30 Entwurfskonzepte

Im Entwurf werden vornehmlich das Entwurfsklassenmodell sowie Interaktions- und Zustandsdiagramme erstellt. Wie bereits in der Softwarespezifikation steht dabei das Klassenmodell eindeutig im Vordergrund. Die in diesem Kapitel behandelten Grundkonzepte des Entwurfs schlagen sich daher in erster Linie im Klassenmodell nieder.

Die beiden nächsten Abschnitte beschäftigen sich mit den Entwurfsprinzipien »Geheimnisprinzip« sowie »schwache Kopplung« und »starke Kohäsion«, welche die Qualität von Entwurfspezifikationen entscheidend bestimmen. Danach wird das Thema Konformität behandelt, um das Konzept der Generalisierung zu präzisieren. Der darauf folgende Abschnitt stellt den »Entwurf mit Verträgen« vor, eine von Bertrand Meyer entwickelte Methode für eine klar geregelte Kommunikation zwischen Klassen bzw. Objekten. Den Abschluss des Kapitels bilden das Interface-Konzept, das einen wichtigen Spezialfall der Generalisierung darstellt, sowie einige Bemerkungen zur Modellierung generischer Klassen, die quasi als Schablonen zur allgemeinen Realisierung wiederholt benötigter Funktionalitäten dienen, die sich konkret nur um bestimmte Nuancen unterscheiden.

30.1 Geheimnisprinzip

Entwürfe hoher Qualität befolgen das *Geheimnisprinzip*. Dieses Prinzip besagt, dass Komponenten (also insbesondere auch einzelne Klassen) Funktionalitäten kapseln, die von anderen Komponenten benutzt werden können, ohne dass die Realisierung dieser Funktionalitäten bekannt ist – jede Komponente verbirgt also die sie betreffenden Entwurfsentscheidungen vor allen anderen Komponenten [Parnas72]. Für eine korrekte Benutzung reicht es, zu wissen, was die Komponenten leisten und wie sie zu benutzen sind. Für Klassen bedeutet dies, dass Dienstnutzer nur die Schnittstelle der Dienstleister kennen müssen und dürfen.

Diese konsequente Umsetzung des Geheimnisprinzips erzwingt zusammen mit dem »Entwurf mit Verträgen« (vgl. Abschnitt 30.4) die korrekte Benutzung von Dienstleistern, reduziert so die Fehlerwahrscheinlichkeit und unterstützt eine arbeitsteilige Entwicklung. Weiterhin sorgt die Anwendung des Geheimnisprinzips dafür, dass Änderungen der Realisierung einer Komponente lokal bleiben und keine Auswirkungen auf andere Komponenten haben.

Die Sichtbarkeit von Attributen und Operationen (vgl. Abschnitt 10.1) steuert, wie rigoros das Geheimnisprinzip auf der Ebene einzelner Klassen angewendet wird.

Attribute werden im Entwurf grundsätzlich nur als privat verwendet. Falls für ein privates Attribut att beide Standardoperationen setzeAtt() und gibAtt() erlaubt sind, werden aus Platzgründen in Klassendiagrammen die Standardoperationen nicht aufgeführt und auch keine Sichtbarkeit für das betreffende Attribut angegeben. Es gilt folgende Regel:

KR1 Attribute werden im Entwurf immer als privat deklariert.

Beispiel 30–1 Abbildung 30–1 zeigt links in vollständiger Notation die privaten Attribute x und y der Klasse Punkt sowie deren Standardoperationen. Rechts ist dieselbe Klasse in abkürzender Notation ohne Angabe von Sichtbarkeiten und Standardoperationen aufgeführt.

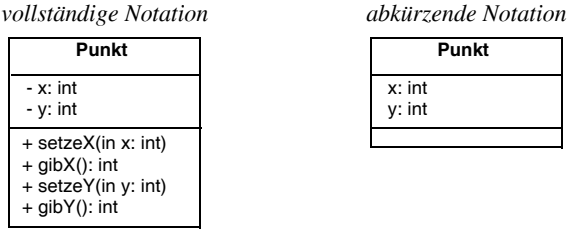

Abb. 30–1 *Notationen für private Attribute*

Auch innerhalb von Generalisierungshierarchien sollten Attribute nie als protected, sondern immer als private deklariert werden. Benötigen Unterklassen Zugriff auf diese Attribute, sieht man dafür entsprechende Standardoperationen vor und deklariert diese als protected. Um in Entwurfsdiagrammen zu einer kompakteren Darstellung zu kommen, können auch hier die Standardoperationen weggelassen werden. Hierbei stellt man dem Bezeichner eines solchen privaten Attributs ein # voran.

Beispiel 30–2 Abbildung 30–2 zeigt die Klasse Punkt mit den privaten Attributen x und y sowie ihren geschützten Standardoperationen links in vollständiger und rechts in abkürzender Notation.

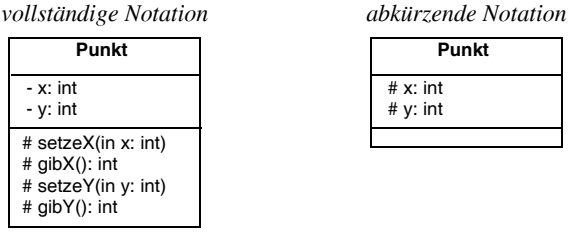

Abb. 30–2 *Notationen für private Attribute mit geschützten Standardoperationen*

Auch für *Operationen* legt man die Sichtbarkeit durch Angabe von public, protected oder private fest, wobei man darauf achtet, nur genau die Operationen einer Klasse als

public zu deklarieren, die auch tatsächlich von Operationen anderer Klassen aufgerufen werden. Soll eine Operation ggf. in Unterklassen benutzt oder sogar redefiniert, aber nicht von Operationen anderer Klassen aufgerufen werden, deklariert man sie als protected. Alle anderen Operationen werden als private deklariert.

In den folgenden Beispielen wird in den Klassendiagrammen verschiedentlich die Kennzeichnung der Sichtbarkeiten von Attributen oder Operationen weggelassen, wenn sie zur Erklärung der Sachverhalte nicht relevant sind. In der abschließenden Entwurfsspezifikation werden Sichtbarkeiten aber ausnahmslos angegeben. Vereinbart sei weiterhin, in Klassendiagrammen des Entwurfs immer mindestens den Typ der Attribute und Parameter anzugeben.

Auf der Ebene von Komponenten wird das Geheimnisprinzip durch die Trennung von Schnittstelle und Realisierung gewahrt (vgl. Abschnitt 10.9).

30.2 Schwache Kopplung und starke Kohäsion

Für die Beurteilung der Qualität eines Entwurfs spielt unter anderem die Komplexität der Beziehungen zwischen seinen Komponenten sowie die Komplexität der Komponenten selbst eine Rolle (vgl. z. B. [PagSix94]). Insbesondere ein objektorientierter Entwurf gewinnt an Transparenz und Wartbarkeit, wenn zwischen Klassen nur wenige, einfache Beziehungen existieren und jede Klasse genau eine wohldefinierte Aufgabe erfüllt.

Die Beziehungskomplexität in einem Entwurf bezeichnet man als *Kopplung*. Zwei Klassen A und B sind beispielsweise gekoppelt, wenn die Instanzen der Klasse A als Dienstnutzer von Instanzen der Klasse B (und damit Instanzen von B als Dienstleister für Instanzen von A) auftreten oder umgekehrt. Typischerweise werden vom Dienstnutzer Nachrichten an den Dienstleister über Verbindungen gesendet, die als Assoziationen zwischen den Klassen modelliert sind.

Allgemein formuliert sind zwei nichtprimitive Typen[1] gekoppelt, wenn

- es eine Assoziation zwischen den Typen gibt,
- ein Typ als Parametertyp in einer Operation des anderen Typs auftritt,
- Instanzen einer Klasse in einer Operation einer anderen Klasse erzeugt werden,
- zwischen den Typen eine Generalisierungsbeziehung existiert oder
- ein Typ ein Interface ist, das von dem anderen Typ (der eine Klasse sein muss) implementiert wird (vgl. Abschnitt 10.5).

Beispiel 30–3 Abbildung 30–3 zeigt einen Ausschnitt aus dem Klassendiagramm des Fallbeispiels SEMINARIS, der die Klassen Person und Adresse und die Aggregation Wohnort enthält. Zur Realisierung der Operation zusendenWerbematerial() verwenden die Instanzen der Klasse Person (als Dienstnutzer) die Operation erstelleAdressaufkleber() der mit ihnen verbundenen (dienstleistenden) Adresse-Instanzen (Abb. 30–4). Die

1. Nichtprimitive Typen (vgl. Abschnitt 10.5) werden durch (Schnittstellen von) Klassen oder Interfaces (vgl. Abschnitt 10.4 und 30.5) definiert.

