen sich auch auf Instanzen anwenden. Hier sollen jedoch noch einmal diejenigen erwähnt werden, die sich speziell auf Instanzen bzw. Beziehungen zwischen Instanzen beziehen:
Stereotypen:

- `instanceOf`: Die Quelle ist eine Instanz des Ziels.
- `instantiate`: Die Quelle erstellt Instanzen des Ziels.
- `become`: Die Quelle ist *dasselbe* Objekt wie das Ziel, allerdings zu einem späteren Zeitpunkt (und mit vermutlich anderem Zustand).
- `copy`: Die Quelle ist eine exakte Kopie des Zielobjektes, aber unabhängig.

`Constraint`:

- `transient`: Bezieht sich auf `roles` und bezeichnet, daß die betreffende Instanz nur vorübergehend besteht und während des modellierten Vorgangs aufgelöst wird.

A Kurzes Glossar

Bemerkung: Im folgenden werden etwa bei der Beschreibung von Attributen nur Klassen als umfassendes Element genannt. I.a. sind alle Eigenschaften aber bei jedem `classifier` gleich, außer daß `interfaces` keine Struktureigenschaft haben und nicht instanziiert werden können.

- `Attribute`: Die Attribute einer Klasse bestimmen ihre *Struktur*. Sie sind abstrakt; eine Klasse muß also erst instanziiert werden, damit aus den Attributen konkrete Zustände, also *Variablen* werden.
- `Constraint`: Stellt eine erweiterte Eigenschaft eines Elements dar; wird in geschweiften Klammern `{...}` an das Element geschrieben.
- `Instanz`: Eine Klasse selbst ist ein abstraktes Konstrukt. Sie stellt nur eine Vorlage für eine konkrete Struktur, einer Instanz einer Klasse, dar. Erst bei Instatiierung einer Klasse existieren die dort definierten Operationen und Attribute. Bei der Instanzbildung wird Speicher für die enthaltenen Variablen alloziert und mit Standard-/Initialwerten belegt.
- `Klasse`: Wichtiges Konstrukt in der objektorientierten Programmierung. In einer Klasse sind i.a. Attribute und Operationen auf diesen Attributen gekapselt. Klassen sind eine logische Abstraktion nach der Idee, daß alles, was zusammengehört, sich auch zusammen organisiert werden soll. So stellen Klassen meist Operationen auf den ihnen eigenen Attributen zur Verfügung. Klassen können erben und vererbt werden; erben sie eine Methode mit gleicher Signatur, so *überschreiben* sie diese, Variablen mit gleichem Bezeichner werden *überdeckt*, existieren also weiterhin, müssen aber gesondert referenziert werden.
  Bemerkung: Wird die Eigenschaft der Polymorphie/Vererbung außer acht gelassen, so können Klassen mit dem `struct`-Blockdatentyp der Sprache `C` verglichen werden - nur mit ebenfalls enthaltenen Funktionen.
- `Komponenten`: Die Gesamtheit von Attributen und Operationen etwa einer Klasse oder auch der Instanz einer Klasse (dort also die Gesamtheit von Variablen und Methoden, auch der geerbten). Nicht zu verwechseln mit der `component` (s. Kap.2.2)!
- `Objekt`: Synonym für *Instanz*.
- `Operationen`: Die Operationen einer Klasse bestimmen ihr *Verhalten*. Operationen sind abstrakt, d.h. es muß erst eine Instanz der Klasse gebildet werden, so daß konkrete Funktionen entstehen, die sogenannten *Methoden*.
- `Signatur`: Reihenfolge, Typ und Anzahl der formalen Parameter.
- `Stereotype`: Stellt (allgemein) eine bei der Umsetzung in ein Programm häufig auftretene Situation dar; wird in spitzen Klammern `<<...>>` an das betroffene Element geschrieben (bei Attributen: An das Ende der vollständigen Darstellung; bei `classifiers`: Unter den Bezeichner).
- `Struktureigenschaft`: s. *Attribute*.
- `Subklasse`: Erbt eine Klasse Komponenten einer anderen, so wird die erbende Klasse Tochter- oder Subklasse genannt.
- `Superklasse`: Erbt eine Klasse Komponenten einer anderen, so wird die vererbende Klasse Vater- oder Superklasse genannt.
- `Verhaltenseigenschaft`: s. *Operationen*.