5

Transformationen für die Codegenerierung

Wer klare Begriffe hat,
kann befehlen.
Johann Wolfgang von Goethe

Dieses Kapitel ergänzt die prinzipiellen Überlegungen zur Codegenerierung aus Kapitel 4 um konkrete Techniken und Transformationen. Dabei wird die Vorgehensweise zur Umsetzung von Klassendiagrammen, Objektdiagrammen, der Codegenerierung aus der OCL, der Ausführung von Statecharts und der Testgenerierung aus Sequenzdiagrammen in Java in jeweils einem der Abschnitte 5.1 bis 5.5 erklärt. Bei den Klassendiagrammen werden dabei eher bereits bekannte Konzepte in kompakter Form als Transformationen aufbereitet und Alternativen diskutiert. Insbesondere der Abschnitt 5.1 über Klassendiagramme dient dabei zur Demonstration der systematisierten Darstellung von Transformationsregeln für die Codegenerierung.

Für Statecharts werden Generierungsalternativen intensiv diskutiert. Für alle anderen Notationen werden vor allem grundlegende Prinzipien der Übersetzung behandelt, weil so Nutzern unter anderem die Möglichkeit gegeben wird, auf die Zielsprache bzw. Zielumgebung zugeschnittene Übersetzungsformen selbst zu entwickeln bzw. in Abwesenheit eines geeigneten Werkzeugs diese manuell durchzuführen.

B.Rumpe, *Agile Modellierung mit UML*, 2. Aufl., Xpert.press
DOI 10.1007/978-3-642-22430-0_5, © Springer-Verlag Berlin Heidelberg 2012

5.1 Übersetzung von Klassendiagrammen

In diesem Abschnitt wird die Transformation von Konzepten des Klassendiagramms und einiger damit zusammenhängender Elemente in Java als komplexeres Beispiel durch eine Sammlung von Transformationsregeln beschrieben, die auch die Möglichkeiten zur Beschreibung von Alternativen und von Kompositionen der Regeln zeigen. Bei dieser Übersetzung werden weder Java-Frameworks oder Infrastrukturkonzepte wie JavaBeans oder Middleware-Komponenten noch Datenbank-Anbindungen berücksichtigt. Dafür sind jeweils spezielle Generatoren notwendig, die in der hier angestrebten allgemeinen Form nicht diskutiert werden können.

5.1.1 Attribute

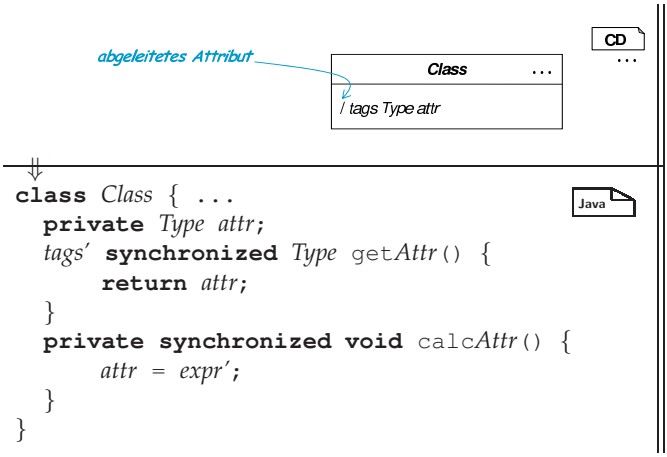
Für normale und statische Attribute von Klassen wurde mit der Transformationsregel *Attribut1* von Seite 100 bereits eine Regel zur Umsetzung angegeben. Diese kann im Prinzip auch für *abgeleitete Attribute* verwendet werden, ignoriert jedoch ein wesentliches Merkmal dieser Art von Attributen. Im Normalfall existiert für ein abgeleitetes Attribut eine als Invariante formulierte Berechnungsvorschrift. Alternativ dazu kann auch eine bereits in der Zielsprache Java formulierte Methode existieren, die die Berechnung des Attributs vornimmt. Nach unserer Konvention heißt eine solche Methode `calc`*Attr*.

Attribut2^{eager}*:* Abgeleitete Attribute - Eager Version	
Erklärung	Für ein abgeleitetes Attribut */attr* existiert eine Berechnungsvorschrift als OCL-Invariante der Form *attr=expr* oder eine Methode `calc`*Attr*. Änderungen der zur Berechnung verwendeten Attribute treten gegenüber der Abfrage des abgeleiteten Attributs selten auf. Deshalb wird das abgeleitete Attribut sofort neu berechnet und gespeichert.

(Fortsetzung auf nächster Seite)

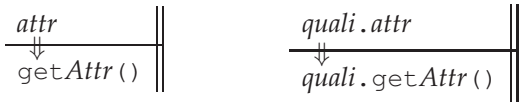
(Fortsetzung von Tabelle 5.1.: Attribut2^{eager}: Abgeleitete Attribute - Eager Version)

Attributdefinition



- Attributdefinition und `getAttr` werden wie bei der Standardregel *Attribut1* transformiert. Eine `setAttr` Methode existiert jedoch nicht.
- Ist die Berechnungsvorschrift als nicht-rekursive OCL-Bedingung in der Form `attr=expr` angegeben, so wird diese in Java-Code `expr'` transformiert und in die Methode `calcAttr` eingebettet. Alternativ kann diese Methode bereits existieren oder durch eine Nachbedingung in der angegebenen Form spezifiziert sein.

Attributzugriff



- Wie in Transformationsregel *Attribut1*.

Attributbesetzung

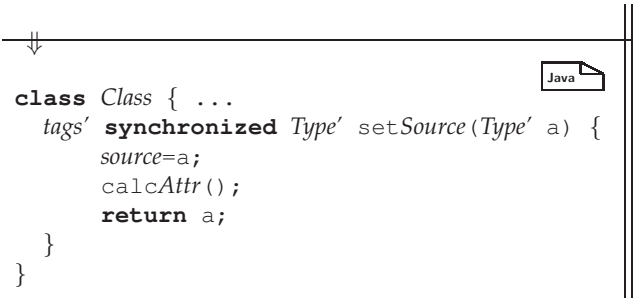
ist nicht möglich.

Besetzung eines Ausgangsattributs

durch Analyse des OCL-Ausdrucks beziehungsweise der vorhandenen `calcAttr`-Implementierung können die Ausgangsattribute ermittelt werden, von denen *attr* abgeleitet ist. Für jedes Ausgangsattribut *source* wird die `set`-Methode erweitert:

(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 5.1.: Attribut2^{eager}: Abgeleitete Attribute - Eager Version)

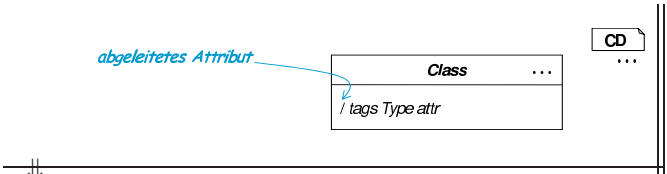
<table>
<tr><td></td><td>


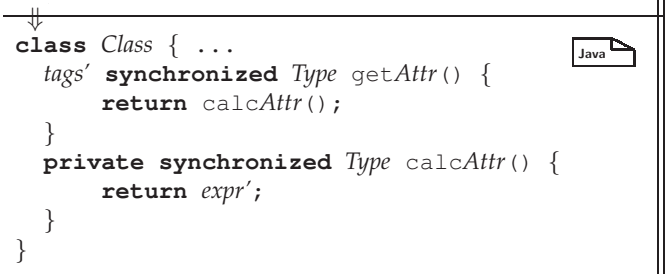
Dies zeigt nur den (einfachen) Fall, dass Ausgangs- und abgeleitetes Attribut in demselben Objekt lokalisiert sind. Ist das nicht der Fall, muss eine bidirektionale Verbindung zwischen beiden Objekten bestehen, die durch die aktuellen Veränderungen nicht beeinträchtigt sein sollte.[1]
</td></tr>
<tr><td>Beachtenswert</td><td>Die Veränderung eines Attributs hat die automatische Veränderung aller davon abgeleiteten Attribute zur Folge. Dies kann zu kaskadenartigen Neuberechnungen abgeleiteter Attribute führen und ineffizient sein, wenn mehr Änderungen als Abfragen auftreten.
Zirkuläre Abhängigkeiten führen darüber hinaus zu nichtterminierenden Neuberechnungen und sind daher verboten.</td></tr>
</table>

Tabelle 5.1.: *Attribut2^{eager}:* Abgeleitete Attribute - Eager Version

Der oben formulierten „eager" Version der Umsetzung kann eine „lazy" Version entgegengesetzt werden, die den Attributwert nur bei Bedarf berechnet. Aufgrund der Ähnlichkeiten zur vorherigen Transformationsregel wird diese verkürzt wiedergegeben:

[1] Die Bidirektionalität ist notwendig, weil die Berechnung nicht dem Kontrollfluss folgt, sondern eine Form der Change-Propagation (Publisher-Subscriber-Pattern) und damit eine Umkehrung der Berechnungsabhängigkeiten realisiert wird. Eine Datenflussanalyse kann prüfen, ob dieses Pattern mit all seiner Infrastruktur eingebaut werden muss oder eine bidirektionale Assoziation vorhanden ist. Bei einem geeigneten Generator kann auch durch den Anwender eingegriffen werden, um eine optimale Umsetzung zu sichern.

<table>
<tr><th colspan="2">$Attribut2^{lazy}$: Abgeleitete Attribute - Lazy Version</th></tr>
<tr><td>Erklärung</td><td>Für ein abgeleitetes Attribut /attr existiert eine Berechnungsvorschrift als OCL-Invariante der Form attr=expr oder eine Methode <code>calc</code>Attr.
Die Häufigkeit der Änderungen der zur Berechnung verwendeten Attribute liegt in einer ähnlichen Größenordnung wie die Abfrage des abgeleiteten Attributs. Deshalb wird das abgeleitete Attribut erst bei Bedarf berechnet und nicht gespeichert.</td></tr>
<tr><td>Attributdefinition</td><td> 




```
class Class { ...
  tags' synchronized Type getAttr() {
        return calcAttr();
  }
  private synchronized Type calcAttr() {
        return expr';
  }
}
```

Siehe Anmerkungen zu $Attribut2^{eager}$.</td></tr>
<tr><td>Attribute</td><td>Attributzugriff erfolgt wie bei $Attribut2^{eager}$.
Die direkte Attributbesetzung ist wie bei $Attribut2^{eager}$ nicht möglich.
Die Besetzung eines Ausgangsattributs (von dem dieses abhängt) muss nicht angepasst werden.</td></tr>
<tr><td>Beachtenswert</td><td>Vorteil gegenüber der $Attribut2^{eager}$ Version ist, dass der Kontrollfluss nicht invertiert wurde und damit keine bidirektionale Assoziationen oder eine andere Infrastruktur notwendig sind. Ineffizienz kann aber durch wiederholt durchgeführte Berechnung des Attributs entstehen.
Zirkuläre Abhängigkeiten führen auch hier zu nichtterminierenden Neuberechnungen und sind daher verboten.</td></tr>
</table>

Tabelle 5.2.: $Attribut2^{lazy}$: Abgeleitete Attribute - Lazy Version

Eine im Bereich der graphischen Oberflächen gelegentlich verwendete Form des Model-View-Controller-Pattern nutzt Vorteile beider Ansätze, in-

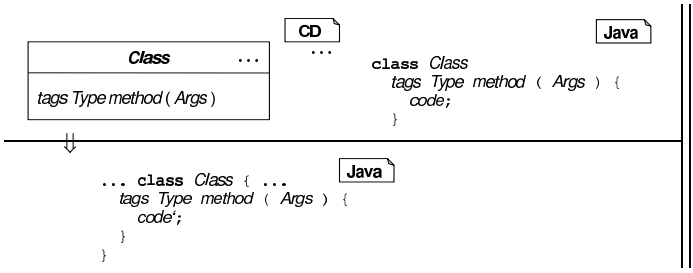
dem die change propagation nur in einer booleschen Statusvariable vermerkt wird, aber eine Neuberechnung erst bei Bedarf erfolgt.

5.1.2 Methoden

Für die Implementierung von Methoden stehen mehrere Strategien zur Verfügung, die von der Ausgangsituation abhängig sind:

1. Der Methodenrumpf ist bereits in einem anderen Artefakt formuliert und muss nur in die Methode eingesetzt werden. Dabei werden auch die notwendigen Transformationen beispielsweise von Attributzugriffen vorgenommen.
2. Die Methode ist durch ein Vor-/Nachbedingungspaar beschrieben, wobei die Nachbedingung, wie in Abschnitt 4.1 diskutiert, algorithmisch formuliert ist und direkt in Code umgesetzt werden kann.
3. Die Methode ist durch ein Vor-/Nachbedingungspaar beschrieben, das aber nicht algorithmisch umsetzbar und deshalb nur für Tests geeignet ist.
4. Für diese Methode gibt es noch keine Implementierung oder Spezifikation.

Für jeden dieser Fälle ist eine eigenständige Vorgehensweise notwendig. Der erste Fall benötigt nur die Integration des Methodenrumpfs mit der Signatur sowie die Umsetzung zum Beispiel der Attributzugriffe im Methodenrumpf.

Methode1^{impl}*:* Methoden mit gegebener Implementierung	
Erklärung	Eine Methode *meth* mit gegebenem Methodenrumpf *code* wird in Java umgesetzt.
Methoden-definition	

(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 5.3.: Methode1^{impl}: Methoden mit gegebener Implementierung)

	• Der Methodenrumpf *code* besteht aus einer Sequenz von Anweisungen. Diese wird entsprechend der gültigen Transformation für Anweisungen in *code'* transformiert, um zum Beispiel Attributzugriff und -besetzung oder die Umsetzung von Zusicherungen (assertions) zu behandeln. • Sind Sichtbarkeitsangaben, Parameternamen, -typen und Ergebnistyp teilweise im Text und im Diagramm gegeben, so dürfen sie sich ergänzen, aber nicht widersprechen.

Tabelle 5.3.: *Methode1^{impl}:* Methoden mit gegebener Implementierung

Für die Beschaffung des Methodenrumpfs gibt es in den heute verfügbaren Werkzeugen mehrere Ansätze. Eine Möglichkeit ist, den Rumpf als Textstück, zum Beispiel als Kommentar, der Methodensignatur im Diagramm zu hinterlegen und durch Anwählen zugänglich zu machen. Dies ist allerdings für große Systeme mit vielen Methoden nicht praktikabel. Die Technik des „Round Trip Engineering" liest die Methodenrümpfe direkt aus dem Quellcode, um sie dorthin zurück zu schreiben.[2] Hier wird eine weitere Alternative vorgeschlagen, die einige Vorteile des Aspect Oriented Programming mit einer kompakten Aufschreibung vereint. Grundidee ist es, Methodenimplementierungen nicht notwendigerweise nur nach der Klassenzugehörigkeit, sondern auch nach funktionalen Kriterien zusammenzufassen. So können beispielsweise die `protocol`-Methoden aller Klassen, die Methoden zum Durchlauf einer Teilehierarchie oder die Serialisierungsmethoden mehrerer Klassen jeweils in eine Datei zusammengefasst werden, die dann diesen *Aspekt* des Programms beinhaltet.

Neben oder statt Java-Implementierungen können auch OCL-Spezifikationen von Methoden in der Form von Vor- und Nachbedingungspaaren verwendet werden. In Abschnitt 3.4.3, Band 1 ist die Integration mehrerer solcher Methodenspezifikationen behandelt worden. Deshalb kann hier von einem einzelnen Paar ausgegangen werden. Ist die Spezifikation algorithmisch in der in Abschnitt 4.1.2 diskutierten Form, so kann daraus direkt Code erzeugt werden.[3]

[2] „Round Trip Engineering" erlaubt es, sowohl in der abstrakten Diagrammsicht als auch in der Implementierung Veränderungen vorzunehmen, die dann in der jeweils anderen Sicht nachgezogen werden. Diese Vorgehensweise ist jedoch fragil, da nach heutigem Stand der Technik nur Kommentare verwendet werden, um die Verbindung zwischen beiden Sichten aufrecht zu erhalten.

[3] Die Ausführbarkeit besteht im Wesentlichen darin, dass die Nachbedingungen aus einer Konjunktion von Gleichungen besteht, deren linke Seiten veränderbare Attribute beziehungsweise das Ergebnis darstellen und die rechten Seiten diese Attribute in nur eingeschränkter Form benutzen. Zum Beispiel dürfen keine zirkulären

Weil die Umsetzung einer derartig spezifizierten Methode im Wesentlichen auf der in Abschnitt 5.3 diskutierten Umsetzung von OCL in Java-Code beruht, soll hier auf eine explizite Formulierung der Transformationsregel verzichtet werden.

Im dritten oben genannten Fall existiert sowohl eine Implementierung als auch eine Spezifikation. Damit ist es sinnvoll, die Spezifikation zur Prüfung während der Laufzeit einzusetzen. Der Generator weiß, ob er effizienten Produktionscode oder mit diesen Prüfungen instrumentierten Code erzeugen soll. Zum Beispiel bieten Eiffel- und Java-Übersetzer die Möglichkeit, Zusicherungen optional zu übersetzen.