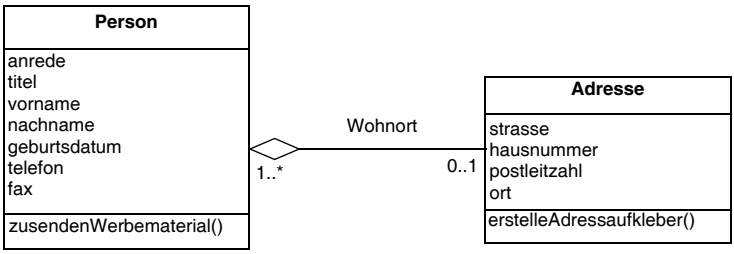

Abb. 30–3 *Aggregation zwischen den Klassen* Person *und* Adresse

beiden Klassen Person und Adresse sind über die Aggregation Wohnort sowie die Operation erstelleAdressaufkleber() gekoppelt.

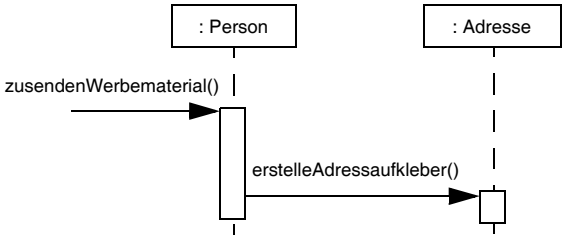

Abb. 30–4 *Dienstnutzung zwischen den Klassen* Person *und* Adresse

Die Komplexität der Klassen selbst ist für die Qualität eines Entwurfs genauso wichtig wie die Beziehungskomplexität. Die Größe einer Klasse ist hierbei nur ein Teilaspekt, wichtiger ist ein starker logischer Zusammenhang der von einer Klasse realisierten Aufgaben, also eine starke *Kohäsion*.

Beispiel 30–4 Im Anwendungssystem SEMINARIS besäße eine Klasse, die sowohl Teilnehmer verwaltet als auch die Entwicklung der Teilnehmerzahlen für Seminare aufzeichnet, keine starke Kohäsion und sollte in jedem Fall zerlegt werden.

Die Kopplung ist ein Maß für die Komplexität der Beziehungen zwischen den Klassen, die Kohäsion ist ein Maß für die Komplexität der einzelnen Klassen. Ein guter, d.h. die Qualitätskriterien aus Abschnitt 2.1 in hohem Maße erfüllender Entwurf zeichnet sich durch *starke Kohäsion und schwache Kopplung* aus.

Man macht sich schnell klar, dass beide Maße miteinander konkurrieren. In der objektorientierten Softwareentwicklung bedeutet dies, dass mit zunehmender Klassenzahl, also abnehmender Klassengröße, die Kohäsion steigt, weil dann jede Klasse nur eine eng umrissene Aufgabe erfüllt. Andererseits nimmt aber auch die Kopplung zu. Der Umfang der Schnittstellen wächst, weil Operationen, die bei größeren Klassen nur innerhalb dieser bekannt waren, jetzt zwecks Weiterverarbeitung über die Schnittstellen auch anderen Klassen bekannt gemacht werden müssen.

Für die Praxis können aus diesen Überlegungen zwei Schlussfolgerungen gezogen werden:

1. Zerlegt ein Entwurf ein Softwaresystem in zu wenige Klassen, so ist oft der logische Zusammenhang innerhalb der Klassen schwach. Die einzelnen Klassen sind groß und unübersichtlich, was wiederum negative Auswirkungen insbesondere auf die Wartbarkeit und Wiederverwendbarkeit hat.
2. Zerlegt ein Entwurf ein Softwaresystem in zu viele Klassen, so wird das Beziehungsgeflecht aufgebläht. Dabei werden Details, die eigentlich innerhalb von Klassen verborgen bleiben könnten und sollten, über Schnittstellen nach außen gegeben und damit letztlich auch das Geheimnisprinzip verletzt.

Gute Entwürfe befolgen die *Entwurfsprinzipien der schwachen Kopplung und starken Kohäsion*. Im Folgenden werden daher Heuristiken und Merkregeln vorgestellt, die dazu beitragen, die Kopplung abzuschwächen und die Kohäsion zu verstärken. Da Kopplung und Kohäsion konkurrierende Kriterien sind, sollten die Heuristiken stets so angewendet werden, dass eine Abschwächung der Kopplung nicht eine unangemessene Abschwächung der Kohäsion nach sich zieht und umgekehrt eine Verstärkung der Kohäsion nicht in einer zu starken Kopplung resultiert.

Abschwächung der Kopplung

Das grundsätzliche Ziel besteht darin, dass die Instanzen einer jeden Klasse so wenig wie möglich mit Instanzen anderer Klassen zusammenarbeiten, d.h. Verbindungen besitzen bzw. eingehen oder Nachrichten senden. Unterstützt wird diese Maxime durch schmale Schnittstellen, d.h., die Anzahl der öffentlichen (Attribute und) Operationen einer Klasse sollte so klein sein, dass gerade noch die Dienstnutzer befriedigt werden und so viel Interna wie möglich verborgen bleiben.

Die erste Heuristik zur Abschwächung der Kopplung ist sehr einfach und besagt:

H1 Sind nur sehr wenige Klassen (Anhaltspunkt: bis zu drei) eng gekoppelt, versucht man, die Klassen zu einer einzigen Klasse zusammenzufassen.

Eine andere Möglichkeit, eine starke Kopplung (mehrerer) Klassen aufzubrechen, besteht in der Einführung einer vermittelnden Klasse (vgl. das *Vermittler*-Entwurfsmuster in Kapitel 31), deren Instanz sämtliche Dienstnutzungen zwischen den Instanzen der stark gekoppelten Klassen abwickelt.

Beispiel 30–5 Abbildung 30–5 zeigt ein Klassendiagramm, in dem eine starke Kopplung der Klassen B, C, D und E zu erkennen ist.Die Kopplung lässt sich mit einer Klasse Vermittler abschwächen, deren Instanz die Dienstleistungen der Instanzen der Klassen D und E gegenüber den Instanzen der Klassen B und C übernimmt. Tatsächlich reicht die Vermittlerklasse die Funktionalitäten der Klasse D und E nur durch. Abbildung 30–6 zeigt die Verwendung der Vermittlerklasse. B und C sind jeweils nicht mehr mit D und E, sondern nur noch mit Vermittler gekoppelt. Lediglich Vermittler kennt die Aufteilung der Beziehungen auf D und E, aber nicht mehr B und C, die nur noch Vermittler kennen. Wird die Schnittstelle von D oder E geändert, wirkt sich die Änderung nur auf die Klasse Vermittler, aber nicht auf B und C aus.

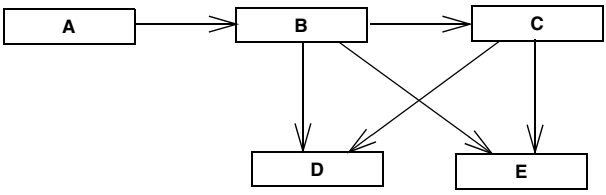

Abb. 30–5 *Stark gekoppelte Klassen* B, C, D *und* E

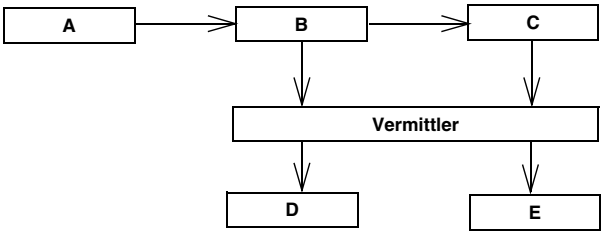

Abb. 30–6 *Abschwächung der Kopplung durch Einfügen einer Vermittlerklasse*

Die Erkenntnisse aus diesem Beispiel lassen sich in der folgenden Heuristik zusammenfassen:

H2 Sind mehrere Klassen (mehr als drei) stark gekoppelt, versucht man, die Kopplung durch eine Vermittlerklasse abzuschwächen.

Eine dritte Variante zur Kopplungsabschwächung besteht darin, die Funktionalität, welche die Kopplung verursacht, in eine andere Klasse zu verschieben.

Beispiel 30–6 Im Fallbeispiel »Bestellwesen« enthalte eine (neue) Klasse Bestellsystem eine Operation wertOffenerBestellungen(), die den Gesamtwert der momentan offenen Bestellungen ermittelt. Diese Operation verschafft sich hierzu für jeden Kunden über die Standardoperation gibBestellungen() zunächst alle seine Bestellungen. Den Bestellwert seiner einzelnen Bestellungen ermittelt die Operation wertOffenerBestellun-

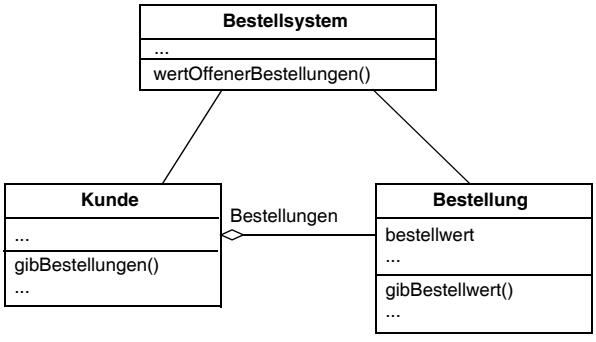

Abb. 30–7 *Bestellwesen: Starke Kopplung dreier Klassen*

gen() dann über die Standardoperation gibBestellwert() der Bestellungsinstanzen. Bei dieser Lösung ist die Klasse Bestellsystem sowohl mit der Klasse Kunde als auch mit Klasse Bestellung gekoppelt.