Im Prinzip ist nur die Vorbedingung vor Start der Methode und die Nachbedingung nach deren Ende zu testen. Dabei sind jedoch unter Umständen mit dem **let**-Konstrukt lokal definierte Variable und eventuell in der Nachbedingung genutzte Anfangszustände von Attributen zu sichern. Diese Sicherung kann komplex sein, wenn die benutzten Attribute in anderen Objekten liegen und die Zugangspfade ihrerseits verändert worden sein können. Eine über den Abschnitt 3.4.3, Band 1 hinausgehende ausführliche Diskussion dieser Problematik ist zum Beispiel in [RG02] zu finden.

Als letzte Variante soll hier noch der Fall kurz diskutiert werden, in dem es weder eine Implementierung noch eine algorithmisch ausführbare Spezifikation für eine Methode gibt. Dann kann die Methode nicht automatisiert implementiert werden. Für Simulationen und Tests, die diese Methode vielleicht nur marginal berühren, sind jedoch Strategien möglich und sinnvoll, Dummy-Implementierungen zu generieren.

- Spielt die Methode bei den durchzuführenden Tests keine Rolle, so kann ein Fehleraufruf oder die Rückgabe eines Default-Werts in die Methode generiert werden.
- Ist die Methode noch nicht realisiert, so kann ein interaktives Eingabefeld während Simulationsläufen dazu benutzt werden, dass der Nutzer auf Basis der aktuellen Parameter jeweils selbst das Ergebnis bestimmt.
- Für eine, endliche Menge von Eingaben können in einer Tabelle Ergebnisse abgelegt sein. Diese Ergebnisse können zum Beispiel aus früheren interaktiven Simulationsläufen mitprotokolliert worden sein.

Einerseits ist eine interaktive Eingabe von Ergebnissen einzelner Methoden für automatisierte Testläufe nicht sinnvoll, andererseits können damit während der Vorführung eines Prototypen sofort Anwenderentscheidungen in das System zurückgeführt werden. Diese können protokolliert und später zum Beispiel als Testdaten genutzt werden. Diese interaktive Form des Erkenntnisgewinns ist sicherlich beschränkt, kann aber unter Umständen zu effektiverer Kommunikation mit Anwendern führen.

Abhängigkeiten zwischen Attributen bestehen, um eine sequentielle Berechnung zu ermöglichen. Jede Gleichung darf zusätzlich mit einer Bedingung versehen sein.

In der UML/P ist es nicht üblich, Hilfsmethoden wie `get`*Attr* in Klassendiagrammen explizit zu vermerken. Dadurch bleibt das Modell kompakter und übersichtlicher. Auch müssen diese Funktionen in Coderümpfen, die bei der Generierung übersetzt werden, nicht explizit verwendet werden. Es reicht aus, den Attributzugriff und die Attributbesetzung in Form von Zuweisungen einzusetzen. Ein Codegenerator übersetzt diese wie in den Transformationsregeln beschrieben in Methodenaufrufe. Es sollte jedoch erlaubt sein, diese Methoden direkt zu verwenden. Außerdem ist unter Umständen sinnvoll, die Generierung einer solchen Methode vorwegzunehmen, indem eine manuelle Implementierung angegeben wird. Dadurch lassen sich eventuell Optimierungen vornehmen oder zusätzliche Funktionalitäten realisieren.

5.1.3 Assoziationen

Eine unidirektionale Assoziation wird standardmäßig durch ein Attribut umgesetzt. Der Rollenname wird dabei als Attributname verwendet. Fehlt der notwendige Rollenname, so wird wie bei den in Abschnitt 3.3.8, Band 1 angegebenen Navigationsregeln ein Attributname aus dem Assoziationsnamen oder dem Namen der gegenüberliegenden Klasse gebildet.

Kardinalitäten werden entsprechend berücksichtigt: „`0..1`“ führt zu einem einfachen Attribut, das den Wert `null` annehmen darf, „`1`“ führt zu einem einfachen Attribut, das immer besetzt ist, und eine Assoziation mit Kardinalität „`*`“ wird mengenwertig. Abhängig von zusätzlichen Merkmalen wie {`ordered`} stehen Mengen- oder Listen-Implementierungen zur Auswahl. Für qualifizierte Assoziationen wird entsprechend eine Abbildung (`Map`) zur Verfügung gestellt.

Bidirektionale Assoziationen werden durch Attribute auf beiden Seiten realisiert, die durch ein geeignetes Methodenprotokoll konsistent gehalten werden. Ist keine Navigationsrichtung angegeben, so wird eine geeignete Navigationsrichtung aus dem Kontext ermittelt und gegebenenfalls werden beide Richtungen realisiert.

Um die oben genannte Konsistenz bidirektionaler Assoziationen zu sichern, werden alle Zugriffe auf die Assoziation über generierte Methoden geführt. Die Form dieser generierten Methoden, also das für eine Assoziation verwendbare API, hängt von den Eigenschaften und Merkmalen der Assoziation ab.

So werden bei den Merkmalen {`addOnly`} und {`frozen`} entsprechende Funktionen zur Modifikation eingeschränkt. Abgeleitete Assoziationen werden mit denselben Prinzipien behandelt, wie abgeleitete Attribute. Das heißt, es werden nur Abfragemethoden zur Verfügung gestellt und diese durch Berechnungen implementiert.

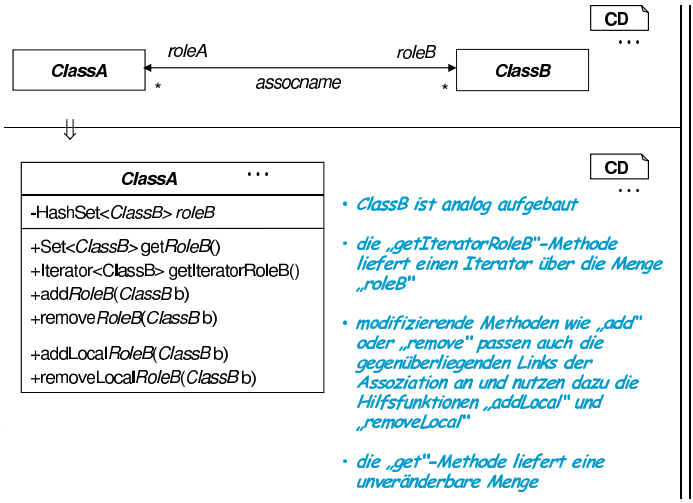
Nachfolgende Transformation ist exemplarisch für bidirektionale, in beiden Richtungen mit Kardinalität „`*`“ versehene Assoziationen.

Assoziation^{*,*,bidir}: Bidirektionale Assoziation

Erklärung

Assoziationen werden in den Zustandsraum zumindest einer der beteiligten Klassen transformiert, indem entsprechende Attribute und Zugriffsfunktionen generiert werden.
Diese Transformationsregel ist für bidirektionale Assoziationen mit Kardinalität „$*$“ in beiden Richtungen geeignet. Die Assoziation ist nicht abgeleitet und keine Komposition.

Definition der Assoziation

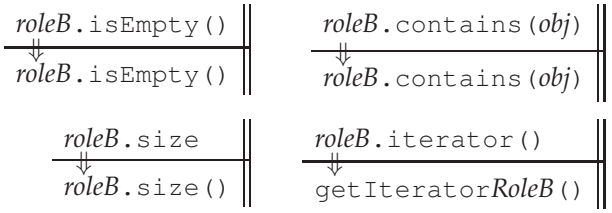


- Nachfolgende Ausführungen gelten für *ClassB* entsprechend, da die Situation symmetrisch ist.
- Zugriffe auf die Assoziation werden durch Zugriffe auf das Attribut *roleB* modelliert, das die in Abschnitt 3.3.5, Band 1 eingeführte Signatur von `Collection`<*ClassB*> besitzt.
- Der Attributname *roleB* extrahiert sich aus dem Rollennamen, dem Namen der Assoziation (*assocname*) oder wenn beide fehlen, dem Namen der gegenüberliegenden Klasse (*classB*). Allerdings muss die Eindeutigkeit des Namens gewährleistet sein (siehe Abschnitt 3.3.8, Band 1).
- Die Umsetzung der zur Modellierung verwendeten Konstrukte in den Implementierungscode erfolgt relativ schematisch, jedoch werden verändernde Operationen wie `add`*RoleB* oder `remove`*RoleB* entsprechend angepasst, um damit die Konsistenz der bidirektionalen Assoziation sicherzustellen.

(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 5.4.: Assoziation^{*,*,bidir}: Bidirektionale Assoziation)

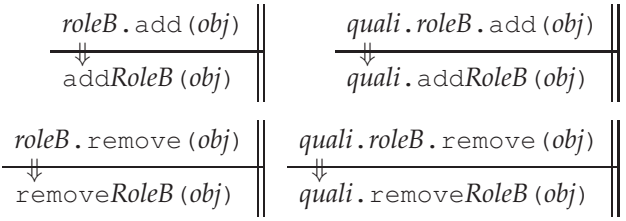
Zugriffsfunktionen



etc.

- Lesende Zugriffe bleiben weitgehend erhalten. Die Umsetzung entspricht der Standardumsetzung des OCL-`Collection`-Interface nach Java.
- Über den Iterator können auch Links gelöscht werden (analog zu `remove`).

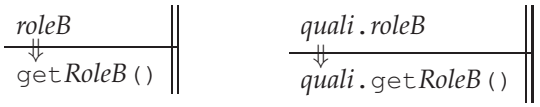
Modifikation



etc.

- Modifizierende Zugriffe werden auf speziell generierte Methoden abgebildet.
- Weitere modifizierende Zugriffe, wie zum Beispiel *roleB*`.clear()`, werden ebenfalls entsprechend abgebildet.

OCL-Navigation



- `get`*RoleB* liefert als Ergebnis eine unveränderbare Menge.[4]

(Fortsetzung auf nächster Seite)

[4] Die Methode `unmodifiableSet` der Klasse `java.util.Collections` produziert aus einer beliebigen eine unveränderbare Menge, ohne allerdings die Signatur zu verändern.

(Fortsetzung von Tabelle 5.4.: Assoziation^{*,*,bidir}: Bidirektionale Assoziation)

Zusätzliche Methoden

⇓

Java

```
class ClassA { ...
  public synchronized Set<ClassB> getRoleB() {
      return Collections.unmodifiableSet(roleB);
  }
  public synchronized void addRoleB(ClassB b) {
      roleB.add(b);
      b.addLocalRoleA(this);
  }
  public synchronized void
                          addLocalRoleB(ClassB b) {
      roleB.add(b);
  }
}
```

- Hilfsfunktionen wie `addLocal`*RoleB* oder `removeLocal`*RoleB* dürfen außerhalb dieses Protokolls nicht benutzt werden, obwohl sie als `public` generiert werden. Sie stehen deshalb dem Entwickler nicht zur Verfügung.[5]
- Weitere Modifikatoren wie `remove`*RoleB* oder `clear`*RoleB* werden in ähnlicher Form generiert. Allerdings erfordert zum Beispiel `clear`*RoleB* in bidirektionalen Assoziationen die Abmeldung jedes Links auf der gegenüberliegenden Seite, hat also lineare Komplexität.
- Ist das Merkmal {`addOnly`} angegeben, so stehen `remove`-Operationen nicht zur Verfügung.
- Ist das Merkmal {`ordered`} angegeben, so wird eine Listen-Implementierung gewählt und die entsprechende Funktionalität zusätzlich angeboten.

Beachtenswert

Die durch ein Protokoll gesicherte Konsistenz zwischen beiden Enden einer bidirektionalen Assoziation besitzt im Normalfall nur konstanten Zusatzaufwand, ist also vertretbar. Ist eine Assoziation nur unidirektional, so kann dieser Aufwand dennoch wegfallen.

Tabelle 5.4.: *Assoziation^{*,*,bidir}:* Bidirektionale Assoziation

[5] Dies kann einerseits durch geeignete Kontextprüfungen bei den Coderümpfen gesichert, andererseits aber auch durch Verwendung außerhalb des Generators nicht bekannter Methodennamen erreicht werden.

Die Umsetzung von Assoziationen in Java-Code zeigt, wie groß die Variationsmöglichkeiten bei der Codegenerierung sind. Variabel abhängig von den Eigenschaften der Assoziation ist nicht nur das API einer Assoziation (also welche Funktionen in UML/P zum Zugriff und zur Manipulation zur Verfügung stehen), sondern auch die intern genutzte Datenstruktur. Da die Wahl der Datenstruktur zumindest Auswirkungen auf das Laufzeitverhalten der Implementierung hat, wird sinnvollerweise durch geeignete Steuerungsmechanismen wie etwa dem Merkmal {`HashMap`} oder durch geeignete Anpassung der Skripte die Auswahl der Implementierung ermöglicht.

Für Assoziationen mit beschränkten Kardinalitäten ist außerdem zu klären, wie der Versuch einer Verletzung der Kardinalität behandelt wird. Dafür gibt es zum Beispiel die Varianten, dies robust zuzulassen, aber gegebenenfalls eine Warnung zu protokollieren, bis hin zur Erzeugung einer Exception, die dann vom aufrufenden Objekt zu behandeln ist.

Neben der oben vorgeschlagenen Form der Implementierung einer Assoziation gibt es Vorschläge, die Links durch eigenständige Objekte zu realisieren oder durch eine global verwaltete Datenstruktur zu ersetzen. All diese Erweiterungen haben als Ziel, zusätzliche Funktionalität anzubieten, die durch das API der Modellierung zugänglich werden, oder Verhaltens- beziehungsweise Sicherheitseigenschaften zu optimieren. Eine globale statische Datenstruktur in Form einer Abbildung von Quell- zu Zielobjekt ist zum Beispiel von Interesse, wenn die Assoziation sehr dünn besetzt ist und der Speicherplatz dadurch effizienter genutzt wird. Dies sollte dem Nutzer der API verborgen bleiben, da es sich um Realisierungsdetails handelt.

Die notwendige Umsetzung von Java-Code zur Sicherung der Konsistenz der Assoziation zeigt, dass es wichtig ist, dass der Codegenerator die vollständige Kontrolle über alle Teile des generierten Codes, also auch über Methodenrümpfe hat. Dadurch wird beispielsweise die für bidirektionale Assoziationen gültige Konsistenzbedingung gesichert:

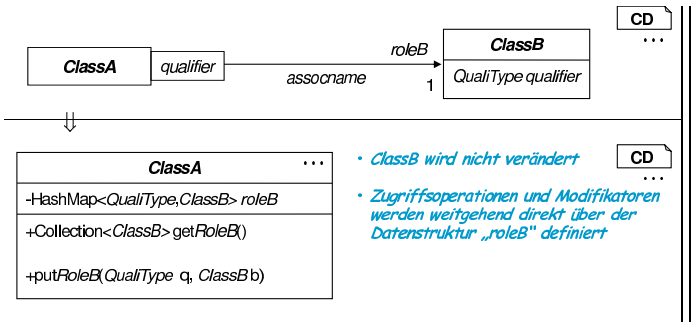
```
context ClassA a, ClassB b inv:
 a.roleB.contains(b) <=> b.roleA.contains(a)
```

Verfahren des Roundtrip-Engineering können dies nicht leisten, da es dem Entwickler die Möglichkeit gibt, beliebig in generierte Datenstrukturen einzugreifen. Dort müsste also diese Konsistenzbedingung zur Laufzeit geprüft werden. Bei einer Transformation der Methodenrümpfe durch den Codegenerator können die Zugriffe und Modifikationen für die Assoziation überprüft beziehungsweise transformiert und damit verhindert werden, dass dem Entwickler die Methode `addLocal`*RoleB* zur Programmierung zur Verfügung steht.[6]

[6] Meistens wird den generierten Methoden ein anderer, nur intern bekannter Name gegeben und damit auch eine zufällige Namensübereinstimmung verhindert. Dadurch kann der Entwickler eine beliebige Methode `addLocal`*RoleB* definieren, die mit der generierten Methode nicht in Zusammenhang steht.

5.1.4 Qualifizierte Assoziation

Die qualifizierte Assoziation bietet gegenüber der normalen Assoziation ein angepasstes API, das die qualifizierte Selektion und Manipulation erlaubt, aber auch einige Operationen zur Modifikation unqualifizierter Assoziationen verbietet. Deshalb wird für die qualifizierte Assoziation eine eigene Transformationsliste angegeben, die auch das API beschreibt.

Assoziation^{quali}: Qualifizierte Assoziation	
Erklärung	Eine qualifizierte Assoziation wird ähnlich der normalen Assoziation umgesetzt, bietet aber angepasste Funktionalität für qualifizierten Zugriff. Diese Transformationsregel ist geeignet für unidirektionale qualifizierte Assoziationen mit Kardinalität „`1`“.[7] Die Assoziation ist weder abgeleitet noch eine Komposition. Der angegebene Qualifikator *qualifier* ist ein Attribut der gegenüberliegenden Klasse. Der Qualifikatorwert ist deshalb identisch zu dem Attributwert.
Definiten der Assoziation	 • Zugriffe auf die Assoziation werden durch Zugriffe auf das Attribut *roleB* modelliert, das eine Signatur der Form `Map<`*QualiType*`,` *ClassB*`>` besitzt. • Zusätzliche Methoden der unqualifizierten Assoziationen, wie das nachfolgend definierte `add`*RoleB*, sind möglich, weil der Qualifikator im Zielobjekt enthalten ist.

(Fortsetzung auf nächster Seite)

[7] Wie in Abschnitt 2.3.7, Band 1 beschrieben, bedeutet die Kardinalität „`1`“, dass der qualifizierte Zugriff genau ein Objekt liefert.

(Fortsetzung von Tabelle 5.5.: Assoziation^{quali}: Qualifizierte Assoziation)

Zugriffsfunktionen

roleB`.get(`*key*`)` ⇓ *roleB*`.get(`*key*`)`

roleB`.isEmpty` ⇓ *roleB*`.isEmpty()`

roleB`.containsValue(`*obj*`)` ⇓ *roleB*`.containsValue(`*obj*`)`

roleB`.keySet()` ⇓ *roleB*`.keySet()`

roleB`.containsKey(`*obj*`)` ⇓ *roleB*`.containsKey(`*obj*`)`

roleB`.values()` ⇓ *roleB*`.values()`

roleB`.size` ⇓ *roleB*`.size()`

- Lesende Zugriffe bleiben weitgehend erhalten.

Modifikation

roleB`.clear()` ⇓ *roleB*`.clear()`

roleB`.put(`*key*`,`*obj*`)` ⇓ `put`*RoleB*`(`*key*`,`*obj*`)`

roleB`.removeValue(`*obj*`)` ⇓ *roleB*`.remove(`*obj*`.`*qualifier*`)`

roleB`.removeKey(`*obj*`)` ⇓ *roleB*`.remove(`*obj*`)`

roleB`.add(`*obj*`)` ⇓ *roleB*`.put(`*obj*`.`*qualifier*`,`*obj*`)`

etc.

- Weitere modifizierende Zugriffe, wie zum Beispiel *roleB*`.putAll`, werden entsprechend abgebildet.
- Die Methode `remove(`*obj*`)` für unqualifizierte Assoziationen wird für diese Form der qualifizierten Assoziationen nicht angeboten, weil eine gleichnamige Methode für `Maps` eine andere Funktionalität erfüllt (sie entfernt Schlüsselwerte). Stattdessen werden zwei Operationen mit jeweils eigenem Namen angeboten.
- Bei gesetztem Merkmal {`addOnly`} stehen die `remove`-Operationen nicht zur Verfügung.

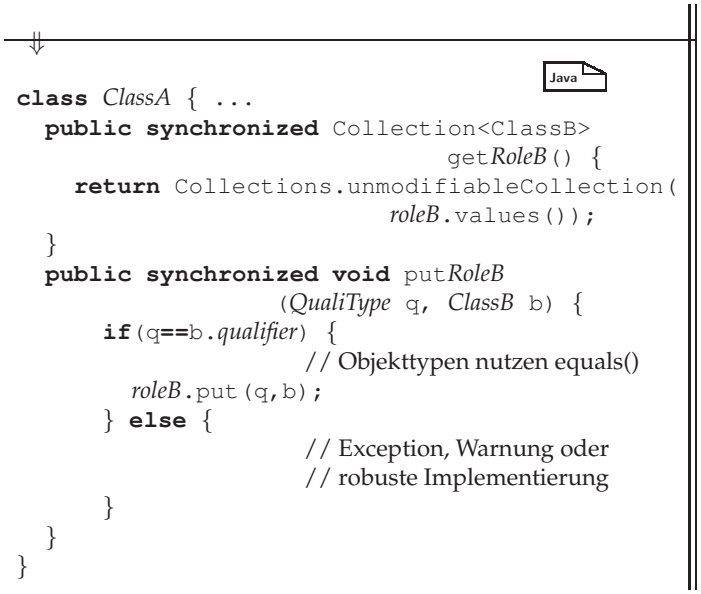
(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 5.5.: Assoziation^{quali}: Qualifizierte Assoziation)

OCL-Navigation

- Qualifizierte und unqualifizierte Navigation sind möglich.

Zusätzliche Methoden




```
class ClassA { ...
  public synchronized Collection<ClassB>
                                  getRoleB() {
    return Collections.unmodifiableCollection(
                                roleB.values());
  }
  public synchronized void putRoleB
                    (QualiType q, ClassB b) {
      if(q==b.qualifier) {
                    // Objekttypen nutzen equals()
        roleB.put(q,b);
      } else {
                    // Exception, Warnung oder
                    // robuste Implementierung
      }
  }
}
```

- Die Methode `putRoleB` wird verwendet, um sicherzustellen, dass der Qualifikatorwert (Schlüssel) und der Wert des Attributs *qualifier* identisch sind.

Beachtenswert

Abhängig vom Zweck des Codes (Test, Simulation, Produktion) werden verschiedene Strategien für die Behandlung des Fehlerfalls von einer Fehlermeldung über eine Mitteilung in einem Protokoll bis hin zur robusten Implementierung eingesetzt.
Der Zugriff auf das hier verwendete Attribut *roleB* ist entsprechend der für dieses Attribut gültigen Transformation ebenfalls umzusetzen.

Tabelle 5.5.: *Assoziation^{quali}:* Qualifizierte Assoziation

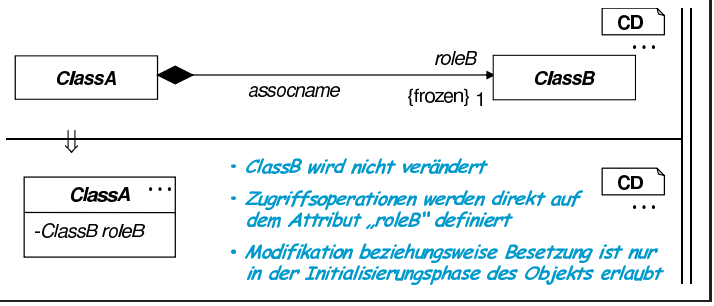
Die Transformation der qualifizierten Assoziation nutzt die Komponierbarkeit von Transformationsregeln, da hier zunächst eine Assoziation in ein Attribut transformiert wird, das durch eine weitere Transformation durch

Zugriffsmethoden gekapselt wird. Bei dieser Kapselung durch Zugriffsmethoden ist allerdings zu beachten, dass die Methode `get`*roleB* zwei unterschiedliche Aufgaben zu erfüllen hat. Bei qualifizierten Assoziationen ist zwischen (1) der Menge aller durch die Links erreichbaren Objekte und dem (2) Attributinhalt zu unterscheiden. Nur bei normalen Assoziationen sind beide Bedeutungsvarianten identisch. Die Methode `get`*roleB* realisiert Variante (1). Für die Variante (2) wird bei Bedarf eine Methode mit dem Namen `get`*roleB*`Attribute` eingeführt, die hier ein `Map`-Objekt zurückgibt. Durch die zahlreichen qualifizierten Zugriffsmöglichkeiten sollte jedoch der Zugriff auf die realisierende `Map`-Datenstruktur durch den Modellierer nicht notwendig sein.

5.1.5 Komposition

Wie bereits in Abschnitt 2.3.4, Band 1 diskutiert, besteht zwischen den Lebenszyklen des Kompositums und den davon abhängigen Objekten eine zeitliche Beziehung. Diese ist jedoch durch erhebliche Interpretationsunterschiede gekennzeichnet. Die Komposition wird strukturell wie eine normale Assoziation behandelt, das Anlegen beziehungsweise Entfernen von Links aus einer Komposition unterliegt aber der jeweiligen Interpretation. Entsprechend werden einige Operationen des Assoziations-API nicht angeboten oder unterliegen Restriktionen.

Eine Interpretation des Kompositums, die relativ verbreitet ist und im Auktionsprojekt als einzige verwendet wurde, wird nachfolgend dargestellt.

Komposition^{frozen}: Fixierte Komposition	
Erklärung	Die fixierte Form der Komposition wird genutzt, wenn das abhängige Objekt dieselbe Lebensspanne wie das Kompositum hat, während der Initialisierungsphase des Kompositums erzeugt wird und der Link zwischen beiden Objekten unveränderbar ist. Diese Transformationsregel ist geeignet für die unidirektionale Kompositionen mit Kardinalität „`1`“.
Kompositions-definition	

(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 5.6.: Komposition^{frozen}: Fixierte Komposition)

	• Die Struktur entspricht einer Assoziation mit derselben Kardinalität. • Zugriffe auf die Assoziation werden durch Zugriffe auf das Attribut *roleB* modelliert, das einen einfachen Objekttyp hat. • Der Attributname *roleB* extrahiert sich aus dem Rollennamen, dem Namen der Assoziation (*assocname*) oder wenn beide fehlen (was bei Kompositionen häufig der Fall ist), dem Namen der gegenüberliegenden Klasse (*classB*). Allerdings muss die Eindeutigkeit des Namens gewährleistet sein (siehe Abschnitt 3.3.8, Band 1).
Zugriffs-funktion	*roleB* ⇓ *roleB* • Lesende Zugriffe auf das Attribut werden durch eine nachgeschaltete Transformationsregel (normalerweise in `get`*RoleB*`()`) umgesetzt.
Modifikation	Die Besetzung des Attributs *roleB* darf ausschließlich im Konstruktor, also der Initialisierungsphase erfolgen.[8] Dafür wird entweder eine Factory oder ein `new`-Kommando eingesetzt: *roleB*=*factory*`.new`*ClassB*`(`*arguments*`)` *roleB*=`new` *ClassB*`(`*arguments*`).` Eine statische Analyse sichert, dass die einmalige Besetzung des Attributs in jedem Fall stattfindet. • Im Fall einer bidirektionalen Komposition wird in dem abhängigen Objekt durch einem in der Transformation *Assoziation*^{*,*,bidir} beschriebenen Verfahren der entsprechende Link ebenfalls gesetzt. Dazu wird die oben gezeigte Besetzung mit *roleB*= durch einen Methodenaufruf `set`*RoleB* ersetzt.
OCL-Navigation	wie in vorangegangenen Transformationsregeln

(Fortsetzung auf nächster Seite)

[8] Bei bestimmten Klassen, zum Beispiel Applets, ist die Initialisierung in eine Methode `init()` ausgelagert, die dann zur Initialisierungsphase zählt.

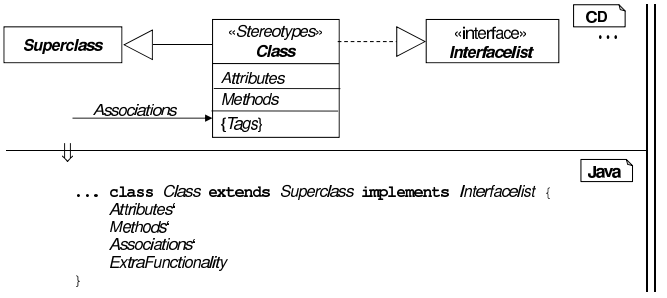
(Fortsetzung von Tabelle 5.6.: Komposition^{frozen}: Fixierte Komposition)

Beachtenswert	Die Restriktion, dass das abhängige Objekt erst im Konstruktor des Kompositums erzeugt wird, stellt sicher, dass abhängige Objekte nicht mehrfach verwendet werden.[9] Eine weniger strikte Umsetzung würde zum Beispiel erlauben, das abhängige Objekt bereits als Parameter an den Konstruktor zu übergeben. Dann kann jedoch nicht mehr sicher festgestellt werden, ob das Objekt neu erzeugt wurde und damit der Kompositionsbeziehung genügt.

Tabelle 5.6.: *Komposition^{frozen}:* Fixierte Komposition

5.1.6 Klassen

Die Übersetzung einer Klasse mit ihren Attributen, Methoden, Assoziationen, Kompositionen und den bislang noch nicht besprochenen Vererbungsbeziehungen ist relativ schematisch, da die kanonische Vorgehensweise die direkte Abbildung der UML-Klasse in die Java-Klasse ist. Die Umsetzung von Klassen ist jedoch stark getrieben durch Stereotypen und Merkmale, die steuern, welche zusätzliche Funktionalität und welche Varianten der Transformation von Attributen vorgenommen werden. In dieser Grundtransformation werden keine Stereotypen berücksichtigt.

Klassen: Umsetzung einer Klasse	
Erklärung	Eine Klasse wird direkt übernommen. In Abhängigkeit der ihr beigefügten Stereotypen und Merkmale sowie genereller Übersetzungsvorgaben wird für die Klasse zusätzliche Funktionalität generiert, die dem Entwickler bei der Benutzung der Klasse zur Verfügung steht.
Klassendefinition	

(Fortsetzung auf nächster Seite)

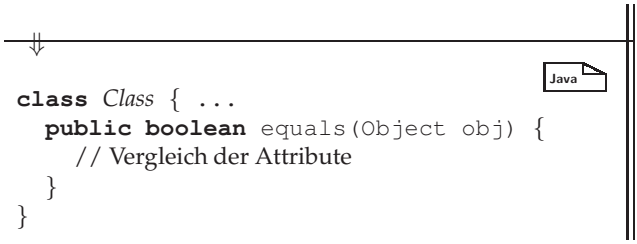
[9] Dabei wird angenommen, dass eine gegebenenfalls verwendete Factory tatsächlich neue Objekte erzeugt.

(Fortsetzung von Tabelle 5.7.: Klassen: Umsetzung einer Klasse)

<table>
<tr><td></td><td>

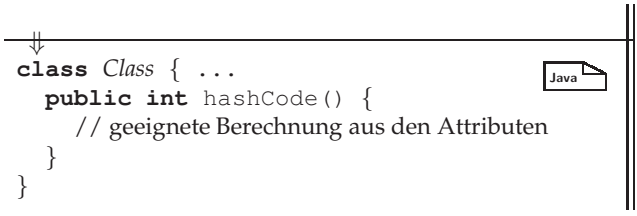
- Vererbung und Interface-Implementierung werden übernommen.
- Attribute, Assoziationen werden entsprechend der jeweils gültigen Regeln zu Code transformiert.
- Stereotypen und Merkmale steuern sowohl die Umsetzung der genannten Modellierungselemente als auch die Generierung zusätzlicher Funktionalität.

</td></tr>
<tr><td>Vergleichs-
funktion</td><td>



- Die Methode `equals` vergleicht die einzelnen, neu definierten Attribute und verwendet die gleichnamige Methode der Oberklasse.
- Assoziationen und abgeleitete Attribute werden im Normalfall zum Vergleich nicht berücksichtigt, jedoch aber Kompositionen, bei denen die gerade bearbeitete Klasse das Kompositum darstellt.
- Das Merkmal {`Equals=`*Liste*} erlaubt die explizite Auflistung, welche Attribute und Assoziationen in den Vergleich einbezogen werden. Abkürzend kann mit dem Merkmal `Equals+` eine Liste zusätzlicher Assoziationen oder mit `Equals-` eine Negativliste auszunehmender Attribute spezifiziert werden.
- Ist für die Klasse eine `equals`-Methode bereits explizit angegeben, so wird diese übernommen anstatt sie zu generieren.

</td></tr>
<tr><td>Hash-
funktion</td><td>



</td></tr>
</table>

(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 5.7.: Klassen: Umsetzung einer Klasse)

<table>
<tr><td></td><td>

- Die Hash-Funktion wird geeignet implementiert.
- Mit den Merkmalen {`Hash=`*Liste*}, {`Hash+`} und {`Hash-`} kann analog zur Vergleichsfunktion gesteuert werden, welche Attribute dafür herangezogen werden.
- Ist für die Klasse eine `hash`-Methode bereits explizit angegeben, so wird diese übernommen.

</td></tr>
<tr><td>Stringumwandlung</td><td>

⇓

```
class Class { ...
  public String toString() {
    // Umsetzung der Attribute, Assoziationen
  }
}
```

Java

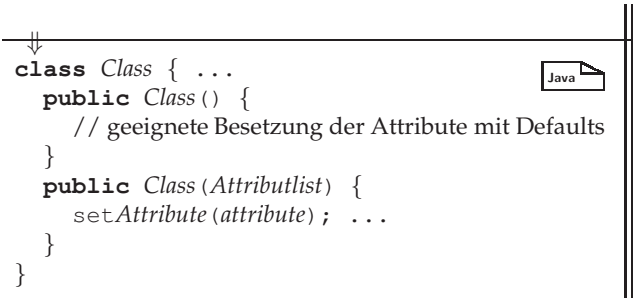
- Die Methode `toString` liefert eine einfache Umsetzung in einen String, der die Inhalte der beteiligten Attribute wiedergibt. Diese Form der Ausgabe dient vor allem für Tests und Simulationen und sollte im Produktionssystem normalerweise nicht eingesetzt werden.
- Assoziationen, die im Zustandsraum der Klasse abgelegt sind, abgeleitete Attribute und Kompositionen werden miteinbezogen.
- Die Merkmale {`ToString=`*Liste*}, {`ToString+`} und {`ToString-`} erlauben die Kontrolle darüber, welche Klassenelemente ausgegeben werden.
- Das Merkmal {`ToStringVerbosity=`*Nummer*} erlaubt die Steuerung der Verbosität. 0: Keine Ausgabe, 1: Klassenname, 2: Attributinhalte sehr kompakt (ohne erreichbare und abhängige Objekte) und 6: verbose Ausgabe jedes Attributs und jeder Assoziation in der Form *Attributname=Attributwert*, die alle erreichbaren Objekte einschließt.
- Ist für die Klasse eine `toString`-Methode bereits explizit angegeben, so wird diese übernommen.