Definiert man stattdessen in der Klasse Kunde eine Operation wertAllerBestellungen(), die von der Operation wertOffenerBestellungen() verwendet wird, so ist die Klasse Bestellsystem nur noch mit der Klasse Kunde gekoppelt (Abb. 30–8).

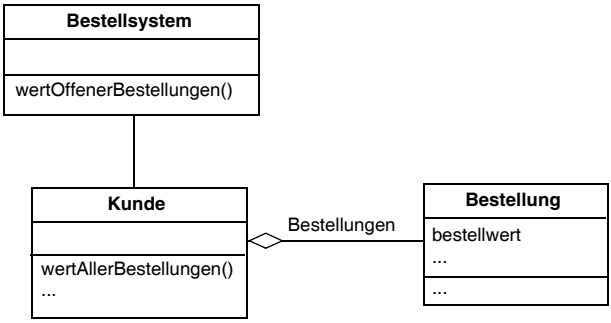

Abb. 30–8 *Bestellwesen: Abgeschwächte Kopplung der drei Klassen*

Beispiel 30–6 motiviert die folgende Heuristik:

H3 Ist eine Funktionalität für die starke Kopplung verantwortlich, versucht man, sie in eine andere Klasse zu verschieben.

Innerhalb von Generalisierungshierarchien ist die Benutzung vieler Standardoperationen der Oberklasse in einer Operation der Unterklasse häufig ein Zeichen für unnötige Kopplung.

Beispiel 30–7 Im Beispiel 10–1 (Seite 107) benutzt die Operation speichern() der Unterklasse Firmenkunde die zwei Standardoperationen gibName() und gibAdresse() der Oberklasse Kunde. Änderungen wie z. B. Hinzufügen oder Streichen eines Attributs der Oberklasse Kunde wirken sich unmittelbar auf die Unterklasse aus, da die entsprechenden Zugriffe zu ändern sind. Würde die Unterklasse Firmenkunde die Attribute name und adresse der Oberklasse Kunde jedoch mit der Operation speichern() von Kunde speichern, fielen die beiden Zugriffe weg. Man ersetzt daher die ersten zwei Anweisungen der Operation speichern() in der Klasse Firmenkunde durch einen Aufruf der in der Oberklasse definierten Operation speichern() (in Java: super.speichern(), vgl. Abschnitt 32.3). Abbildung 30–9 zeigt das resultierende Klassendiagramm.

Die Standardoperationen für die Attribute name und adresse sind jetzt in der Schnittstelle der Klasse Kunde nicht mehr erforderlich, wodurch die Kopplung zwischen den Klassen Firmenkunde und Kunde abgeschwächt wird. Wird nun die Oberklasse Kunde z. B. um weitere Attribute ergänzt, muss lediglich die Methode speichern() der Oberklasse Kunde, aber nicht die Methode speichern() der Unterklasse Firmenkunde geändert werden.

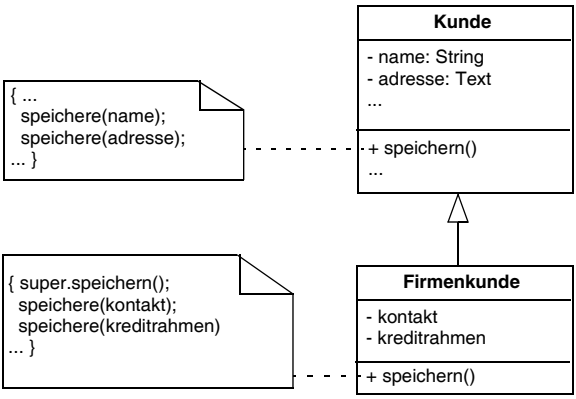

Abb. 30–9 *Die Unterklassenmethode speichern() benutzt die Oberklassenmethode*

Es ergibt sich die folgende Heuristik:

H4 Sind innerhalb einer Generalisierungshierarchie mehrere Standardoperationen der Oberklasse für die starke Kopplung verantwortlich, versucht man, sie durch den Aufruf einer (komplexeren) Operation der Oberklasse zu vermeiden.

Die folgenden zwei Merkregeln beschäftigen sich mit Kopplungsproblemen im Zusammenhang mit Aggregation und Komposition.

KR2 Jede »Ganzes-Klasse« muss wissen, welche Teil-Klassen sie enthält, aber keine Teil-Klasse sollte ihre Ganzes-Klasse(n) kennen.

KR3 Teil-Klassen derselben Ganzes-Klasse sollten sich nicht gegenseitig benutzen (»Untergebene flüstern nicht in Gegenwart ihrer Vorgesetzten«).

Eine Benutzung der Teil-Klassen untereinander kann aufgehoben werden, indem die Instanz der Ganzes-Klasse ihre Teil-Instanzen in geeigneter Weise koordiniert.

Den Abschluss zum Thema »Abschwächen der Kopplung« bildet eine Heuristik, die als *Law of Demeter* [LieHol89] bekannt ist und die »erlaubten« Dienstleister auf der Ebene von Operationsaufrufen eingrenzt. Diese Heuristik verhindert insbesondere, dass ein Objekt über mehrere Objektverbindungen hinweg Operationen von Klassen aufruft, die nicht direkt mit der Klasse des ausführenden Objekts assoziiert sind.

H5 In einer Operation `o()` einer Klasse `K` sollten nur Operationen folgender Klassen benutzt werden (Law of Demeter):

- `K` selbst,
- Klassen, die als Parametertypen von `o()` vorkommen,
- mit `K` assoziierte Klassen und
- Klassen, die bei der Ausführung von `o()` instanziiert werden.

Die konsequente Anwendung des »Law of Demeter« macht es oft erforderlich, dass eine Klasse in ihrer Schnittstelle auch einige oder alle Operationen der mit ihr assozi-

ierten Klassen anbietet, diese dann per Delegation benutzt und somit eine Art Vermittlerrolle spielt (vgl. Heuristik H2).

Verstärkung der Kohäsion

Eine zu schwache Kohäsion einer Komponente kann man grundsätzlich durch Zerlegung der Komponente in kleinere beheben. Oben wurde aber bereits darauf hingewiesen, dass hierbei eine zu starke Kopplung der resultierenden Komponenten untereinander und mit anderen Komponenten zu vermeiden ist. Die folgenden Heuristiken betreffen Situationen, in denen die Kohäsion verstärkt werden kann, ohne dass die Kopplung über Gebühr strapaziert wird.

Woran erkennt man nun die zu schwache Kohäsion einer Klasse? Die erste Heuristik dieses Abschnitts liefert eine einfache Antwort:

H6 Lassen sich die Attribute und die Operationen einer Klasse in (weitgehend) disjunkte Teilmengen zerlegen, so dass jede Operationsmenge nur auf jeweils einer Attributmenge arbeitet, dann ist die Aufteilung der Klasse zu empfehlen.

Die Anwendung der Heuristik ist umso sinnvoller, je kleiner die jeweiligen Durchschnitte der Attribut- und Operationsmengen sind, da die durch die Kohäsion hervorgerufene Kopplung schwach ausfällt. Lassen sich keine weitgehend disjunkten Teilmengen der Attribute und Operationen finden, liegt ein starkes Indiz dafür vor, dass die Kohäsion der Klasse stark genug ist.

Kohäsion ist ein Maß, das nicht nur auf Klassen (oder größere Komponenten), sondern auch auf Operationen anzuwenden ist. Eine Operation besitzt eine starke Kohäsion, wenn sie eine einzige, eng umrissene und klar abgegrenzte Funktionalität erbringt. Das folgende Beispiel beinhaltet eine Operation mit schwacher Kohäsion.

Beispiel 30–8 Eine Klasse Binaerbaum enthalte eine Operation durchlaufeBaum() mit der Signatur

```
durchlaufeBaum (in modus: Integer): Sequence(Object).
```

Hierbei gibt der Werteparameter modus an, welcher Art der Durchlauf sein soll (1 = preorder, 2 = inorder, 3 = postorder). Die Operation hat eine schwache Kohäsion, da sie eigentlich drei unterschiedliche Operationen realisiert:

```
durchlaufeBaumPreorder(): Sequence(Object),
durchlaufeBaumInorder(): Sequence(Object) und
durchlaufeBaumPostorder(): Sequence(Object)
```

Diesen Sachverhalt gibt die folgende Heuristik wieder:

H7 Erbringt eine Operation mehr als eine einzige eng umrissene Funktionalität, ist die Funktionalität auf mehrere Operationen aufzuteilen.

30.3 Generalisierung und Konformität

Das Konzept der Generalisierung soll jetzt im Kontext des Entwurfs präzisiert werden. Wenn im Weiteren einfach von Klassen die Rede ist, sind damit sowohl reguläre und abstrakte Klassen als auch Interfaces gemeint.

Bei der Generalisierung erbt eine *Unterklasse* B alle Attribute und Operationen ihrer *Oberklasse* A, einschließlich der Implementierung der Operationen (Methoden). Die Klasse B kann die Implementierung einer oder mehrerer geerbter Operationen auch durch eine eigene Implementierung mit der gleichen Semantik ersetzen (*überschreiben, overriding*). Dabei wird die Einhaltung des Substituierbarkeitsprinzips gefordert, das besagt, dass Unterklasseninstanzen die Instanzen ihrer Oberklasse vertreten können. Um Substituierbarkeit zu ermöglichen, unterstützen objektorientierte Programmiersprachen die Konzepte Polymorphismus und dynamisches Binden. Diese eher (Programmiersprachen-)technischen Aspekte werden kurz in Abschnitt 32.3 rekapituliert.

Damit die Instanzen der Unterklasse die Instanzen ihrer Oberklasse auch semantiktreu vertreten können, ist es erforderlich, dass (die Spezifikationen von) Unterklassen zu (den Spezifikationen) ihrer Oberklasse *konform* sind. Dieser Begriff wird nun zunächst informell skizziert, dann für einzelne Operationen definiert und letztendlich auf Klassen in Generalisierungsbeziehungen erweitert. Im Mittelpunkt stehen dabei die in Abschnitt 11.3 eingeführten Spezifikationselemente Klasseninvariante und Vor- und Nachbedingungen von Operationen.

Vor- und Nachbedingung einer Operation ordnen der aktuellen Objektkonstellation (aktuelle Werte der Eigenschaften und Verbindungen aller Objekte des Anwendungssystems)[1] ein boolesches Ergebnis (true oder false) zu. Die Vorbedingung einer Operation beschreibt die Menge aller Objektkonstellationen, bei denen die Operation aufgerufen werden darf. Die Nachbedingung beschreibt die Objektkonstellation, die nach Ausführung der Operation erreicht sein muss. Die Implementierung der Operation muss garantieren, dass nach ihrer Ausführung die Nachbedingung erfüllt ist, wenn vor ihrer Ausführung die Vorbedingung erfüllt war.

Die Objekte einer Klasse besitzen häufig Eigenschaften, die vor und nach jeder Ausführung einer öffentlichen Operation gelten müssen. Um die Bedingung, die diese Eigenschaften sicherstellt, nicht in jeder Vor- und Nachbedingung angeben zu müssen, formuliert man sie als *Klasseninvariante*. Die Invariante ist somit Bestandteil der Vor- und Nachbedingung jeder öffentlichen Operation. Vor und nach der Durchführung einer privaten Operation muss die Klasseninvariante nicht gelten.

Während sich die Vor- und die Nachbedingung einer Operation sowohl auf den Zustand des ausführenden Objekts (Attributwerte und Verbindungen, vgl. Kapitel 15) als auch auf die Parameter der Operation beziehen, ist für die Klasseninvariante nur der Zustand erlaubt.

Informell soll das Konzept der Konformität zunächst an den in Abbildung 30–10 gezeigten Klassen skizziert werden. Damit Instanzen von B überall dort eingesetzt werden können, wo Instanzen von A auftreten können, darf die Vorbedingung der Operation op() in B höchstens das fordern, was bezüglich dieser Operation auch in A gefordert

1. Zur Laufzeit entspricht dies dem aktuellen Programmzustand der Anwendung.

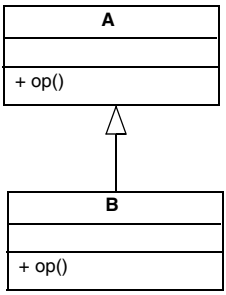

Abb. 30–10 *Operation* op() *in Oberklasse* A *und Unterklasse* B

wird. Ihre Nachbedingung darf in B lediglich so verändert werden, dass die Operation in B mehr erbringt wie diejenige in A. Natürlich müssen alle B-Instanzen mindestens die Invariante von A erfüllen, dürfen aber auch weitere Bedingungen garantieren. Dies wird im Weiteren präzisiert.

Für eine Klasse A mit Unterklasse B bezeichnen op_A bzw. op_B die in A definierte bzw. in B überschriebene Operation op() (siehe Abb. 30–10). Die (Spezifikation der) Operation op_B aus B heißt *konform* zur (Spezifikation der) in der Klasse A definierten Operation op_A, wenn die folgenden drei *Konformitätsbedingungen* gelten:

(1) op_B und op_A haben die gleiche Signatur (d.h. op_B überschreibt op_A)[1],

(2) Vorbedingung(op_A) $\Rightarrow$ Vorbedingung(op_B)[2] und

(3) Nachbedingung(op_B) $\Rightarrow$ Nachbedingung(op_A).

Im Weiteren bezeichnen INV_A bzw. INV_B die Invariante der Klasse A bzw. B. Sind alle von A geerbten Operationen in B konform zu den entsprechenden Operationen in A und gilt darüber hinaus $INV_B \Rightarrow INV_A$, so heißt die (Spezifikation der) Unterklasse B konform zur (Spezifikation der) Oberklasse A. Abbildung 30–11 stellt den Sachverhalt grafisch dar.

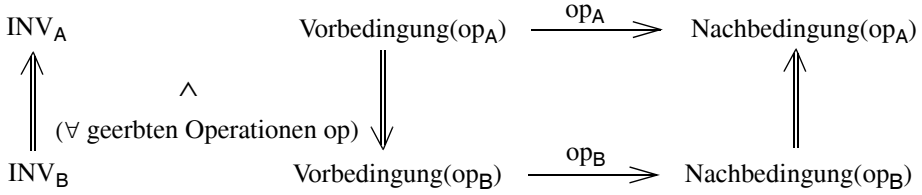

Abb. 30–11 *Konformitätsbedingungen für Unterklasse* B *bezüglich Oberklasse* A

1. Genau genommen bräuchten die in- und inout-Parameter der Signatur von op_B nur *kontravariant* sein (also gleiche oder allgemeinere Typen wie in op_A haben) und ihre out-Parameter *kovariant* (gleiche oder speziellere Typen wie in op_A). Die meisten objektorientierten Programmiersprachen (und insbesondere auch Java bis Version 1.4) erlauben jedoch bei überschreibenden Operationen der Unterklassen ausschließlich identische Signaturen (andernfalls ist op_B unabhängig von op_A und man spricht vom *Überladen* (*overloading*) der Operation op_A durch die Operation op_B). Java erlaubt ab Version 1.5 auch lediglich kovariante Rückgabewerte. Daher werden hier ausschließlich identische Signaturen der überschriebenen und der überschreibenden Operation betrachtet.
2. Das Symbol $\Rightarrow$ steht für die logische Implikation.

Die Definition der Konformität präzisiert das Prinzip der Substituierbarkeit: Ist Klasse B konform zur Klasse A, dann können Instanzen von A durch Instanzen von B ersetzt werden. Damit lässt sich jetzt die Definition der Generalisierung präzisieren:

- Eine Ober-Unterklassenbeziehung ist eine *Generalisierung*, wenn die Unterklasse konform zur Oberklasse ist.

Für die Spezifikation von in der Unterklasse zusätzlich definierten Operationen besteht keine Einschränkung.

Ist die Unterklasse zu ihrer Oberklasse nicht konform, spricht man anstelle von Generalisierung von *Vererbung*. Im Kontext von Programmiersprachen wird meistens der Begriff Vererbung verwendet, da weder heute verfügbare Compiler noch Laufzeitsysteme die Konformität einer Vererbungsbeziehung überprüfen.

Generalisierung im Sinne der Konformität bietet gegenüber der allgemeineren Vererbung erhebliche Vorteile. So lässt sich beispielsweise das wichtige Konzept des Vertragsentwurfs (vgl. Abschnitt 30.4), das die Rechte und Pflichten von Dienstnutzern und Dienstleistern klar regelt, direkt auf Unterklassen der Dienstleister übertragen, sofern diese konform zu ihren Oberklassen sind.

Ein weiterer Vorteil ergibt sich für das Testen. Ist die Spezifikation einer Klasse B konform zu der Spezifikation einer Klasse A und sind die Implementierungen von A und B gegen ihre Spezifikationen erfolgreich getestet oder verifiziert, so brauchen Operationen, die als Parameter Objekte der Klasse A erwarten, nicht zusätzlich mit Objekten der Klasse B als Parameter getestet zu werden.