</td></tr>
</table>

(Fortsetzung auf nächster Seite)

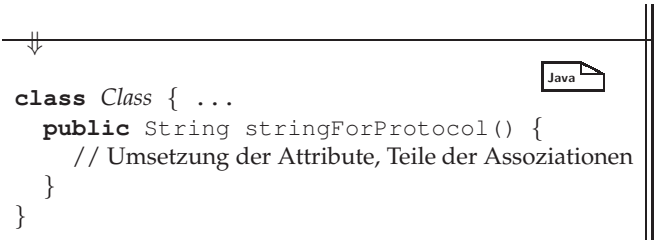
(Fortsetzung von Tabelle 5.7.: Klassen: Umsetzung einer Klasse)

Konstruktoren



- Konstruktoren werden gemäß der Generierungsstrategie erzeugt.
- Wenn nicht explizit ausgeschlossen, dann ist standardmäßig der leere Konstruktor und ein Konstruktor zur Besetzung aller Attribute dabei.
- Weil das Merkmal {new(*Attributlist*)} mehrfach anwendbar ist, können beliebig viele Konstruktoren erzeugt werden. Alternativ ist es auch möglich, Konstruktoren direkt anzugeben, weil so zusätzliche Funktionalität im Konstruktor realisiert werden kann.

Protokollausgabe



- Diese Methode arbeitet ähnlich wie `toString`, wird aber für die Ausgabe in Protokollen verwendet. Sie kann analog parametrisiert beziehungsweise manuell implementiert werden.

(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 5.7.: Klassen: Umsetzung einer Klasse)

Beachtenswert	Neben `stringForProtocol` gibt es eine Reihe weiterer Funktionen, die in entsprechender Form realisiert werden, aber hier nicht erwähnt wurden. Einige sind aus der von allen Objekten abgeleiteten Klasse `Object` (beispielsweise `clone`), andere folgen aus Interfaces, die zu implementieren sind (beispielsweise `compareTo` aus dem Interface `Comparable`) und wieder andere sind bedingt durch Implementierungsvorgaben für den Codegenerator. Dazu gehören Funktionalitäten für die Protokollausgabe wie oben beschrieben, Speicherung, Fehlerbehandlung und zusätzliche Funktionen, die zur Bearbeitung von Tests hilfreich sind.

Tabelle 5.7.: *Klassen:* Umsetzung einer Klasse

Gerade für den Einsatz in Testumgebungen sind unter Umständen eine Reihe weiterer Methoden und Datenstrukturen für eine Klasse zu generieren. Bei der Generierung solcher uniformen Methoden für Implementierung und Tests kann ein Codegenerator wertvolle Dienste leisten.

Eine der wenigen und eher selten gewählten Alternativen zu der hier beschriebenen Abbildung sei dennoch erwähnt. Sie verzichtet darauf, das Typsystem der Zielsprache Java zu nutzen und legt stattdessen Attribute als Abbildung des Attributnamens auf den Wert mit dem `HashMap (String, Object)` ab. Es ist dann im Prinzip ausreichend, eine einzige Java-Klasse in der in Abbildung 5.8 dargestellten Form zu realisieren, die zwar einiges an zusätzlicher Flexibilität mit sich bringt, aber ineffizienter ist. Eine ähnliche Form wird zum Beispiel zur Ressourcen-Verwaltung von Parametern verwendet.

Java

```
class Chameleon { ...
  // Träger aller Attribute
  HashMap(String,Object) attributes;

  public Object get(String attributeName) {
      return attributes.get(attributeName);
  }

  // Typprüfung: feststellen, ob bestimmte Attribute vorhanden sind
  public boolean isInstanceOf(Set<String> attributeNames) {
      return attributes.keySet().containsAll(attributeNames);
  }
}
```

Abbildung 5.8. Dynamische Verwaltung von Attributen

5.1.7 Objekterzeugung

Ein letzter interessanter Punkt im Kontext der Codeerzeugung für Klassen ist das Management ihrer Objekte. Dazu gehört beispielsweise die Erzeugung von Objekten, die Verwaltung und der effiziente Zugriff auf einzelne Objekte oder das Speichern und Laden von Datenbanken. Verwaltungstätigkeiten werden oft so genannten „Management-Objekten“ auferlegt, die neben einer Sammlung der im Speicher befindlichen Objekte die transaktionsgesteuerte Abbildung auf die Datenbank und den effizienten Zugriff geladener Objekte erlauben. Von all diesen Tätigkeiten soll nachfolgend nur die Objekterzeugung in Java diskutiert werden, da sie unter anderem für Tests instrumentierbar sein muss.

Die in den Coderümpfen verwendete Form des `new Class(...)` kann bei der Codeerzeugung durch den Aufruf geeigneter Factory-Methoden umgesetzt werden. Dies erhöht die Flexibilität bei der Codeerzeugung beträchtlich, da so Unterklassen verwendet oder in automatisierten Tests Dummies eingesetzt werden können.[10] Dieses Verfahren basiert auf dem Entwurfsmuster *Abstract Factory* aus [GHJV94].

Objekterzeugung: Objekte mit einer Factory erzeugen	
Erklärung	Im Quellcode wird die Objekterzeugung mit dem `new`-Konstrukt vorgenommen. Der generierte Code enthält stattdessen Factory-Aufrufe. Eine Standard-Factory wird generiert und kann durch Bildung von Unterklassen auf spezifische Situationen angepasst werden.
Objekterzeugung	`new` *Class*`(`*Arguments*`)` ⇓ `Factory.new`*Class*`(`*Arguments*`)` • Die generierte Klasse `Factory` besitzt eine statische Methode `new`*Class* die das neue Objekt erzeugt. • Das Attribut `f` wird bei der Systeminitialisierung standardmäßig belegt, darf aber überschrieben werden. • Überschreiben von `create`*Class* erlaubt Erzeugung von Objekten aus Subklassen, Singletons, Management von Objektmengen und mehr.

(Fortsetzung auf nächster Seite)

[10] Dummies simulieren ein Objekt, ohne die Funktionalität tatsächlich zu implementieren. Dies ist für die Simulation einer Interaktion mit der Systemumgebung genauso geeignet wie für Komponentengrenzen.

(Fortsetzung von Tabelle 5.9.: Objekterzeugung: Objekte mit einer Factory erzeugen)

Klasse `Factory`

⇓

```
public class Factory { ...                                   Java
  public static initFactory() {
    f = new Factory();
  }

  // für jede Klasse Class
  public static Class newClass(Arguments) {
    return f.createClass(Arguments);
  }

  // erlaubt Überschreiben obiger statischen Methode
  protected static Factory f;

  protected Class createClass(Arguments) {
    return new Class(Arguments);
  }
}
```

- Entsprechende Factory-Methoden werden für jede Klasse des Systems generiert.
- Mehrere Factory-Methoden für dieselbe Klasse mit unterschiedlichen Parametersätzen werden erzeugt, wenn es entsprechende Konstruktoren gibt.
- Eine Aufteilung der Factory in mehrere Klassen, zum Beispiel entsprechend einer Subsystem-Struktur, kann vorgenommen werden, muss dann aber vom Generator-Skript gesteuert werden.

Alternative

Es ist unter anderem möglich, statt einem einzelnen Attribut `f` für mehrere Gruppen von zu erzeugenden Klassen beziehungsweise sogar für jede Klasse ein eigenes Attribut einzusetzen, so dass die Generierung von Objekten individuell angepasst werden kann:

(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 5.9.: Objekterzeugung: Objekte mit einer Factory erzeugen)

Java

```
public class Factory { ...
  public static initFactory() {
    fClass = new Factory(); ... // für jede Klasse
  }

  // für jede Klasse Class
  public static Class newClass(Arguments) {
    return fClass.createClass(Arguments);
  }

  protected static Factory fClass; ... // für jede Klasse

  protected Class createClass(Arguments) {
    return new Class(Arguments);
  }
}
```

wobei Aufrufe wieder so transformiert werden:

```
new Class(Arguments)
⇓
Factory.newClass(Arguments)
```

Tabelle 5.9.: *Objekterzeugung:* Objekte mit einer Factory erzeugen

Mehrstufige Übersetzung

Die exemplarisch diskutierten Varianten zur Umsetzung von Klassendiagrammen zeigen die hohe Bandbreite an möglichen Generierungsformen. Wie bereits diskutiert folgt daraus, dass die Codegenerierung eine grosse Flexibilität benötigt, um die jeweils notwendigen Aufgaben zu erfüllen. Ein Weg, die Flexibilität zu steigern, ist die Möglichkeit, aus mehreren Templates oder Skripten auszuwählen. Darüber hinaus nutzen sich die Templates gegenseitig, indem zum Beispiel Assoziationen zunächst in Attribute transformiert und diese dann durch Zugriffsmethoden gekapselt werden. Die in diesem Abschnitt gezeigten Regeln zur Transformation von Konzepten der Klassendiagramme in Java-Code sind daher nicht unabhängig voneinander. Abbildung 5.10 zeigt die Abhängigkeiten der Transformationsregeln.

Dabei sind nur die explizit definierten Regeln beschrieben, aber es sollte für ein geeignetes Framework weitere Transformationsregeln geben, die durch weitere Templates festgelegt werden. Die Auswahl der Alternativen ist manchmal durch den Kontext oder Eigenschaften des übersetzen Konzepts vorgegeben (wie hier zum Beispiel bei den Assoziationen) oder kann durch Einstellungen des Generators gesteuert werden (wie zum Beispiel bei den abgeleiteten Attributen).

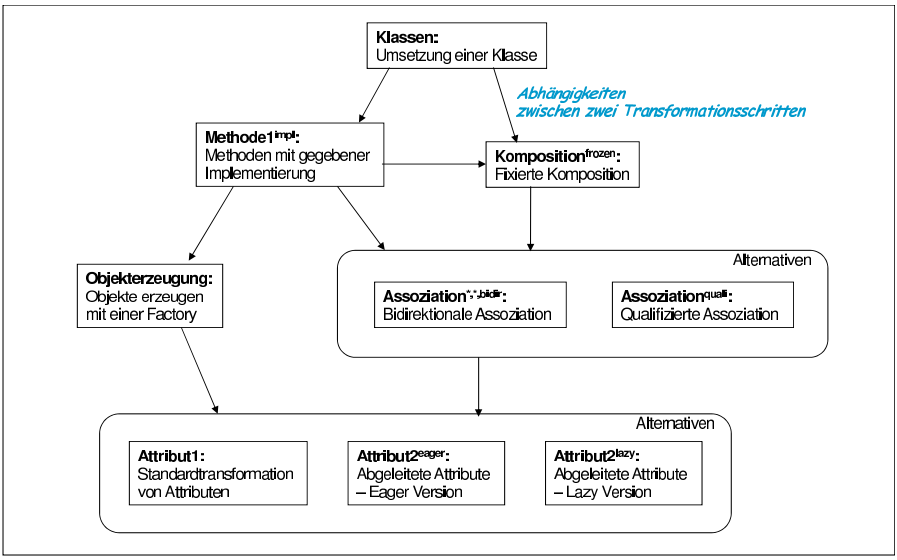


Abbildung 5.10. Abhängigkeiten der behandelten Transformationsregeln zur Codegenerierung aus Klassendiagrammen

5.2 Übersetzung von Objektdiagrammen

Die vollständige Beschreibung zur Generierung von Code aus der UML/P würde den Rahmen dieses Buchs sprengen. Deshalb werden in den folgenden Abschnitten einige der interessantesten Aspekte der Transformation weiterer Diagramm- und Textarten in Java-Code erläutert, ohne alle Details zu diskutieren.

Objektdiagramme können auf zwei Arten verwendet werden. Zum einen können Objektdiagramme konstruktiv eingesetzt werden, um damit Objektstrukturen zu erzeugen. Diese Funktionalität kann sowohl im Produktionssystem als auch zur Darstellung von Objektstrukturen, auf denen automatisierte Tests stattfinden sollen, verwendet werden. Zum anderen werden Objektdiagramme als Prädikate eingesetzt, um zu prüfen, ob eine bestimmte Objektstruktur vorhanden ist. In Abschnitt 4.4, Band 1 wurden diese Verwendungsarten bereits vom methodischen Standpunkt aus beleuchtet. Dieser Abschnitt wird deshalb Aspekte der Codegenerierung aus Objektdiagrammen diskutieren. Dabei sei zunächst die in Kapitel 4, Band 1 vorgenommene Integration von Objektdiagrammen und OCL vernachlässigt.

5.2.1 Konstruktiv eingesetzte Objektdiagramme

Die Transformation eines Objektdiagramms in Funktionalität zum konstruktiven Aufbau von Objektstrukturen erfolgt gesteuert durch ein Skript beziehungsweise durch Merkmale, die dem Objektdiagramm angefügt werden.

Dabei sind einige Parameter festzulegen, die bei der Codegenerierung wichtig sind:

1. Die Klasse, der die Methode zur Erzeugung der Objektstruktur zugeordnet wird. Ist im Diagramm ein Objekt eindeutig ausgezeichnet, so kann dies entfallen.
2. Der Name der zu erzeugenden Methode. Als Default wird `setup`*DiagramName* verwendet, wenn dieser Name eindeutig ist.
3. Die Objekte des Diagramms, die bereits existieren und als Parameter übergeben werden. In der Regel ist dies keines oder eines, das als übergeordnetes Objekt bereits erzeugt wurde.
4. Treten im Objektdiagramm freie Variablen auf, so werden sie ebenfalls als Parameter für die generierte Methode interpretiert.

Da die Ausgangssituation und die damit jeweils bereits existenten Objekte unterschiedlich sein können, kann es sinnvoll sein, mehrere Methoden aus einem Objektdiagramm zu generieren. Diese können sich durch die Signatur oder, falls dies nicht eindeutig ist, auch durch die Methodennamen unterscheiden.

Die Verwendung freier Variablen und unbesetzter Attribute als Parameter der generierten Funktion erlaubt es, Objektdiagramme wie in Abschnitt 4.2.2, Band 1 diskutiert als *Muster* mit *prototypischen* Objekten zu interpretieren, die eine mehrfache Instanziierung mit unterschiedlichen Inhalten erlauben.

Bei der Generierung der `setup`-Methoden gibt es mehrere Aspekte, die zu beachten sind. Für die Erzeugung eines Objekts ist die Verwendung eines Konstruktors notwendig. Idealerweise sollte ein Konstruktor ohne Parameter zur Verfügung stehen, der nur das leere Objekt erzeugt. Steht ein solcher Konstruktor nicht zur Verfügung, so kann im Testsystem ein solcher generiert werden. Im Produktionssystem muss allerdings auf einen existenten Konstruktor zurückgegriffen werden, der entsprechend ausgezeichnet wurde.

Ein weiteres Problem ist die Besetzung der Attribute entsprechend der im Objektdiagramm vorgegebenen Werte. Dafür sollten geeignete Hilfsfunktionen zur Verfügung stehen oder direkter Zugriff auf die Attribute erfolgen. Die in Abschnitt 5.1.1 diskutierten `set`-Methoden sind dafür nur partiell geeignet, da sie unter Umständen zusätzliche Funktionalität beinhalten.

Im Prinzip können dafür auch Konstruktoren mit Parametern weiterhelfen, wenn die Zuordnung zwischen dem Parameter und dem zu besetzenden Attribut aus der Konstruktordefinition eindeutig hervorgeht.[11]

Ein Objektdiagramm kann grundsätzlich unvollständig sein, indem etwa die Klasse eines Objekts nicht angegeben ist oder manchen Attributen kein Wert zugewiesen wurde. Zum einen können unbesetzte Attribute als freie

[11] Das ist zum Beispiel dann der Fall, wenn der Konstruktor wie in Abschnitt 5.1 beschrieben ebenfalls generiert wurde.

Variable verstanden werden und als Parameter in die generierte Methode aufgenommen werden. Ist dies nicht gewünscht, so sind abhängig von der Art des Einsatzes im Testsystem, zur Simulation oder im Produktionssystem verschiedene Strategien zur Behandlung unbesetzter Attribute möglich. Im Testsystem wird eine Failure-Strategie genutzt: Unbesetzte Attributwerte sollten für den getesteten Systemablauf keine Rolle spielen und der Zugriff darauf mit einem sofortigen Scheitern des Tests reagieren. Entsprechendes Verhalten kann in die `get`-Funktionen integriert werden. Bei der Simulation ist die bereits in Abschnitt 5.1.2 diskutierte Vorgehensweise sinnvoll, fehlende Attributwerte während des Simulationslaufs interaktiv zu erfragen oder mit Default-Werten zu arbeiten. Bei der Codegenerierung für das Produktionssystem ist schließlich eine vollständige Definition der Objekte im Objektdiagramm Voraussetzung. Dies verhindert Unachtsamkeiten bei der Definition von Objektdiagrammen und gibt so dem Entwickler Zutrauen in die Zuverlässigkeit des modellierten Systems.