Verletzung der Konformität

Erfahrungsgemäß wird die Konformität einer Unterklasse zu ihrer Oberklasse oft verletzt, wenn die Implementierung einer Operation der Oberklasse verändert wird (um z. B. die speziellen Bedingungen der Unterklasse zu berücksichtigen). Dass die Konformität aber auch ohne Überschreibung einer Operation der Oberklasse verletzt werden kann, zeigt das nächste Beispiel.

Beispiel 30–9 Abbildung 30–12 zeigt die Klasse Angestellter mit einem Attribut Stellenbezeichnung, das die Werte »Programmierer«, »Administrator«, »Gruppenleiter« und »Abteilungsleiter« annehmen kann. Die Bedingung, dass nur diese vier Werte zulässig sind, ist Bestandteil der Klasseninvariante der Klasse Angestellter. Die Unterklasse Vorgesetzter definiert spezielle Angestellte mit Vorgesetztenaufgaben, z. B. beurteilungenSchreiben. Vorgesetzte erben das Attribut Stellenbezeichnung der Klasse Angestellter, jedoch sind Vorgesetzte immer Gruppenleiter oder Abteilungsleiter, aber niemals Programmierer oder Administrator. Diese Einschränkung ist Bestandteil der Klasseninvariante der Klasse Vorgesetzter.

Auf den ersten Blick sieht es so aus, als wäre die Klasse Vorgesetzter konform zur Klasse Angestellter. Vorgesetzte scheinen Angestellte ersetzen zu können, da keine geerbte Methode überschrieben wird und Vorgesetzenobjekte auch die Invariante für Angestelltenobjekte erfüllen. Probleme gibt es jedoch, wenn ein Vorgesetzter über einen Aufruf der Operation setzeStellenbezeichnung() z. B. zum Programmierer

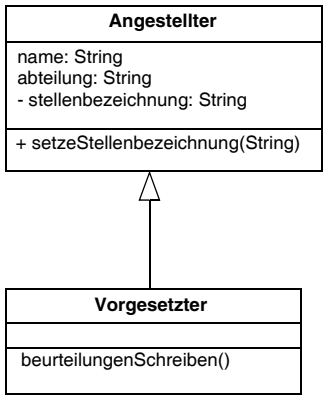

Abb. 30–12 *Vorgesetzte sind spezielle Angestellte*

gemacht wird, denn dann verlässt das Vorgesetztenobjekt den zulässigen Zustandsraum für Vorgesetzte und verletzt seine Klasseninvariante. Tatsächlich ist die Klasse Vorgesetzter zur Klasse Angestellter nicht konform, da wegen der stärkeren Invariante die Konformitätsbedingung (2) von Seite 341 für die Operation setzeStellenbezeichnung() nicht erfüllt ist.

Die Verletzung der Konformität hat die Konsequenz, dass Vorgesetzte nicht an Stellen verwendet werden können, an denen auf das Attribut Stellenbezeichnung von Angestellten schreibend zugegriffen wird. Bei der vorliegenden Vererbung handelt es sich somit nicht um eine Generalisierung. Der folgende Abschnitt zeigt eine konforme Lösung für diese Problemstellung.

Erweiterung der Generalisierungshierarchie

Sollen Generalisierungshierarchien durch neue Klassen ergänzt werden, wird häufig nur an die Bildung neuer Unterklassen gedacht. Dies ist aber keineswegs immer sinnvoll. Gelegentlich empfiehlt sich nämlich auch die Einführung neuer Oberklassen, beispielsweise wenn Unterklassen einige Eigenschaften der Oberklasse entweder gar nicht oder nicht konform anbieten können.

Beispiel 30–10 Die Klasse Vorgesetzter aus Beispiel 30–9 sollte keine Unterklasse der Klasse Angestellter sein, da die Operation setzeStellenbezeichnung() in Vorgesetzter nicht konform zur entsprechenden Operation in Angestellter ist. Besser ist die in Abbildung 30–13 dargestellte Definition einer neuen Oberklasse Firmenangehoeriger, von der die Klassen Angestellter und Vorgesetzter Unterklassen sind.

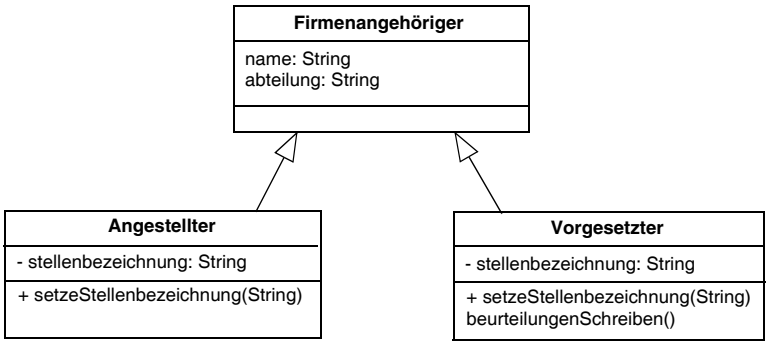

Abb. 30–13 Angestellter *und* Vorgesetzter *als »Geschwisterklassen«*

Es ergibt sich die folgende Heuristik:

H8 Ist eine Unterklasse nicht konform zu ihrer Oberklasse, weil die Oberklasse Eigenschaften hat, die für die Unterklasse nicht zutreffen, sollte eine neue gemeinsame Oberklasse gebildet werden, die in einer Generalisierungsbeziehung zu jeder der beiden ursprünglichen Klassen steht.

Minimierung des Überschreibens

Ziel bei der Bildung neuer Unterklassen ist es, möglichst wenige Operationen zu überschreiben, damit der Implementierungs- und Testaufwand schwach ist. Um dieses Ziel zu erreichen, sollte schon bei der Definition der Oberklasse an später zu definierende Unterklassen gedacht werden.

Beispiel 30–11 Gegeben ist die Klasse LinkedList mit Operationen zum Einfügen und Löschen von Elementen und zum Durchsuchen der Liste (Abb. 30–14).

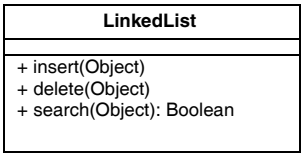

Abb. 30–14 *Die Klasse* LinkedList

Ein verketteter Ring unterscheidet sich von einer verketteten Liste nur dadurch, dass das letzte Element wieder einen Verweis auf das erste Element enthält. Definiert man eine Klasse LinkedRing als Unterklasse von LinkedList, so muss man alle drei Operationen vollständig neu implementieren, da Listendurchläufe in den geerbten Operationen nicht mehr terminieren. Hätte LinkedList dagegen eine protected-Operation atEnd(Object): boolean, welche die Abfrage auf das Listenende realisiert und von allen anderen Opera-

tionen verwendet wird, so wäre nur die Operation atEnd() in der Klasse LinkedRing neu zu implementieren.

Aus diesem Beispiel lässt sich folgender Schluss ziehen:

H9 Klassen werden so definiert, dass sie für Unterklassenbildung offen, aber für unkontrollierte Zugriffe geschlossen sind (*Open-Closed-Principle*). Um möglichst viele Operations-Implementierungen in Unterklassen wiederverwenden zu können, werden Operationen einer Oberklasse, die von Operationen der Klasse selbst und von (zukünftigen) Unterklassen benutzt werden, als protected gekennzeichnet.

30.4 Entwurf mit Verträgen

Dieser Abschnitt beleuchtet das Verhältnis zwischen Dienstnutzern und Dienstleister im Rahmen der Spezifikation der Operationen einer Schnittstelle. Dieses Verhältnis weist eine deutliche Analogie zu dem Verhältnis zwischen einem Auftraggeber und einem Auftragnehmer im Geschäftsleben auf. Aufträge sind im Normalfall vertraglich geregelt. Ein Vertrag enthält für jede Partei sowohl Auflagen (Pflichten) als auch einen Nutzen (Rechte). Der Auftraggeber beispielsweise erhält eine Ware oder Dienstleistung, wenn er seine Auflagen wie z. B. Zahlungsmodalität und Bereitstellen einer bestimmten Umgebung erfüllt.

Der Vertrag schützt einerseits den Auftragnehmer, indem er die Abnahmebedingungen und den Preis spezifiziert. Er schützt andererseits den Auftraggeber durch Festschreiben, was zum vereinbarten Preis zu erwarten ist. Zusätzlich sollte vertraglich geregelt sein, wie in Sonderfällen, z. B. bei Lieferverzögerungen, zu verfahren ist.

Auf dieser Idee basiert der von Bertrand Meyer vorgeschlagene *Entwurf mit Verträgen* (*design by contract*, vgl. [Meyer92], [Meyer97]), der die Zusammenarbeit von Klassen bzw. Objekten regelt. Dabei werden drei Techniken eingesetzt:

- Spezifikation (der Operationen) der Schnittstellen mit Vor- und Nachbedingungen und Klasseninvarianten.
- Klar geregelte Verantwortlichkeiten von Dienstnutzer und Dienstleister.
- Systematische Behandlung von Ausnahmefällen.

Bezüglich der Spezifikation der Schnittstelle mit Vor- und Nachbedingungen sowie der Klasseninvarianten muss jede Unterklasse (vertrags-)konform zu ihrer Oberklasse sein (vgl. Abschnitt 30.3), damit der Vertrag der Oberklasse auch für die Unterklasse gilt.

- Für die Verantwortlichkeiten von Dienstnutzer und Dienstleister gilt:
- Wenn der Dienstnutzer die Vorbedingung erfüllt, dann garantiert der Dienstleister die Nachbedingung.
- Dienstnutzer und Dienstleister erfüllen die Invariante ihrer Klasse vor und nach jeder Ausführung einer öffentlichen Operation.

Diese Verantwortlichkeiten sowie die Elemente, auf denen sich die einzelnen Vertragsanteile abstützen können, sind in der Tabelle 30–1 zusammengefasst. Darüber hinaus

werden alle die betrachteten Klassen betreffenden Einschränkungen des Klassenmodells wie z.B. Multiplizitäten der Assoziationsenden zu den Invarianten gezählt, denn auch diese gelten vor und nach jeder Ausführung einer öffentlichen Operation.

	Dienstnutzer	Dienstleister	Verwendbare Elemente
Vorbedingung	Pflicht	Recht	in- und inout-Parameter Attributwerte Verbindungen
Nachbedingung	Recht	Pflicht	in-, inout- und out-Parameter Attributwerte Verbindungen (Anfangswerte mit @pre)
Invariante	Pflicht	Pflicht	Attributwerte Verbindungen

Tab. 30–1 *Verantwortlichkeiten und Elemente der Vertragsgestaltung*

Das strikte Einhalten der Verträge ist zentrale Voraussetzung für die korrekte Funktion des Anwendungssystems, denn Vor- und Nachbedingungen werden nicht gesondert in den Implementierungen der Operationen geprüft, sondern als gültig vorausgesetzt. Als Konsequenz führt eine Vertragsverletzung in den meisten Fällen zu einem Programmfehler. Das Vertragskonzept erleichtert dann die Lokalisierung der Fehlerursache:

- Eine verletzte Vorbedingung weist auf ein Fehlverhalten des Dienstnutzers hin.
- Eine verletzte Nachbedingung weist auf ein Fehlverhalten des Dienstleisters hin.

Verträge werden nicht dazu mißbraucht, um z.B. das Verhalten des Dienstleisters bei speziellen Parameterbelegungen zu definieren. Bei insgesamt erfüllter Vorbedingung müssen alle möglichen Fälle innerhalb der Methode z.B. mit Fallanweisungen behandelt werden. Es gilt die folgende Regel:

KR4 Vor- bzw. Nachbedingungen beinhalten keine vorhersehbaren Ausnahmefälle, die immer gesonderter Behandlung bedürfen.

Schließlich muss noch vertraglich festgelegt werden, was zu tun ist, wenn der Dienstleister seine Aufgabe nicht erfüllen kann, obwohl der Dienstnutzer die Vorbedingung erfüllt hat. Das kann vorkommen, wenn z.B. ein Hardwarefehler vorliegt oder ein vom Dienstleister selbst benutzter Dienstleister vertragsbrüchig wird. Tritt eine solche Ausnahme (exception) auf, so wird nach folgender Strategie verfahren:

1. Gibt es einen alternativen Lösungsweg, z.B. die Benutzung eines anderen Geräts oder Dienstleisters, kann zunächst dieser versucht werden (*Wiederanlauf*).
2. Sind alle Alternativen ausprobiert, wird die Operation endgültig abgebrochen, was zu einer Ausnahme für den Dienstnutzer führt: *organisierte Panik*. In diesem Fall sind folgende Optionen denkbar (vgl. [Meyer97], [Siedersleben04]):
   - Protokollieren und weitermachen, wenn die Ausnahme für den Dienstnutzer unerheblich ist.

- Protokollieren und den Schaden begrenzen, indem z. B. Transaktionen zurückgesetzt oder offene Dateien geschlossen werden, wenn die betroffenen Teile des Anwendungssystems beendet werden müssen.
- Im Rahmen der Ausnahmebehandlung als Dienstnutzer abwarten und die Operation wiederholt aufrufen, wenn z. B. Timeouts bei Netzwerkverbindungen oder Datenbankzugriffen die Ursache für eine Ausnahme sind.
- Rekonfiguration des gesamten Systems, wenn alternative Komponenten mit den gleichen Diensten zur Verfügung stehen.

Zum Abschluss hier noch das »Standardbeispiel« für die Vertragsgestaltung.

Beispiel 30–12 Ein Stapel (Klasse Stack) soll mittels Vertragsbedingungen spezifiziert werden, die in der Object Constraint Language (OCL) zu notieren sind (vgl. Abschnitt 11.3). Einen Ausschnitt aus der Spezifikation zeigt Abbildung 30–15. Die durch das Schlüsselwort inv gekennzeichnete Klasseninvariante sagt aus, dass die Anzahl der gestapelten Elemente höchstens gleich dem Wert des Attributs MAXSIZE und größer gleich 0 ist. Die mit pre gekennzeichnete Vorbedingung der Operation top() schränkt

```
context Stack
   inv:  (self.size() >= 0)  AND  (self.size() <= self.MAXSIZE) ;
-- attributes
context Stack::MAXSIZE: Integer {constant};
context Stack::size: Integer init: 0;
-- queries
context Stack::size (): Integer {isQuery}
            -- Anzahl gestapelter Elemente zurückgeben
            pre: true;
            post: return >= 0;
context Stack::top (): Object {isQuery}
            -- Oberstes Element zurückgeben
            pre: (self.size() > 0;
            post: return != null AND self.equals( self@pre);
context Stack::all (): Collection {isQuery}
            -- Kollektion mit allen gestapelten Elementen erzeugen
            pre: true
            post: self.equals( self@pre) AND // Achtung: self@pre ist nicht OCL-konform!
                  (self.size() > 0 implies return.size() = self.size()) AND
                  (self.size() = 0 implies return = null)
-- modifier
context Stack::push (item:object)
            -- Element stapeln
            pre: self.size() < self.MAXSIZE();
            post: self.size() = self.size()@pre+1 AND self.top() = item@pre;
context Stack::pop ()
            -- Element entstapeln
               pre: self.size() > 0
               post: self.size() = self.size()@pre - 1
      ...
```

Abb. 30–15 *Klassenspezifikation mit Vertragsbedingungen*

deren Anwendbarkeit auf nichtleere Stapelobjekte ein, die mit post gekennzeichnete Nachbedingung sichert die Rückgabe einer gültigen Objektreferenz zu. Die Nachbedingung der Operation push() sichert zu, dass der Stapel nach ihrer erfolgreichen Beendigung um ein Element angewachsen ist und dass das als Parameter beim Aufruf der Operation übergebene Element zuoberst gestapelt wurde.

Zu beachten ist, das der Ausdruck self@pre in der Nachbedingung der Operation all() nicht OCL-konform ist. Er steht als Abkürzung dafür, dass der Zustand des Objekts bei der Ausführung dieser Operation nicht verändert wird. Bei einer OCL-konformen Spezifikation müssten alle Attribute einzeln aufgeführt werden.

30.5 Interfaces

In vielen Anwendungen können Objekte unterschiedliche Rollen spielen, die im Klassenmodell an den Enden von Assoziationen zwischen den entsprechenden Klassen notiert werden (vgl. Abschnitt 9.4). Abhängig von seiner gerade gespielten Rolle geht ein Objekt bestimmte Verbindungen mit anderen Objekten ein. Oft bedeutet das Spielen einer bestimmten Rolle, dass die verbundenen Objekte nur eine Teilmenge von Operationen und Attributen des die Rolle spielenden Objekts benötigen.

Beispiel 30–13 In dem Getränkeautomaten-Beispiel aus Kapitel 8 nimmt der Getränkeautomat einem Kunden gegenüber die Rolle eines Getränkelieferanten und dem Wartungspersonal gegenüber die Rolle eines Wartungsobjekts ein (vgl. Abb. 30–16).