Nicht alle im Objektdiagramm formulierbaren Angaben werden bei konstruktiver oder auch der später diskutierten prädikativen Codegenerierung verwendet. Sichtbarkeiten, die Information über Kompositionalität eines Links, Merkmale wie {`frozen`} und ähnliches mehr bedürfen keiner Umsetzung in den hier diskutierten Code, sondern werden mit den in Klassendiagrammen vorhandenen Informationen abgeglichen oder in Tests eingesetzt.

Ein alternativer, mit Methodenspezifikationen verbundener Ansatz zum konstruktiven Einsatz von Objektdiagrammen ist bereits in Abschnitt 4.4.7, Band 1 diskutiert. Er nutzt ein Objektdiagramm in der Nachbedingung eines Konstruktors beziehungsweise einer Initialisierungsmethode, das nach denselben Prinzipien wie den hier gezeigten in konstruktiven Code umgesetzt werden kann.

5.2.2 Beispiel einer konstruktiven Codegenerierung

Statt nun die Transformationsregeln für jedes Modellelement des Objektdiagramms zu diskutieren sollen diese anhand des in Abbildung 5.11 gezeigten Objektdiagramms beispielhaft erläutert werden. Dieses Diagramm beschreibt einen Ausschnitt der initialen Objektstruktur des Applets im Auktionssystem und ist eingebettet in eine OCL-Methodenspezifikation für die Initialisierungsfunktion `init()`. Der generierte Code ist in Abbildung 5.12 dargestellt, wobei hier und in den folgenden Beispielen vereinfachend angenommen wird, dass der direkte Zugriff auf die Attribute erst noch transformiert wird:

Aus dem zugehörigen (hier nicht wiedergegebenen) Klassendiagramm kann das System ableiten, dass die Links durch die Attribute `loginPanel` und `httpServerProxy` in der Klasse `WebBidding` realisiert werden. Für die Klassen `HttpServerProxy` und `LoginPanel` wird ein geeignet parametrisierter Konstruktor verwendet, der die Besetzung der angegebenen

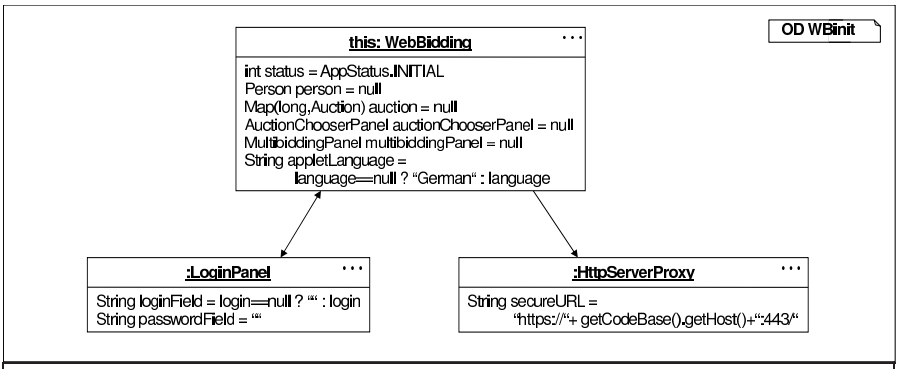


Abbildung 5.11. Objektdiagramm zur Initialisierung einer Struktur

```
class WebBidding { ...                                          Java
  public void init() {
    // aus dem let-Konstrukt
    String language = getParameter("language");
    String login    = getParameter("login");
    // aus Objekt this
    status = AppStatus.INITIAL;
    person = null;
    auction = null;
    auctionChooserPanel = null;
    multibiddingPanel = null;
    appletLanguage = language==null ? "German" : language;

    // aus Objekt :LoginPanel
    setLoginPanel(new LoginPanel(login==null ? "":login,""));

    // aus Objekt :HttpServerProxy
    setHttpServerProxy(
      new HttpServerProxy(
          "https://"+getCodeBase().getHost()+":443/"));
  }
}
```

Abbildung 5.12. Aus dem Objektdiagramm generierte `init()`-Funktion

Attribute vornimmt.[12] Die Links werden durch `set`-Methoden besetzt, die auch die korrekte Besetzung der Rückrichtung sichern.

[12] Wenn kein geeigneter Konstruktor existiert, dann kann ein Codegenerator einen solchen Konstruktor generieren.

5.2.3 Als Prädikate eingesetzte Objektdiagramme

Der Einsatz eines Objektdiagramms als Prädikat, das prüft, ob eine bestimmte Objektstruktur vorliegt und die im Objektdiagramm angegebenen Werte übereinstimmen, wird in eine boolesche Methode transformiert. Ähnlich wie bei der konstruktiven Variante steuern mehrere Parameter den zu erzeugenden Code:

1. Die Klasse, der die boolesche Methode zugeordnet wird. Ist im Diagramm ein Objekt eindeutig ausgezeichnet, zum Beispiel durch den Namen `this:Classname`, so kann dies entfallen. Alternativ kann die Methode als statisch definiert und/oder einer Testklasse zugeordnet werden.
2. Der Name der zu erzeugenden Methode. Ist der Name nicht gegeben, so wird als Default `isStructuredAs`*DiagramName* verwendet.
3. Die Objekte des Diagramms, die als Ausgangsobjekte bereits identifiziert sind und deshalb als Parameter übergeben werden. In der Regel ist dies ein einzelnes Objekt, das als eine Art Master für die Objektstruktur gilt.
4. Treten im Objektdiagramm freie Variablen auf, so werden sie im Normalfall nicht weiter beachtet. Sollen diese Variablen aber bestimmte Werte annehmen, so werden die Variablen ebenfalls als Parameter für die generierte Methode interpretiert.

Im Gegensatz zur konstruktiven Variante eines Objektdiagramms kann ein prädikativ eingesetztes Diagramm in mehrerer Hinsicht unvollständig sein. Attribute und Attributwerte dürfen ebenso weggelassen werden wie die Klassen der dargestellen Objekte. Auch Eigenschaften mit einer zeitlichen Implikation, wie etwa das Merkmal `{frozen}` für Links, können nicht in einem Prädikat über einen Zustand geprüft werden. Dazu ist zusätzliche Infrastruktur nötig, die dies entweder konstruktiv sichert, indem keine Methoden zur Modifikation eines Links angeboten werden, oder zur Laufzeit prüft, indem der ursprüngliche Zustand des Links in einer Kopie aufgehoben wird.

In Abschnitt 4.3, Band 1 wurde die Bedeutung eines Objektdiagramms als Prädikat im Kontext der Integration mit OCL-Bedingungen bereits ausführlich diskutiert. Dabei wurde festgestellt, dass ein Objektdiagramm grundsätzlich als OCL-Bedingung dargestellt werden kann. In genau dieser Bedeutung werden prädikative Objektdiagramme in entsprechende boolesche Methoden, die mit dem in Abschnitt 3.4.1, Band 1 eingeführten Stereotyp «query» markiert sind, übersetzt. Die so generierten Methoden können genau wie die Referenz auf das Objektdiagramm bei Invarianten, Vor- und Nachbedingungen von Methoden und von Transitionen in Statecharts, aber auch in Java-Rümpfen des Produktionscodes eingesetzt werden.

Für die Verwendung im Produktionscode ist jedoch auf die Effizienz der Umsetzung zu achten. Wie in Abschnitt 4.3, Band 1 diskutiert, wirken anonyme Objekte des Objektdiagramms als existenzquantifiziert. In derselben

Weise werden benannte Objekte behandelt, die jedoch nicht bereits als Parameter an das boolesche Prädikat übergeben werden. Diese Objekte werden vom Prädikat selbst gesucht, indem die entsprechenden Assoziationen geprüft werden. Die Belegung der freien Objekte erfolgt in einer dem Struktur-Matching der Graph Grammatiken [Roz99, EEKR99] analogen Form.

Bei mengenwertigen Assoziationen kann eine derartige Suche von linearer Komplexität sein und sollte deshalb vermieden werden. Möglichkeiten zur Verbesserung der Situation bieten der Einsatz eines Qualifikators bei der Assoziation oder die explizite Übergabe gesuchter Objekte als Parameter, wenn diese aus dem Kontext effizient ermittelt werden können.

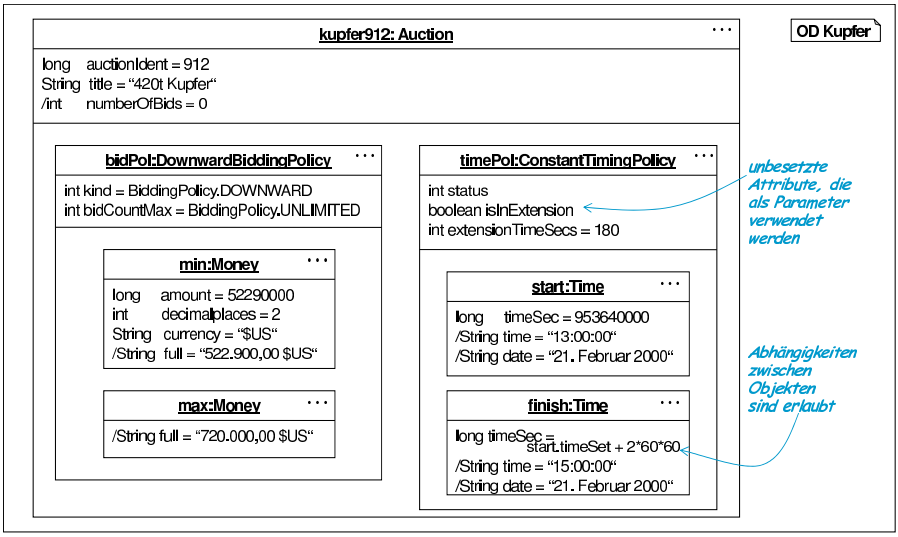


Abbildung 5.13. Für ein Prädikat bestimmtes Objektdiagramm

Anhand des aus dem Auktionssystem stammenden und in Abbildung 5.13 dargestellten Objektdiagramms wird die Transformation in ein Prädikat illustriert. Der für den Einsatz im Testsystem generierte Code ist in Abbildung 5.14 dargestellt.

Nach einer Phase der Belegung von Objektnamen werden alle Attribute geprüft. Dabei dürfen Attributwerte aufeinander Bezug nehmen. Werden statt der im (hier nicht dargestellten) Klassendiagramm angegebenen Typen echte Unterklassen angegeben, so wird geprüft, ob das entsprechende Objekt tatsächlich zu dieser Unterklasse gehört.

Anhand der Länge des in Abbildung 5.14 dargestellten Codes ist ersichtlich, dass eine Darstellung im Objektdiagramm kompakter und übersichtlicher ist, also dem Modellierer insbesondere bei der schnellen Erfassung eines Überblicks und der Suche einzelner Werte Vorteile bringt.

Java

```
class Auction { ...
  public static boolean isStructedAsKupfer(Auction kupfer912,
                          int status, boolean isInExtension) {
    // Namen festlegen (optimierbar)
    BiddingPolicy bidPol = kupfer912.bidPol;
    TimingPolicy timePol = kupfer912.timePol;
    Money min = bidPol.min;
    Money max = bidPol.max;
    Time start = timePol.start;
    Time finish = timePol.finish;

    return
     // Hauptobjekt
     kupfer912.auctionIdent == 912 &&
     kupfer912.title.equals("420t Kupfer") &&
     kupfer912.numberOfBids == 0 &&

     // BiddingPolicy
     bidPol instanceof DownwardBiddingPolicy &&
     bidPol.kind == BiddingPolicy.DOWNWARD &&
     bidPol.bidCountMax == BiddingPolicy.UNLIMITED &&

     // Money Objekte
     min.amount == 52290000 &&
     min.decimalplaces == 2 &&
     min.currency.equals("$US") &&
     min.full.equals("522.900,00 $US") &&
     max.full.equals("720.000,00 $US") &&

     // TimingPolicy
     timePol instanceof ConstantTimingPolicy &&
     timePol.status == status &&
     timePol.isInExtension == isInExtension &&
     timePol.extensionTimeSecs == 180 &&

     // Times
     start.timeSec == 953640000 &&
     start.time.equals("13:00:00") &&
     start.date.equals("21. Februar 2000") &&
     finish.timeSec == start.timeSec + 2*60*60 &&
     finish.time.equals("15:00:00") &&
     finish.date.equals("21. Februar 2000") ;
  }
}
```

Abbildung 5.14. Aus Objektdiagramm generiertes Prädikat

Als Erweiterung beziehungsweise Alternative bei der Codegenerierung kann eine weitere Methode mit dem Namen `isExactlyStructuredAs`*DiagramName* generiert werden. Diese Methode kann zusätzlich sichern, dass die Objektstruktur nur die im Diagramm angegebenen Objekte enthält. Das ist besonders bei mehrwertigen und optionalen Assoziationen von Interesse und kann zum Beispiel durch einen geeigneten Stereotyp ≪complete≫ für Objektdiagramme gesteuert werden. Tabelle 5.15 gibt eine knappe Einführung zu diesem Stereotyp.

Stereotyp ≪complete≫	
Modell-element	Objektdiagramm.
Motivation	Die Bedeutung eines Objektdiagramms als Prädikat ist normalerweise so festgelegt, dass die explizit angegebenen Eigenschaften erfüllt sein müssen. Weitere, im Diagramm nicht angegebene Objekte können existieren.
Rahmen-bedingung	In einem mit ≪complete≫ markierten Objektdiagramm müssen alle Attribute und Links angegeben sein. Geordnete Assoziationen sind vollständig darzustellen.
Wirkung	Der Stereotyp ≪complete≫ fordert, dass keine weiteren Objekte in der angegebenen Objektstruktur existieren. Das angegebene Objektdiagramm ist also eine vollständige Darstellung der Objektstruktur. Damit kann die Methode `isExactlyStructuredAs`*DiagramName* generiert werden, die diese Vollständigkeit zusätzlich zur Erfüllung der angegebenen Eigenschaften des Objektdiagramms prüft.

Tabelle 5.15.: Stereotyp ≪complete≫

5.2.4 Objektdiagramm beschreibt Strukturmodifikation

Eine weitere interessante Form des Einsatzes von Objektdiagrammen ergibt sich aus der Kombination beider Einsatzformen. Dabei wird eine existente Objektstruktur auf das Vorhandensein der im Objektdiagramm beschriebenen Objekte geprüft, die fehlenden Objekte generiert und die falsch besetzten Attribute modifiziert. Derartige Methoden erhalten als Namen `adaptTo`*DiagramName*. Mit diesen Methoden kann eine bereits vorhandene Objektstruktur in Abhängigkeit des aktuell gewünschten Zustands umgebaut werden. Damit lassen sich Objektdiagramme zum Beispiel als Zustandsinvarianten im Statechart oder zur Adaption der jeweiligen Objektstruktur in der Entry-Aktion von Zuständen einsetzen. Wird beispielsweise das in Abbildung 5.11 gegebene Objektdiagramm (ohne Einbettung in die

dort stehende OCL-Bedingung) in dieser Form eingesetzt, so wird die in Abbildung 5.16 dargestellte Methode erzeugt.

```
class WebBidding {                                                    Java

  public void adaptToWBinit(String language, String login) {
    // aus Objekt this
    status = AppStatus.INITIAL;
    person = null;
    auction = null;
    auctionChooserPanel = null;
    multibiddingPanel = null;
    appletLanguage = language==null ? "German" : language;

    // aus Objekt :LoginPanel
    if(loginPanel != null &&
               loginPanel instanceof LoginPanel) {
      loginPanel.loginField = (login==null) ? "" : login;
      loginPanel.passwordField = "";
    } else {
      setLoginPanel(new LoginPanel(
               login==null ? "" : login, ""));
    }

    // aus Objekt :HttpServerProxy
    if(httpServerProxy != null &&
               httpServerProxy instanceof HttpServerProxy) {
      httpServerProxy.secureURL =
               "https://"+getCodeBase().getHost()+":443/";
      httpServerProxy.connectStatus =
               HttpServerProxy.NOT_CONNECTED;
      httpServerProxy.lastConnectionTime = Time.now();
    } else {
      setHttpServerProxy(
        new HttpServerProxy(
               "https://"+getCodeBase().getHost()+":443/"));
    }
}}
```

Abbildung 5.16. Aus Objektdiagramm generierte Adaptionsmethode

Fehlt ein Objekt oder hat es den falschen Typ, so wird es neu erzeugt. Ist das Objekt bereits vorhanden, so werden seine Attribute in der gewünschten Form modifiziert. Deshalb werden sowohl Konstruktoren, die in diesem Beispiel bereits Attributwerte als Parameter enthalten, als auch `set`-Methoden zur Attributbesetzung verwendet.

Die Eindeutigkeit des entlang eines Links zu identifizierenden Objekts wie zum Beispiel dem anonymen Objekt `:LoginPanel` ist nur bei Assoziationen der Kardinalität „`1`“ oder „`0..1`“ gegeben. Bei mengenwertigen Assoziationen ist ein Vergleich mit allen vorhandenen Objekten durchzuführen, der auch dessen Attribute einbezieht. Findet sich kein Objekt in der Objektstruktur, das dem im Diagramm angegebenen prototypischen Objekt entspricht, so wird in diesem Fall keines der vorhandenen Objekte angepasst, sondern ein neues Objekt erzeugt. Dadurch vergrößert sich die Menge der Links entsprechend. Der Nachteil dieser Methode ist allerdings, dass die Löschung unerwünschter Objekte nicht dargestellt werden kann. Außerdem kann diese Methode ineffizient werden, wenn zum Beispiel mit dem in Abbildung 3.28 gegebenen Objektdiagramm (ohne die OCL-Bedingung) und der daraus generierten und in Abbildung 5.17 dargestellten Methode hundert Personen einzeln angelegt werden.

```
class Auction { ...                                          Java/P

  public void adaptToNPersons(Auction test32, int x) {
    // setze Objekt test32
    test32.auctionIdent = 32;
    test32.title = "Testauktion";

    // gibt es p:Person?
    Person p = null;
    for(Iterator<Person> ip = participants.iterator();
                                  ip.hasNext() && p==null; ) {
      Person pit = ip.next();
      if(pit.personIdent == 1000+x &&
         pit.login == "log" +x &&
         pit.name == "Tester " +x &&
         pit.isActive == (x%2 == 0)) {
             p = pit;
      }
    }
    // p:Person anlegen
    if(p == null) {
      p = new Person(); // Default-Konstruktor
      p.personIdent = 1000+x;
      p.login = "log" +x;
      p.name = "Tester " +x;
      p.isActive = (x    }
  }
}
```

Abbildung 5.17. Adaptionsmethode mit Suche in $\star$-Assoziation

5.2.5 Objektdiagramme und OCL

Ein wesentliches Mittel zur Steigerung der Ausdrucksmächtigkeit von Objektdiagrammen ist die ab Abschnitt 4.3, Band 1 durchgeführte Integration mit der OCL. Die Transformation eines um OCL-Bedingungen erweiterten Objektdiagramms in konstruktiven Code beziehungsweise in ein Prädikat hängt daher von der Umsetzbarkeit der OCL-Bedingungen ab. Die Umsetzung und die Ausführbarkeit von OCL-Bedingungen wurde bereits in Abschnitt 4.1.2 diskutiert.

Zu beachten ist aber auch, dass die in der vorangegangenen Diskussion erwähnte lineare Suchkomplexität für existenzquantifizierte Objekte polynomial ansteigen kann, wenn über mehrere mengenwertige Assoziationen navigiert wird und die so erreichten Objekte über eine OCL-Bedingung verknüpft sind.

5.3 Codegenerierung aus OCL

Auf Basis der bisher beschriebenen Möglichkeiten, die OCL als Spezifikationssprache für UML/P-Programme zu nutzen, gibt es zum Beispiel die Möglichkeit, die OCL zur Modellierung von Invarianten in Datenbanksystemen einzusetzen [DH99] oder mit der OCL die Geschäftslogik zu beschreiben [DHL01]. Die dabei zugrunde liegende Semantik der OCL ist in ihrer Essenz mit der bisher diskutierten Bedeutung der OCL identisch, jedoch ist die Einsatzform und damit die Abbildung der OCL in ausführbaren Code wesentlich anders. Statt Klassen und Objekten stehen in relationalen Datenbanken *Entitäten* und *Zeilen* zur Verfügung, über denen OCL-Ausdrücke interpretiert werden.

Mehrere, teils prototypische Werkzeuge bieten bereits Realisierungen für OCL-Codegeneratoren an. In [HDF00] wird eine modulare Architektur für die OCL vorgestellt, die als Grundlage für eine Integration mit anderen UML-Werkzeugen gut geeignet ist [BS01a, BBWL01]. Eine der OCL verwandte Sprache, genannt „Java Interface Specification Language“ (JISL), wird in [MMPH99] zur Methodenspezifikation vorgestellt und diskutiert, wie die Ausführbarkeit der Spezifikation durch eine Transformation nach Java vorgenommen werden kann.

Weil zwischen OCL/P und Java aufgrund der syntaktischen und semantischen Nähe relativ viele Konzepte eindeutig nach Java umgesetzt werden, sollen nachfolgend vor allem die interessanten OCL-Konstrukte diskutiert werden. Dazu gehören unter anderem die Umsetzung der Kontextdefinition, der OCL-Logik, der Komprehension, der Navigation, der Quantoren und der Spezialoperatoren.

Viele, jedoch nicht alle der nachfolgend beschriebenen Codestücke lassen sich in wiederverwendbare Methoden kapseln. Es wird hier auf die Darstellung der Kapselung verzichtet, da dies nur den dargestellten Code

vergrößert. Bei einer manuellen Verwendung dieser Codestücke (als eine Art von Entwurfsmuster) ist allerdings eine Kapselung zu empfehlen.

5.3.1 OCL-Aussage als Prädikat

Obwohl OCL-Bedingungen, wie in Abschnitt 4.1.2 beschrieben, in sehr eingeschränkter Form in Implementierungscode umgesetzt werden können, hat eine OCL-Aussage eine natürliche Bedeutung als Prädikat. Deshalb werden OCL-Aussagen, wie etwa

```
context Auction a inv Bidders1:
  a.activeParticipants <= a.bidder.size
```


in boolesche Methoden übersetzt. Dabei gibt es zwei Varianten. Die Interpretation als Invariante produziert:

```
public static boolean invariantBidders1() {
  boolean res = true;
  Set<Auction> auctions = Auction.getAllInstances();
  for(Iterator<Auction> ia = auctions.iterator();
                             ia.hasNext() && res; ) {
   Auction a = ia.next();
   res &= a.activeParticipants <= a.bidder.size;
  }
  return res;
}
```


Die generierte Methode prüft die Invariante `Bidders1` für alle Objekte der Klasse `Auction`. Um dies zu ermöglichen, ist eine Infrastruktur zur Verwaltung der Menge aller Auktionsobjekte erforderlich, die zum Beispiel durch die Methode `getAllInstances` zur Verfügung gestellt wird.

Tatsächlich ist für praktische Anwendungen ein Einsatz der OCL-Bedingung in der oben beschriebenen Form ungünstig. Im laufenden System werden seit der letzten Prüfung der Invariante normalerweise nur relativ wenige Auktionsobjekte modifiziert worden sein. Deshalb ist es unter Umständen günstig, zusätzliche Infrastruktur zu investieren, um die Invariantenprüfung effizienter zu gestalten. Dies kann erreicht werden, indem die Prüfung der Invariante explizit an bestimmten Stellen gefordert wird. Dann sind aber nicht alle, sondern nur bestimmte Auktionsobjekte zu testen. Die dafür geeignete Methode ist

```
public static boolean checkBidders1(Auction a) {
  return a.activeParticipants <= a.bidder.size();
}
```


und ihre der Klasse `Auction` zugeordnete Variante

```
class Auction ... {                                          Java
  public boolean checkBidders1() {
    return this.activeParticipants <= this.bidder.size();
  }
```

Alle diese Formen eignen sich zum Einsatz in Tests, in denen die Prüfung von Invarianten explizit gefordert werden kann. Zu beachten ist, dass nach Konvention die vollständige Invariante das Präfix `invariant` besitzt, während `check` für die Einzelfall-Prüfung verwendet wird. Die explizite Übergabe des Kontexts einer OCL-Aussage in Form der zu prüfenden Objekte wird normalerweise durch das Schlüsselwort **import** vorgenommen. Deshalb werden die letzten beiden Methoden auch aus folgender Bedingung erzeugt:

```
import Auction a inv Bidders1:                               OCL
  a.activeParticipants <= a.bidder.size
```

Da in der Praxis ein mit dem Schlüsselwort **context** geschlossener Kontext oft auch auf einzelne Objekte angewandt werden soll, werden daraus sowohl eine geschlossene als auch zusätzlich parametrisierte Methoden erzeugt. Besteht die Kontextangabe aus mehreren Objekten, so werden alle als Parameter verwendet. Folgende Tabelle 5.18 beschreibt die Transformation:

OCL-Kontext: Umsetzung des Kontexts einer Bedingung

Erklärung	Der Kontext wird in OCL explizit in Form von Objekten angegeben. Diese wirken als Parameter bei den daraus generierten booleschen Methoden.

Kontext	**context** *Classcontext* **inv** *Name*: *Body* (OCL)
	⇓
	(Java/P)
	public static boolean invariant*Name*() {
	boolean res = **true**;
	// Iteration nur einmal dargestellt
	Set<*Class*> s = *Class*.getAllInstances();
	for(Iterator<*Class*> it = s.iterator();
	it.hasNext() **&&** res;) {
	Class ob = it.next();
	res **&**= *Body′*;
	}
	return res;
	}
	public static boolean check*Name*(*Classcontext*) {
	Body′
	}

(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 5.18.: OCL-Kontext: Umsetzung des Kontexts einer Bedingung)

	• Die Iteration wird für jedes Element *Class ob* des Kontexts geschachtelt wiederholt. Entsprechend ergeben sich polynomiale Komplexitäten bei der Prüfung der vollständigen Invariante. • Der generierte Code ist in einer statischen Methode abgelegt und wird einer geeigneten Klasse zugeordnet. Besteht der Kontext aus nur einem Objekt, so kann zusätzlich eine nicht-statische Methode in der Klasse des Objekts generiert werden. • Die Methode `getAllInstances` wird nachfolgend beschrieben. • Das alternative Schlüsselwort `import` generiert nur die parametrisierten Formen. • Der Rumpf der Bedingung *Body* wird entsprechend der nachfolgenden Regeln in *Body'* umgesetzt.

Tabelle 5.18.: *OCL-Kontext:* Umsetzung des Kontexts einer Bedingung

5.3.2 OCL-Logik

Wie in Abschnitt 3.2.2, Band 1 besprochen ist die OCL-Logik zweiwertig und nutzt einen impliziten *Liftingoperator*, um undefinierte Ergebnisse als **false** zu interpretieren. Der mit ↑ bezeichnete Liftingoperator ist wegen seiner Eigenschaft ↑**undef**==**false** nicht (vollständig) implementierbar. Jedoch existiert eine bereits in Abschnitt 3.2.2, Band 1 diskutierte Umsetzung dieses Operators in Java, die Exceptions abfängt und so nur nichtterminierende Berechnungen nicht erkennen kann. Der Liftingoperator ist implizit bei allen booleschen Operationen der OCL zu finden, so dass die Umsetzung der booleschen Operationen **&&**, `||`, `!`, `implies` und `<=>` jeweils der Verwendung des Liftings bedarf.

OCL-Logik: Logikoperatoren	
Erklärung	Die zweiwertige Logik wird mittels Lifting des undefinierten Pseudowerts **undef** auf **false** umgesetzt.

(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 5.19.: OCL-Logik: Logikoperatoren)

Negation

```
... !a ...
```
⇓
```
boolean res;
try {
  res = a;
} catch(Exception e) {
  res = false;
}
... !res ...
```
Java

- Die Einbettung der Auswertung des booleschen Ausdrucks *a* in eine `catch`-Anweisung macht eine Zerschlagung des Kontexts von *a* notwendig. Der Teilausdruck *a* wird vorab berechnet und in der Ergebnisvariable `res` zwischengespeichert. Die Umordnung der Berechnung ist erlaubt, da Seiteneffekte in OCL-Ausdrücken nicht auftreten.
- Eine Nichtterminierung der Auswertung von *a* wird nicht erkannt.

Äquivalenz

```
... a <=> b ...
```
⇓
```
boolean resa, resb;
try {
  resa = a;
} catch(Exception e) {
  resa = false;
}
try {
  resb = b;
} catch(Exception e) {
  resb = false;
}
... (resa==resb) ...
```
Java

Weitere Operatoren

werden in analoger Form umgesetzt. Implikation `a implies b` wird auf `!a || b`, Konjunktion und Disjunktion werden auf die jeweiligen Java-Operatoren abgebildet.

Tabelle 5.19.: *OCL-Logik:* Logikoperatoren

Zur Umsetzung der OCL-Kontrollstrukturen können die entsprechenden Java-Kontrollanweisungen eingesetzt werden. Das **let**-Konstrukt führt lokale Variablen ein und fängt bei der Berechnung der Variablenwerte auftretende Exceptions ab.

5.3.3 OCL-Typen

Die OCL/P-Grunddatentypen stimmen mit denen aus Java überein, so dass eine Umsetzung nicht erforderlich ist. Auch der Vergleich der Grunddatentypen == sowie die arithmetischen Operationen werden unverändert übernommen.

Die Container `Set<X>`, `List<X>` und `Collection<X>`, die die OCL anbietet, stellen eine Teilmenge der Container aus Java dar. Deshalb können die OCL-Container nahezu unverändert nach Java übernommen werden. Die explizite Aufzählung wird im Wesentlichen auf einen Konstruktor und elementweises Hinzufügen abgebildet. Dabei wird vom Generator festgelegt, welche konkrete Mengen- oder Listenimplementierung verwendet wird.

Die Verwendung von Containern über Grunddatentypen bedarf einer besonderen Behandlung. Java ist im Gegensatz zur OCL nicht in der Lage, zum Beispiel `int`-Werte direkt in Containerstrukturen zu speichern. Stattdessen ist die objektifizierte Form „`Integer`“ zu verwenden. Der Typ `Set<int>` wird also nach `Set<Integer>` überführt. Das in Java vorhandene Boxing von Werten in ihre objektivierten Fassungen erlaubt es allerdings Anwendungsstellen unverändert zu lassen.

Da für Container nicht die Objektidentität, sondern der inhaltliche Vergleich von Interesse ist, wird der Vergleich == in eine eigens generierte Methode übersetzt, die Gleichheit auf Containern wie in Abschnitt 3.3.3, Band 1 beschrieben interpretiert.

Die Umsetzung der in der OCL/P eingeführten Komprehensionskonstrukte wird in der folgenden Tabelle 5.20 erklärt.

OCL-Komprehension: Umformung der Komprehension	
Erklärung	Die für Mengen und Listen verwendeten Komprehensionsformen werden in Java durch zusätzliche Infrastruktur ergänzt.

Aufzählung	OCL
	`Set{ a..b }`
	⇓
	Java/P
	`Set<Integer> res = ...`
	`for(int i = a; i <= b; i++)`
	`  res.add(i);`
	`}`
	`// Weiterverwendung von res`

(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 5.20.: OCL-Komprehension: Umformung der Komprehension)

<table>
<tr><td></td><td>

- Die Aufzählung wird durch eine Schleife realisiert, da Java keine Kompaktform anbietet.
- Die Variable `res` erlaubt die Weiterverwendung für den Aufbau einer Menge aus mehreren Aufzählungen, zum Beispiel der Form `Set{`*a*`,`*b*`..`*c*`,`*d*`..`*e*`}`.
- `res` ist zum Beispiel mit `new HashSet()` zu besetzen.

</td></tr>
<tr><td>Komprehension mit Generator</td><td>

OCL

```
Set{ expr | var in expr2 }
```

⇓

Java/P

```
Set<Class> res = new HashSet<Class>;
for(Iterator<Class2> it = expr2'.iterator();
                              it.hasNext(); ) {
  Class2 var = it.next();
  res.add(expr');
}
// Weiterverwendung von res
```

- Die Berechnung des OCL-Ausdrucks wird in mehrere Anweisungen mit Zwischenergebnis `res` aufgebrochen. Die Umordnung der Berechnung ist erlaubt, da OCL und auch seine Umsetzung seiteneffektfrei ist.
- Die Ausdrücke *expr* und *expr2* werden ebenfalls transformiert.
- *expr* hat den Typ *Class*.
- *expr2* hat den Typ `Set<`*Class2*`>`.

</td></tr>
<tr><td>Komprehension mit Filter</td><td>

OCL

```
Set{ expr | var in expr2, boolexpr }
```

⇓

Java/P

```
Set<Class> res = new HashSet<Class>;
for(Iterator<Class2> it = expr2'.iterator();
                              it.hasNext(); ) {
  Class2 var = it.next();
  if(boolexpr)
    res.add(expr');
}
// Weiterverwendung von res
```

</td></tr>
</table>

(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 5.20.: OCL-Komprehension: Umformung der Komprehension)

<table>
<tr><td></td><td>

- Wie in Abschnitt 3.3.2, Band 1 beschrieben, entfernt ein Filter *boolexpr* Elemente aus der Menge.
- Der Filter gewinnt seine Mächtigkeit in Kombination mit dem bereits formulierten Generator. Deshalb wird die Transformation mit angefügtem Generator dargestellt.