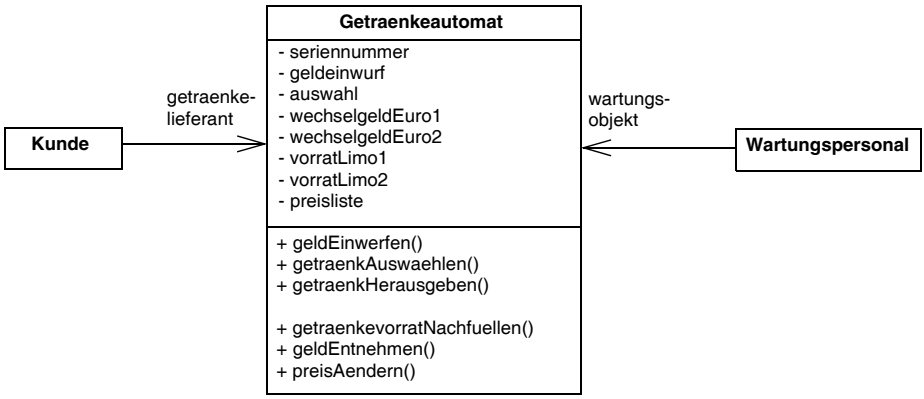

Abb. 30–16 *Die beiden Rollen des Getränkeautomaten*

Zur Getränkelieferantenrolle gehören die ersten drei Operationen (die Kundenschnittstelle), zur Wartungsobjektrolle die letzten drei (die Wartungsschnittstelle). Unter dem Gesichtspunkt des Geheimnisprinzips wäre es wünschenswert, die Wartungsschnittstelle vor den Kunden und die Kundenschnittstelle vor dem Wartungspersonal zu verbergen.

Eine nahe liegende Möglichkeit wäre, die Klasse Getraenkeautomat in zwei Klassen aufzuteilen, von denen die eine die Kunden- und die andere die Wartungsschnittstelle anbietet. Leider wird aber von beiden Schnittstellen aus auf die gleichen Daten zugegriffen: z. B. greifen die Operationen geldEinwerfen() und geldEntnehmen() auf die Attribute wechselgeldEuro1 und wechselgeldEuro2 zu.

Ebenso wie die UML (vgl. Abschnitt 10.4) bieten auch einige Programmiersprachen (z. B. Java, C#) zur Lösung dieses Problems das Konzept des Interface an. Ein *Interface* definiert eine Menge von öffentlich sichtbaren Operationen. Im Unterschied zu Klassen definiert ein Interface weder Attribute noch Implementierungen von Operationen. Daher können von ihm – ebenso wie von abstrakten Klassen – keine Instanzen gebildet werden. Ein Interface definiert jedoch einen Typ, von dem Variablen deklariert werden können, und außerdem die Verträge, die von den Klassen, die das Interface realisieren, und ihren Dienstnutzern einzuhalten sind. Eine Klasse *realisiert* ein Interface, indem sie für jede seiner Operationen eine (konforme) Implementierung zur Verfügung stellt.

Beispiel 30–14 In dem obigen Fall definiert man zwei Interfaces GetraenkeautomatKundenschnittstelle und GetraenkeautomatWartungsschnittstelle, welche die Operationen der Rollen getraenkelieferant bzw. wartungsobjekt zusammenfassen. Ein Kunde beispielsweise greift nun nicht mehr direkt auf den Getränkeautomaten, sondern auf das Interface GetraenkeautomatKundenschnittstelle zu. Analog benutzt das Wartungspersonal das Interface GetraenkeautomatWartungsschnittsstelle mit den Operationen der Rolle wartungsobjekt. Die Klasse Getraenkeautomat enthält die Implementierungen der Operationen für beide Interfaces.

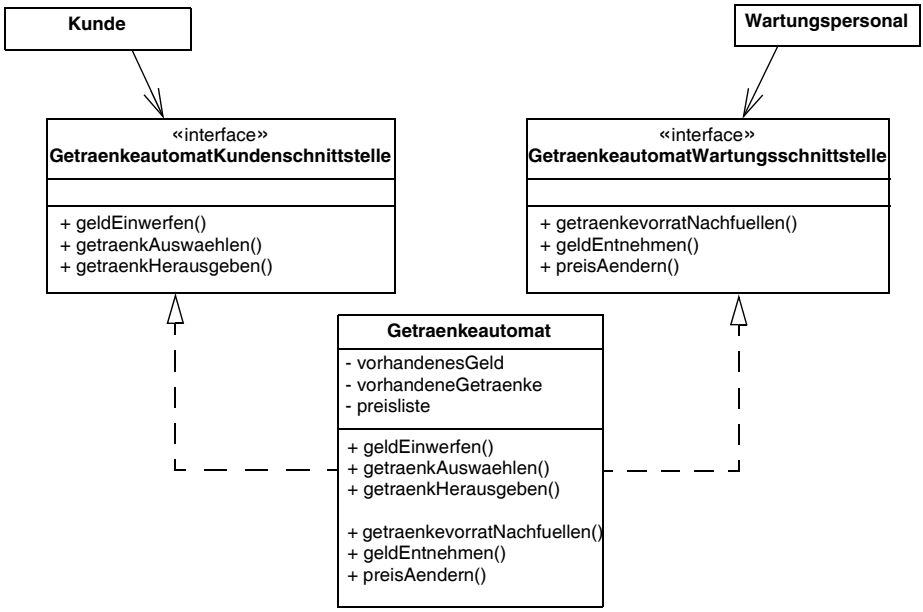

Abb. 30–17 *Schnittstellen als Interfaces realisiert*

Abbildung 30–17 zeigt das zugehörige Klassendiagramm. Die gestrichelten Generalisierungspfeile stellen die *Realisierungsbeziehungen* dar. Technisch gesehen hat die Klasse Kunde ein Attribut vom Typ GetraenkeautomatKundenschnittstelle, das eine Instanz der Klasse Getraenkeautomat enthält. Gemäß der Typdefinition dieses Attributs können auf dieses Objekt nur Operationen des Interfaces GetraenkeautomatKundenschnittstelle und nicht die anderen Operationen der Klasse Getraenkeautomat angewendet werden. Analoge Einschränkungen gelten für Zugriffe des Wartungspersonals über das Interface GetraenkeautomatWartungsschnittstelle.

Im Gegensatz zum Klassendiagramm in Abbildung 30–16 wird jetzt das Geheimnisprinzip viel besser gewahrt. Ein weiterer Vorteil der Verwendung von Interfaces besteht darin, dass sogar eine spätere Aufteilung der Klasse Getraenkeautomat in zwei Klassen, die jeweils eine Rolle implementieren, ohne Auswirkungen auf die Dienstnutzung durch Kunden und Wartungspersonal möglich ist.

Interfaces bieten außerdem eine elegante Möglichkeit, Instanzen verschiedener Klassen mit einer gemeinsamen Eigenschaft einheitlich zu verwenden.

Beispiel 30–15 Das Klassendiagramm in Abbildung 30–18 stellt einen Ausschnitt des Entwurfs für eine Druckerwarteschlange dar. Die Klasse Druckerwarteschlange verwaltet eine Warteschlange von Druckaufträgen, die sie der Reihe nach an den Drucker schickt. Die Klassen FormularDruckauftrag und BlankoDruckauftrag definieren zwei unterschiedliche Arten von Druckaufträgen. Wird eine der Operationen druckeFormularDruckauftrag() bzw. druckeBlankoDruckauftrag() der Klasse Druckerwarteschlange aufgerufen, wird die als Parameter übergebene Instanz der Klasse FormularDruckauftrag bzw. BlankoDruckauftrag in die Warteschlange aufgenommen und der entsprechende Druckauftrag ausgeführt, sobald der Drucker frei ist. Die Klassen FormularDruckauftrag und BlankoDruckauftrag haben jeweils eine Operation gibDruckdaten(), welche die zu druckenden Daten liefert. Als Konsequenz muss die Druckerwarteschlange für jede der beiden Klassen eine spezielle Druckoperation bereitstellen.

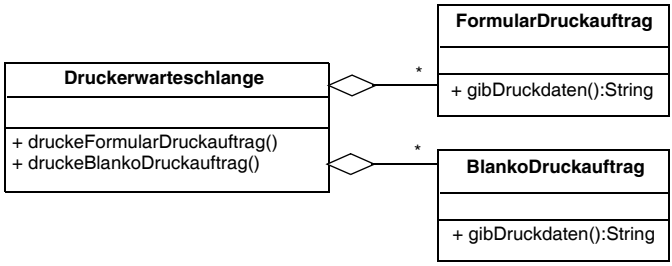

Abb. 30–18 *Die Druckerwarteschlange druckt verschiedene Arten von Druckaufträgen*

Definiert man stattdessen ein Interface Druckbar mit der Operation gibDruckdaten(), das alle Klassen mit druckbaren Instanzen realisieren, reduziert sich die Schnittstelle der Klasse Druckerwarteschlange auf eine einzige Operation drucken() (vgl. Abb. 30–19).