</td></tr>
<tr><td>Lokale Variablen</td><td>

OCL

```
Set{ expr | var in expr2, Type x = expr3 }
```

⇓

Java/P

```
Set<Class> res = new HashSet<Class>;
for(Iterator<Class2> it = expr2'.iterator();
                              it.hasNext(); ) {
  Class2 var = it.next();
  Type x = expr3;
  res.add(expr');
}
// Weiterverwendung von res
```

- Eine lokale Variablendefinition wirkt wie ein **let**-Konstrukt.

</td></tr>
<tr><td>Kombination</td><td>von Generatoren, lokalen Variablendefinitionen und Filtern innerhalb von Komprehensionen ist beliebig möglich. Bei mehreren Generatoren steigt jedoch die Komplexität schnell an.</td></tr>
</table>

Tabelle 5.20.: *OCL-Komprehension:* Umformung der Komprehension

Die Komprehension besitzt Verwandtschaft mit SQL-Statements, ist allerdings deutlich ausdrucksstärker. Eine Übersetzung von OCL in eine Datenbankanfrage kann zumindest teilweise die Effizienz der Datenbankanfragen nutzen.

5.3.4 Typ als Extension

Die OCL erlaubt die Verwendung von Typausdrücken wie `Auction` als Extension, die damit gleichzeitig die Menge der aktuell existierenden Objekte dieses Typs beschreibt. So kann zum Beispiel formuliert werden:

```
forall a in Auction: a.person.size < 500
```

OCL

Um eine Überprüfung dieser Bedingung zur Laufzeit zu erlauben, ist der Zugriff auf alle zu einem Zeitpunkt existenten Objekte des Typs `Auction` erforderlich. Deshalb ist eine entsprechende Infrastruktur zur Verwaltung der

Extensionen der Objekte zur Verfügung zu stellen. Diese kann auf Basis von `WeakHashMap`s realisiert sein, die mit der Garbage Collection zusammenarbeiten.

Eine boolesche Variable `OCL.oclmode` beschreibt, ob das System gerade eine OCL-Bedingung auswertet und daher eventuell neu angelegte Objekte nicht zur Extension hinzuzurechnen sind[13].

In der Verwaltung sind zusätzliche Maßnahmen zu treffen, um die in den nachfolgend besprochenen Methodenspezifikationen möglichen Konstrukte *Class*`@pre` und `new(.)` zur Charakterisierung neuer Objekte zu evaluieren. Da Methodenaufrufe geschachtelt sein können, wird der Aufrufkeller bei der Speicherung der Instanzmengen nachgebildet, um so jeder Nachbedingung den Zugriff auf den jeweiligen Zustand zur Vorbedingung zu ermöglichen. Wesentlich ist hier die effiziente Verwaltung und Optimierung in Abhängigkeit tatsächlich benötigter Extensionen.

5.3.5 Navigation und Flattening

Der Flatten-Operator `flatten` wird für die verschiedenen Varianten der Collections in der in 3.3.6, Band 1 beschriebenen Form realisiert und bei dem Flachdrücken von Navigationsergebnissen angewandt.

Die Navigation der OCL ist mengenwertig. Da Java eine Navigation ausgehend von mengen- und listenwertigen Strukturen nicht kennt, sind diese entsprechend zu transformieren. Tabelle 5.21 zeigt eine effiziente Umsetzung der Navigation.

Navigation: Mengen- und listenwertige Navigation	
Erklärung	Navigation ausgehend von Containerstrukturen ist in Java durch Iteratoren zu realisieren.
einfache Navigation	$a.role \Downarrow a.role$ • Ausdruck *a* beschreibt ein einzelnes Objekt. Die Navigation entlang der gewünschten Assoziation ist durch *role* möglich. • Eine weitere Umsetzung durch Regeln für Assoziationen ist zu beachten. Zum Beispiel kann eine Kapselung von `a` durch Zugriffsmethoden vorgenommen werden. • Die Navigation kann ein mengenwertiges Ergebnis haben oder qualifiziert sein. Dann ist gegebenenfalls eine Anpassung der Container-Form notwendig.

(Fortsetzung auf nächster Seite)

[13] Siehe Abschnitt 3.4.1, Band 1 über die Objekterzeugung in Queries.

(Fortsetzung von Tabelle 5.21.: Navigation: Mengen- und listenwertige Navigation)

Navigation aus einer Menge

sa`.`*role* ⇓

Java/P

```
Set<Class> res = new HashSet<Class>;
for(Iterator<Class2> it = sa'.iterator();
                          it.hasNext(); ) {
  Class2 a = it.next();
  if(a.role != null)
    res.add(a.role);
}
// Weiterverwendung von res
```

- Ausdruck *sa* beschreibt eine Menge von Objekten. Die Navigation entlang der gewünschten Assoziation ist durch *role* möglich. Die Kardinalität ist „`1`“ oder „`0..1`“.
- Siehe auch Aussagen zur einfachen Navigation.
- *Class* ist der Typ der durch die Assoziation erreichten Klasse.
- *Class2* ist der Typ der Ausgangsklasse.

Mengenwertige Navigation aus einer Menge

sa`.`*role* ⇓

Java/P

```
Set<Class> res = new HashSet<Class>;
for(Iterator<Class2> it = sa'.iterator();
                          it.hasNext(); ) {
  Class2 a = it.next();
  res.addAll(a.role);
}
// Weiterverwendung von res
```

- Im Unterschied zum obigen Fall ist die Kardinalität nun „`*`“, also `a.`*role* mengenwertig.
- Ansonsten gelten Aussagen der vorherigen Transformation.

Tabelle 5.21.: *Navigation:* Mengen- und listenwertige Navigation

5.3.6 Quantoren und Spezialoperatoren

Unendliche Quantoren, also Quantoren über den Grunddatentypen wie `int` und über Containerstrukturen wie `List<Auction>` werden nicht übersetzt. Alle objektwertigen Quantoren jedoch sind endlich und damit in Java übersetzbar. Eine entsprechende Realisierung in Form eines Iterators ist

bereits im Eingangsbeispiel dieses Abschnitts gezeigt worden. Operatoren wie **any** oder **iterate** können in ähnlicher Form umgesetzt werden. Auch die Umsetzung von Referenzen auf Objektdiagramme in der OCL sind einfach. Ein Objektdiagramm wirkt als Prädikat, kann daher selbst in eine boolesche Methode transformiert und der Aufruf in den generierten Code eingesetzt werden. Das von der OCL/P zur Verfügung gestellte **typeif**-Konstrukt zur typsicheren Konversion von Objekten wird durch eine Abfrage mit **instanceof** und eine Typkonversion realisiert.

5.3.7 Methodenspezifikation

Die folgende Methodenspezifikation beschreibt für einen eingeschränkten Bereich, wie nach Eingang eines neuen Gebots eine neue Zeit für das Auktionsende festgelegt wird. Dabei sind einige zeitliche Abhängigkeiten zu sichern:

```
context Time ConstantTimingPolicy.newCurrentClosingTime(
          Auction a, Bid b)
let   long old   = a.closingTime.timeSec;
      long now   = b.time.timeSec
pre:  status==RUNNING && isInExtension && now <= old
post: result.timeSec ==
          min(now + extensionTimeSecs,a.finishTime.timeSec)
```

Die Vorbedingung wirkt wie ein zu Beginn der Methode formuliertes `ocl`-Statement und die mit dem in der OCL zur Verfügung stehenden **let**-Konstrukt definierten Variablen werden als lokale Variable verstanden. Die Übersetzung dieser Spezifikation in erweitertes Java erfolgt in Abbildung 5.22 unter der Annahme, dass der eigentliche Methodenrumpf gegeben ist.

Diese Methode aus Abbildung 5.22 kann, wie in in Anhang B, Band 1 beschrieben, in normales Java übersetzt werden. Gemäß der in Abschnitt 3.4.3, Band 1 beschriebenen Bedeutung von Methodenspezifikationen muss die Nachbedingung nur erfüllt sein, wenn die Vorbedingung gilt. Deshalb wird die Vorbedingung evaluiert und ihr Ergebnis in der Variable `precond` gespeichert, um in der Nachbedingung überprüft zu werden. Mit dieser Form der Transformation ist es möglich, mehrere Methodenspezifikationen, die unabhängig voneinander definiert oder geerbt wurden, gleichzeitig zu testen. Eine wie in Abschnitt 3.4.3, Band 1 beschriebene Integration ist daher für diesen Zweck nicht notwendig.

Je nach Generatoreinstellung werden `ocl`- und `let`-Anweisungen nicht übersetzt, im Fehlerfall eine Exception erzeugt, eine Warnung im Protokoll vermerkt und nach Bedarf ein Auszug des aktuellen Objekts ausgegeben. Das `let`-Konstrukt zur Definition von Hilfsvariablen in Java, die nicht in den Produktionscode gehören, wird wie erwartet durch lokale Variablendefinitionen realisiert. Es entsteht der in Abbildung 5.23 dargestellte Code,

Java/P

```
class ConstantTimingPolicy {
  public Time newCurrentClosingTime(Auction a, Bid b) {
    let long old   = a.closingTime.timeSec;
    let long now   = b.time.timeSec;
    let boolean precond = (status==RUNNING && isInExtension
                                           && now <= old);

    // Rumpf der Java Funktion (ohne return-Anweisung)
    Time result = // Ausdruck der Return-Anweisung
    ocl !precond || (result.timeSec
         == OCL.min(now + extensionTimeSecs,
                    a.finishTime.timeSec));
    return result;
  }
}
```

Abbildung 5.22. OCL transformiert in erweitertes Java

Java

```
class ConstantTimingPolicy {
  public Time newCurrentClosingTime(Auction a, Bid b) {
    long old   = a.closingTime.timeSec;
    long now   = b.time.timeSec;
    boolean precond =
           (status==RUNNING && isInExtension && now <= old);
    // Rumpf der Java Funktion (ohne return-Anweisung)
    Time result = // Ausdruck der Return-Anweisung
    if(precond && !(result.timeSec
         == OCL.min(now + extensionTimeSecs,
                    a.finishTime.timeSec))
           alert(...);
    return result;
  }
}
```

Abbildung 5.23. OCL transformiert in Java

bei dem allerdings vereinfachend das Abfangen von Exceptions weggelassen wurde.

Dieser relativ einfachen Umsetzung steht eine substantielle Komplexität der Umsetzung von Methodenspezifikationen gegenüber, wenn die Nachbedingung auf die Ursprungswerte der Variablen zu Beginn des Methodenaufrufs zugreift. Für die Übersetzung solcher Zugriffe ist bei der Generierung eine Infrastruktur notwendig, die feststellt, welche früheren Werte gegebenenfalls benötigt werden und diese geeignet zwischenspeichert. Dabei können wie nachfolgend beschrieben deutliche Aufwände entstehen. Dazu wird das folgende aus Abschnitt 3.4.3, Band 1 adaptierte Beispiel betrachtet, das den

Effekt eines Unternehmenswechsels einer Person unter der Annahme, dass das Unternehmen bereits erfasst ist, beschreibt:

```
context Person.changeCompany(String name)                          OCL
pre:  exists Company co: co.name == name
post: company.name           == name &&
      company.employees      == company.employees@pre     +1 &&
      company@pre.employees  == company@pre.employees@pre -1
```

In der Nachbedingung wird unter anderem mit `company@pre` auf das vorherige Unternehmen der wechselnden Person und mit `company.employees@pre` auf die Anzahl der vorherigen Mitarbeiter des neuen Unternehmens zugegriffen. Der Ausdruck `company@pre` ist für den Codegenerator leicht zu identifizieren und sein Inhalt zu speichern. Dies wirkt, als ob dabei eine interne Hilfsvariable angelegt wird. Gleiches gilt für den Ausdruck `company@pre.employees@pre`:

```
context Person.changeCompany(String name)                          OCL
let   Company companyPre = company;
      int   employeesPre = company.employees
pre:  exists Company co: co.name == name
post: company.name           == name &&
      company.employees      == company.employees@pre +1 &&
      companyPre.employees   == employeesPre -1
```

Schwierigkeiten bereitet jedoch `company.employees@pre`, weil hier auf den früheren Zustand des neuen Unternehmens zugegriffen wird. Für einen Codegenerator ist es faktisch unmöglich Code zu generieren, der zu Beginn einer Methodenausführung errät, welches das neue `company`-Objekt sein wird und dann dessen alten Zustand speichert. Dieses Problem wird noch deutlicher am Ausdruck `ad.auction@pre[id]` sichtbar, der in der Klasse `AllData` zur Selektion einer Auktion mit dem Identifikator `id` verwendet werden kann. Da `id` erst nach Methodenausführung evaluiert wird, ist unklar, welche Auktion des Ursprungszustands von `ad.auction` selektiert werden wird. Deshalb gibt es drei Strategien damit umzugehen:

1. Es werden die alten Zustände aller *company*-Objekte gespeichert. Dies erfordert allerdings bereits für kleine Testdatenstrukturen einen deutlichen Effizienzverlust und ist im Testeinsatz des Produktionssystems mit großen Datenbeständen nicht umzusetzen.
2. Für Tests wird eine Infrastruktur generiert, die eine interne Protokollierung aller Attributänderungen einschließlich der ursprünglichen Werte durchführt. Dies erfordert ebenfalls eine geeignete Infrastruktur, hat aber unter Umständen den Vorteil, dass die Änderungshistorie eines gescheiterten Tests analysiert werden kann.
3. Der Codegenerator warnt vor der Ineffizienz der Spezifikation oder weist sie sogar zurück mit dem Hinweis, eine effizientere Formulierung

zu finden. Der Entwickler hat meist eine gute Vorstellung, welche Objektzustände tatsächlich zu speichern sind und legt diese explizit in `let`-Anweisungen ab.

Wird eine Effizienzsteigerung durch manuelle Umsetzung gewünscht, so kann im Beispiel etwa aus der Vorbedingung „erraten" werden, welches `company`-Objekt relevant ist und damit die Vorbedingung vereinfacht werden kann:

```
context Person.changeCompany(String name)                    OCL
let   Company newCo = AllData.ad.company[name]
pre:  newCo != null
post: company.name           == name &&
      company.employees      == newCo.employees@pre +1 &&
      company@pre.employees  == company@pre.employees@pre -1
```

Um sicherzugehen, dass das neue Unternehmen tatsächlich mit dem in der **let**-Anweisung definierten Unternehmen übereinstimmt, kann zusätzlich `company==newCo` in die Nachbedingung aufgenommen werden. Diese Form benötigt nun nur minimalen zusätzlichen Speicherplatz, wird allerdings bereits so detailliert und implementierungsnah, dass dann eine direkte Implementierung unter Umständen vorzuziehen ist.

5.3.8 Vererbung von Methodenspezifikationen

Ob eine Methodenspezifikation für die Implementierungen der Unterklassen gilt, wird gemäß Abschnitt 3.4.3, Band 1 durch den Stereotyp «not-inherited» bestimmt. Im Allgemeinen sind deshalb die geerbten Vor- oder Nachbedingungen zusätzlich zu testen. Für eine redundanzfreie Implementierung werden die zu prüfenden Bedingungen aber nicht direkt als Zusicherungen formuliert, sondern in eigenständige Methoden ausgelagert, die in Unterklassen in geeigneter Form zur Verfügung stehen. Dieses Verfahren wird „Percolation" genannt [Bin99] und führt zu der in Abbildung 5.24 dargestellten Implementierung.

Ein Vorteil dieses Verfahrens ist, dass ein Testtreiber zusätzlich prüfen kann, ob die Vorbedingung der Methodenspezifikation auch erfüllt ist, indem er diese vorab auf den gegebenen Testdaten prüft. Dadurch wird verhindert, dass ein Test als Erfolg gewertet wird, weil die Testdaten fälschlicherweise so aufgebaut wurden, dass sie die Vorbedingung nicht erfüllen.

Das in Abbildung 5.24 dargestellte Verfahren kann analog für Invarianten verwendet werden, so dass auch diese in den Unterklassen für Tests zur Verfügung stehen.