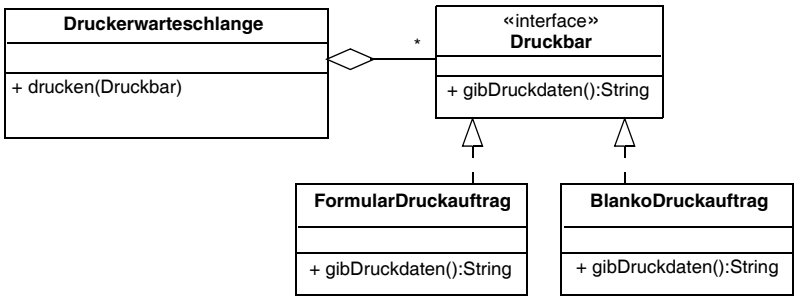

Abb. 30–19 *Die Druckerwarteschlange druckt alles, was »druckbar« ist*

Das Interface Druckbar ermöglicht, beliebig viele verschiedene Objekte mit einer gemeinsamen Eigenschaft (»druckbar«) einheitlich zu verwenden. Soll die Druckerwarteschlange z. B. auch noch Instanzen der Klassen PostscriptDokument und ASCIIText drucken können, müssen auch diese Klassen das Interface Druckbar realisieren und die Operation gibDruckdaten() implementieren. Die Klasse Druckerwarteschlange bleibt dabei völlig unverändert. Abbildung 30–20 zeigt das resultierende Klassendiagramm. Die »Stecknadelköpfe« mit der Beschriftung »Druckbar« an den Klassen FormularDruckauftrag, BlankoDruckauftrag, PostscriptDokument und ASCIIText bedeuten, dass jede dieser Klassen das Interface Druckbar realisiert.

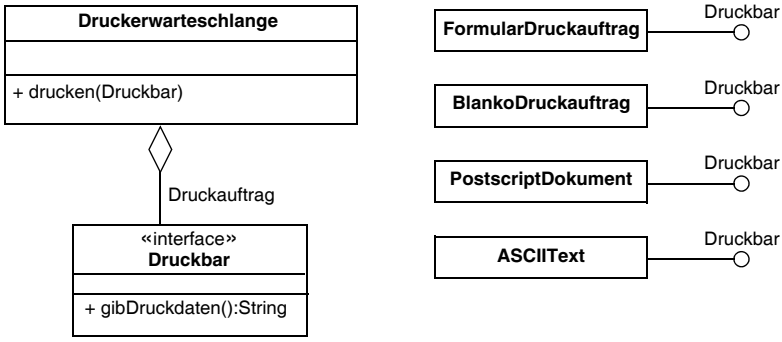

Abb. 30–20 *Alternative UML-Notation der Realisierungen eines Interfaces*

30.6 Parametrisierte Klassen und Interfaces

Im Entwurf werden immer wieder bestimmte allgemeine Funktionalitäten benötigt, die sich in ihren konkreten Realisierungen nur um bestimmte Nuancen unterscheiden. Standardbeispiel hierfür sind Behälter wie Stapel oder Listen, in denen Objekte zwischengespeichert und nach bestimmten Regeln wieder entnommen werden können.

Um nicht für jede Klasse, deren Instanzen in einem solchen Behälter abgelegt werden sollen, eine komplette Realisierung des zur Aufnahme der Instanzen dieser Klasse

dienenden Behälters angeben zu müssen, verwendet man im Entwurf parametrisierte Klassen (vgl. Abschnitt 10.7), welche den »gemeinsamen Nenner« der konkreten Realisierungen definieren.

Jede parametrisierte Klasse bzw. jedes parametrisierte Interface definiert also zunächst eine Familie von Typen, die sich im Wesentlichen durch unterschiedliche (Sub-)Typen von Parametern und Instanzvariablen unterscheiden, und enthält darüber hinaus auch Implementierungen der Operationen, die unabhängig von den verwendeten Typparametern sind.

30.7 Pakete und Teilsysteme

Beim Entwurf großer Softwaresysteme entstehen rasch mehrere hundert, ja sogar mehr als tausend Klassen, so dass die Entwurfsspezifikation schnell unübersichtlich wird. Auch ist die Granularität einer Zerlegung in Klassen oft zu fein, als dass sie in einem einzigen Abstraktionsschritt geleistet werden kann. Hier schaffen *Teilsysteme* Abhilfe, die strukturell mächtiger als Klassen sind. Teilsysteme bilden neben Klassen die zweite Art von *Komponenten* eines objektorientierten Entwurfs. Ihre Verwendung führt zu weiteren Abstraktionsstufen, welche die Übersichtlichkeit und Strukturierung komplexer Entwürfe verbessern. Außerdem ermöglichen sie in vielen Fällen die Übertragung von »Komponenten« aus der Spezifikation in den Entwurf und bilden einen wichtigen Bezugspunkt für die Projektorganisation (Teambildung, Terminplanung).

Teilsysteme fassen nach logischen Kriterien und unter Beachtung des Geheimnisprinzips verschiedene Klassen unter einem Dach zusammen. Teilsysteme können geschachtelt werden. Ein Teilsystem definiert eine *Enthaltenseinsbeziehung* zwischen sich und den in ihm organisierten Komponenten. Es kontrolliert die Benutzbarkeit dieser Komponenten, steuert aber keine eigenen Ressourcen bei. Der Name eines Teilsystems endet stets mit dem Kürzel TS.

Wie eine Klasse besteht ein Teilsystem aus einer Schnittstelle und einem Rumpf. Der *Rumpf* umfasst die zum Teilsystem gehörenden Klassen bzw. Teilsysteme, die man als *lokale Komponenten* bezeichnet. Die in der *Schnittstelle eines Teilsystems* aufgeführten Komponenten bilden eine Untermenge der Komponenten. Es sind genau diejenigen Komponenten, die für Klassen bzw. Teilsysteme außerhalb des Teilsystems sichtbar sind. Alle anderen lokalen Komponenten sind nur *lokal benutzbar*.

Parameter der Operationen von Teilsystem-Schnittstellen werden »by-value« übergeben. Das bedeutet insbesondere, dass keine Referenzen auf Instanzen der im Teilsystem gekapselten Klassen dessen Schnittstelle überqueren, sondern höchstens ihre Attributwerte (ggf. zusammengefasst in *Datentransferobjekten* mit ausschließlich öffentlichen Attributen primitiven Typs und ohne Operationen). Innerhalb eines Teilsystems können natürlich Instanzen von im Teilsystem gekapselten Klassen als Operationsparameter auftreten.

30.8 Zusammenfassung der Regeln und Heuristiken

Dieser Abschnitt führt zur besseren Übersichtlichkeit die angegebenen Regeln und Heuristiken der Reihe nach auf.

Geheimnisprinzip

KR1 Attribute werden im Entwurf immer als privat deklariert.

Kopplung und Kohäsion

H1 Sind nur sehr wenige Klassen (Anhaltspunkt: bis zu drei) eng gekoppelt, versucht man, die Klassen zu einer einzigen Klasse zusammenzufassen.

H2 Sind mehrere Klassen (mehr als drei) stark gekoppelt, versucht man, die Kopplung durch eine Vermittlerklasse abzuschwächen.

H3 Ist eine Funktionalität für die starke Kopplung verantwortlich, versucht man, sie in eine andere Klasse zu verschieben.

H4 Sind innerhalb einer Generalisierungshierarchie mehrere Standardoperationen der Oberklasse für die starke Kopplung verantwortlich, versucht man, sie durch den Aufruf einer (komplexeren) Operation der Oberklasse zu vermeiden.

KR2 Jede »Ganzes-Klasse« muss wissen, welche Teil-Klassen sie enthält, aber keine Teil-Klasse sollte ihre Ganzes-Klasse(n) kennen.

KR3 Teil-Klassen derselben Ganzes-Klasse sollten sich nicht gegenseitig benutzen (»Untergebene flüstern nicht in Gegenwart ihrer Vorgesetzten«).

H5 In einer Operation o() einer Klasse K sollten nur Operationen folgender Klassen benutzt werden (Law of Demeter):

- K selbst,
- Klassen, die als Parametertypen von o() vorkommen,
- mit K assoziierte Klassen und
- Klassen, die bei der Ausführung von o() instanziiert werden.

H6 Lassen sich die Attribute und die Operationen einer Klasse in (weitgehend) disjunkte Teilmengen zerlegen, so dass jede Operationsmenge nur auf jeweils einer Attributmenge arbeitet, dann ist die Aufteilung der Klasse zu empfehlen.

H7 Erbringt eine Operation mehr als eine einzige eng umrissene Funktionalität, ist die Funktionalität auf mehrere Operationen aufzuteilen.

Konformität

H8 Ist eine Unterklasse nicht konform zu ihrer Oberklasse, weil die Oberklasse Eigenschaften hat, die für die Unterklasse nicht zutreffen, sollte eine neue gemeinsame Oberklasse gebildet werden, die in einer Generalisierungsbeziehung zu jeder der beiden ursprünglichen Klassen steht.

H9 Klassen werden so definiert, dass sie für Unterklassenbildung offen, aber für unkontrollierte Zugriffe geschlossen sind (Open-Closed-Principle). Um möglichst viele Operations-Implementierungen in Unterklassen wiederverwenden zu können, werden Operationen einer Oberklasse, die von Operationen der Klasse selbst und von (zukünftigen) Unterklassen benutzt werden, als protected gekennzeichnet.

Entwurf mit Verträgen

KR4 Vor- bzw. Nachbedingungen beinhalten keine vorhersehbaren Ausnahmefälle, die immer gesonderter Behandlung bedürfen.