5.4 Ausführung von Statecharts

Wie David Harel, dem die Erfindung der Statecharts zugeschrieben wird, gerne bemerkt, „Buildings are there to be, but Software is there to do".

```
class ConstantTimingPolicy {                                    Java
  // Vorbedingung ohne/mit vorbelegten Parametern
  boolean preNewCurrentClosingTime(Auction a, Bid b) {
    long old   = a.closingTime.timeSec;
    long now   = b.time.timeSec;
    return preNewCurrentClosingTime(a,b,old,now);
  }
  boolean preNewCurrentClosingTime(Auction a, Bid b,
                                   long old, long now){
    return status==RUNNING && isInExtension && now <= old;
  }

  // Nachbedingung
  boolean postNewCurrentClosingTime(Time result,
                    Auction a, Bid b, long old, long now) {
    return result.timeSec
         == OCL.min(now +
                    extensionTimeSecs,a.finishTime.timeSec);
  }

  // Eigentliche Funktion
  public Time newCurrentClosingTime(Auction a, Bid b) {
    long old   = a.closingTime.timeSec;
    long now   = b.time.timeSec;
    boolean precond = preNewCurrentClosingTime(a,b,old,now);
    // Rumpf der Java Funktion (ohne return-Anweisung)
    Time result = // Ausdruck der Return-Anweisung
    if(precond &&
             !postNewCurrentClosingTime(result,a,b,old,now))
            alert(...);
    return result;
  }
}
```

Abbildung 5.24. OCL-Bedingungen transformiert in Java-Prädikate

Da Statecharts in natürlicher Weise zur vollständigen Beschreibung von Verhalten genutzt werden können, ist der Aspekt der Codegenerierung, also der Ausführung von Statecharts, von besonderem Interesse. Deshalb wird in diesem Abschnitt auf Realisierungsstrategien für Statecharts eingegangen, ohne jedoch vollständige Übersetzungsalgorithmen zu beschreiben. Das Ziel dieses Abschnitts ist es auch, dem Leser durch die Beschreibung der Beziehung zwischen syntaktischen Konzepten der Statecharts und Java-Codeelementen zum einen die Möglichkeit zu geben, manuell ein Statechart in Code zu übersetzen. Genauso wichtig ist die dadurch entstehende Hilfestellung zum Verständnis von Statecharts und ihrer Verwendung bei der Codegenerierung. Die Übersetzung der Statecharts in Java dient sowohl als

Semantikdefinition, als auch zur Verbesserung des intuitiven Zugangs den Statecharts. Die nachfolgend diskutierten Realisierungsstrategien zeigen alternative Umsetzungsmöglichkeiten von Statecharts in Java, die jedoch semantisch äquivalent sind. Die Auswahl der für ein Projekt geeigneten Umsetzungsstrategie hängt daher von der gewünschten Flexibilität und Effizienzüberlegungen ab.

Die Codegenerierung aus Statecharts ist noch keineswegs so verbreitet wie die auf Basis von Klassendiagrammen, jedoch nimmt die Verwendung von Statecharts für die Implementierung stetig zu. Die eingebetteten Systeme sind hier der Wegbereiter: „... automatically generated code can and is being used today in a variety of hard real-time and embedded systems." [Dou99, S. 156].

Die nachfolgend diskutierten Varianten zur Umsetzung sind keineswegs vollständig, sondern stellen nur einen Ausschnitt der Umsetzungsmöglichkeiten dar. Die bereits eingeführten Stereotypen ≪statedefining≫, ≪completion:ignore≫ und weitere können zur Auswahl zwischen diesen Varianten verwendet werden. Nicht alles lässt sich jedoch bereits mit Stereotypen beschreiben, weshalb die Verwendung von Templates und Skripten für die Codegenerierung von Statecharts sehr hilfreich ist.

5.4.1 Methoden-Statecharts

Die Zustände eines Statecharts können auf verschiedene Arten interpretiert werden. In einem Methoden-Statechart, wie dem in Abbildung 5.25, repräsentieren die Diagrammzustände Zwischenzustände innerhalb des Ablaufs einer Methode. Die Diagrammzustände werden deshalb durch den Programmzähler unterschieden. Diese Diagrammzustände können noch einmal in zwei Klassen unterschieden werden. Zum einen gibt es Diagrammzustände, die Stellen zwischen Anweisungen entsprechen und deren Fortführung durch spontane ε-Transitionen erfolgt. Zum anderen gibt es Zustände, in denen auf das Ergebnis eines Methodenaufrufs gewartet wird, der in einer ankommenden Transition gestartet wurde. Während die Quellzustände spontaner Transitionen vor allem dazu genutzt werden, den Kontrollfluss, wie zum Beispiel Verzweigungen, zu modellieren, stellen Quellzustände von Transitionen, die mit dem Stimulus `return` markiert sind, echte Unterbrechungen der Ausführungen einer Methode dar.

Die Umsetzung eines Methoden-Statecharts erfolgt kanonisch durch die Generierung der beschriebenen Methode und ihres Rumpfs.

Von besonderem Interesse ist die Behandlung von ε-Schleifen innerhalb eines Statecharts. Die Existenz einer ε-Schleife bedeutet, dass der Kontrollfluss innerhalb eines Methoden-Statecharts eine Schleife besitzt. Das Statechart selbst ist daher nicht mehr vollständig in der Lage, das Verhalten der Methode zu beschreiben. Die Initialisierung, der Rumpf und die Abbruchbedingung der Schleife sind jeweils durch Aktionen zu beschreiben. Dabei ist nicht gesichert, dass die Schleife definitiv abbricht. Dennoch kann davon

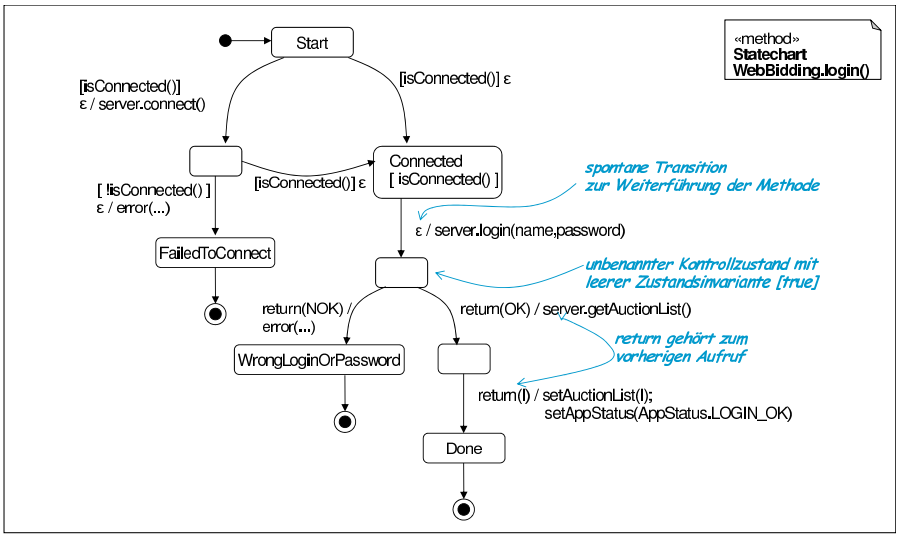


Abbildung 5.25. Repräsentation von Zwischenzustände einer Methode

ausgegangen werden, dass modelliertes Verhalten grundsätzlich terminiert. Diese Annahme ist aus pragmatischen Gründen sinnvoll, weil in der vorgeschlagenen Entwicklungsmethode Schleifen grundsätzlich mit Tests so überprüft werden, dass eine nichtterminierende Schleife entdeckt werden würde.

5.4.2 Umsetzung der Zustände

Die Statecharts, die nicht zur Darstellung eines Methodenablaufs dienen, beinhalten keine Kontrollzustände und daher auch keine spontanen Transitionen. Die Zustände eines solchen Statechart entsprechen daher Datenzuständen des Objekts. Das heißt, der Diagrammzustand des Statecharts muss aus dem Datenzustand des Objekts rekonstruierbar sein. Dies wurde bereits bei der Transformation in vereinfachte Statecharts in Abschnitt 5.6.3, Band 1 diskutiert und ein Verfahren zur Darstellung von Diagrammzuständen gezeigt. Hier werden weitere Verfahren besprochen.

Die Statecharts, die nicht zur Darstellung eines Methodenablaufs dienen, beinhalten keine Kontrollzustände und daher auch keine spontanen Transitionen. Die Zustände eines solchen Statechart entsprechen daher Datenzuständen des Objekts. Das heißt, der Diagrammzustand des Statecharts muss aus dem Datenzustand des Objekts rekonstruierbar sein. Dies wurde bereits bei der Transformation in vereinfachte Statecharts in Abschnitt 5.6.3, Band 1 diskutiert und ein Verfahren zur Darstellung von Diagrammzuständen gezeigt. Hier werden weitere Verfahren besprochen.

Leichtgewicht: Nutzung der Zustandsinvarianten

Als besonders einfach anzusehen ist die Strategie, die paarweise disjunkten Zustandsinvarianten des vereinfachten Statecharts zu nehmen, um aus einem Objekt den Diagrammzustand jeweils zu *berechnen*. Abbildung 5.26 zeigt schematisch eine solche Umsetzung. Bei dieser Transformation wird der linken Transition der Vorrang vor der mittleren Transition gegeben, da ihre Vorbedingung zuerst evaluiert wird. Überlappen die beiden Vorbedingungen, also *vorb1*`&&`*vorb2* ist nicht äquivalent zu `false`, dann wurde damit eine Entwurfsentscheidung getroffen, die eine spezielle Implementierung aus der Menge der möglichen Implementierungen auswählt.

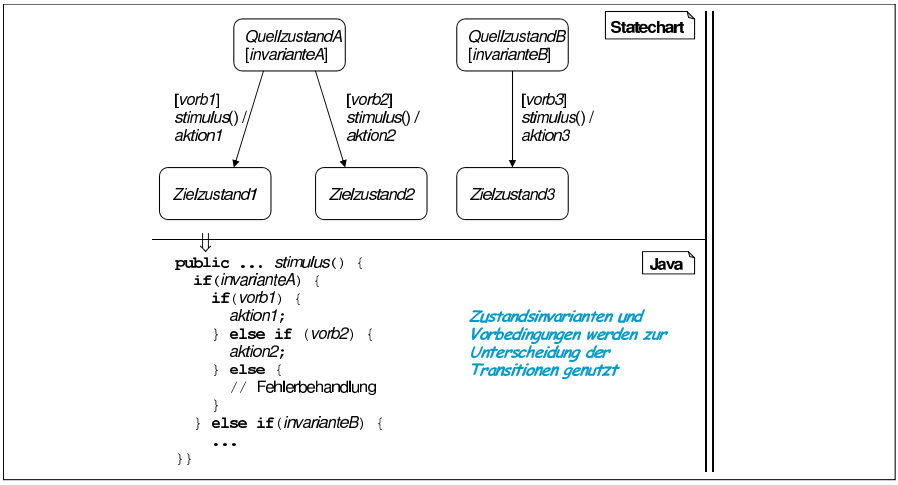


Abbildung 5.26. Übersetzung unter Ausnutzung von Zustandsinvarianten

Der Vorteil dieser Umsetzung ist, dass kein zusätzliches Attribut notwendig ist, um den Diagrammzustand im Objekt zu speichern. Umgekehrt müssen im ungünstigsten Fall jedoch die Zustandsinvarianten aller Zustände evaluiert werden, wodurch ein deutlicher Effizienzverlust auftreten kann. Deshalb ist diese Umsetzung nur für effizient evaluierbare Zustandsinvarianten sinnvoll. Meist bestehen eine Reihe von Optimierungsmöglichkeiten, weil Zustandsinvarianten „verwandter“ Zustände oft gemeinsame Teilbedingungen besitzen, deren Evaluierung nur einmal notwendig ist. Ähnliches gilt für die Evaluierung der Vorbedingungen von Transitionen mit demselben Quellzustand. Gilt zum Beispiel *vorb1* <=> ! *vorb2*, so lässt sich die innere if-Abfrage in Abbildung 5.26 deutlich vereinfachen.

Optimierungen bei der Codegenerierung, wie die oben besprochenen, sind im Allgemeinen nicht automatisiert erkennbar. Jedoch besteht die berechtigte Hoffnung, dass die Werkzeuge zur Generierung von Code aus Modellen in der nächsten Zukunft ähnliche Optimierungsalgorithmen einbauen

werden, wie dies für Compiler textueller Programmiersprachen heute bereits der Fall ist. Bis dahin ist damit zu rechnen, dass die gesteigerte Entwicklereffektivität bei der Verwendung abstrakter Modelle zu gewissen Effizienznachteilen beim realisierten Code führt. Unter Umständen ist dann, wie etwa bei den Herstellern eingebetteter Systeme, durchaus zu beobachten, dass nach Fertigstellung der Modelle zu Simulations- und Validierungszwecken ein zusätzlicher manueller Schritt zur Optimierung des Ergebnisses sinnvoll ist. Dafür lassen sich Refactoring-Techniken auf Ebene der Zielsprache ebenso verwenden, wie die Verwendung zusätzlicher Steuerungsmechanismen bei der Generierung des Codes. Beispielsweise können durch die Vergabe von Prioritäten und die Zusicherung der Disjunktheit von Vorbedingungen dem Codegenerator Optimierungen vorgeschlagen werden.

Zustände als Prädikate

Eine Variation der gezeigten Umsetzung ist in 5.27 zu sehen, in der die Auswertung einzelner Zustandsinvarianten und Aktionen in eigene Methoden ausgelagert wurde. Die Hilfsmethoden zur Umsetzung von Aktionen lassen sich dann gegebenenfalls wiederverwenden, wenn verschiedene Transitionen dieselbe Aktion besitzen. Die Auslagerung der Evaluierung von Zustandsinvarianten in eigene Prädikate, die den Zustandsnamen als Prädikatnamen tragen, hat darüber hinaus den Vorteil, dass auf diese Weise der Diagrammzustand des Statecharts an beliebigen Stellen im Code festgestellt werden kann. Dies ist besonders für Methoden hilfreich, deren Verhalten nicht im Statechart spezifiziert ist, aber dennoch von dem im Statechart modellierten Zustandskonzept abhängt.

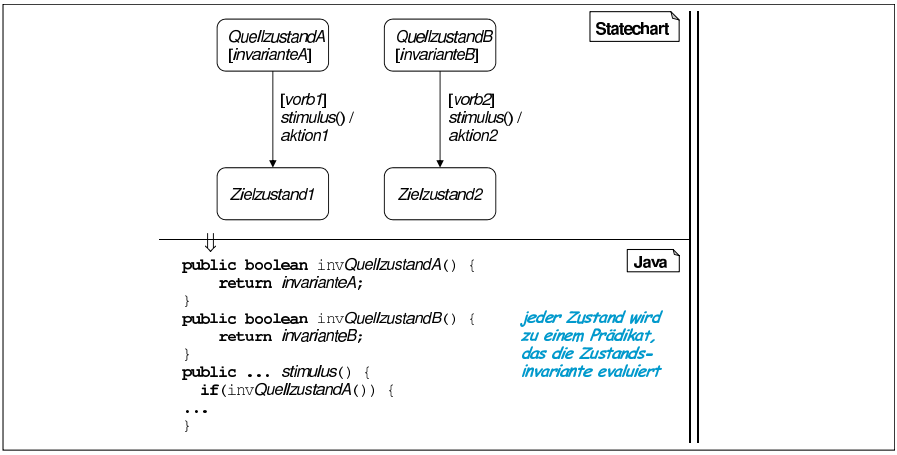


Abbildung 5.27. Prädikate evaluieren Zustandsinvarianten

Aufzählungsattribut als Speicher für den Zustand

Ist die Evaluierung der Zustandsinvarianten zu ineffizient, so eignet sich der heute meistens genutzte Standardweg, den Diagrammzustand in Form eines Attributs, das eine Aufzählung beinhaltet, explizit im Zustandsraum des Objekts abzulegen. Zur Feststellung des Diagrammzustands reicht es nun, das `status`-Attribut zu prüfen. Abbildung 5.28 zeigt die verwendbare Codegenerierung.

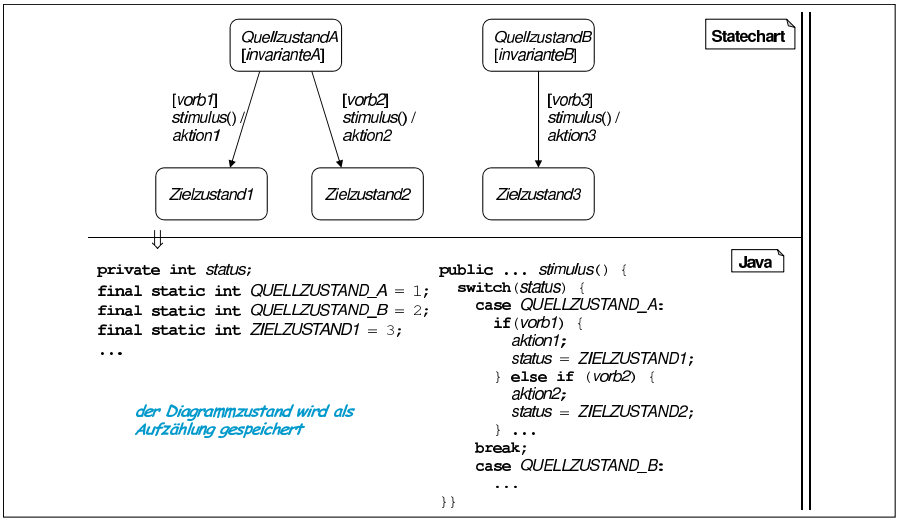


Abbildung 5.28. Diagrammzustand wird in Attribut gespeichert

Vollständiges Statechart

Die Verwendung eines Attributs zur Speicherung des Diagrammzustands führt zu einer besseren Laufzeiteffizienz, verhindert jedoch nicht, dass weiterhin die Vorbedingungen von alternativen Transitionen geprüft werden müssen. Allerdings kann die jeweils letzte Vorbedingung sowie auch die Zustandsinvariante in einer `assert`-Anweisung eingesetzt werden, um statt im konstruktiven Anteil der Implementierung zu Testzwecken verwendet zu werden.[14] Abbildung 5.29 zeigt eine entsprechend modifizierte Umsetzung.

Wurde das Statechart nicht, wie in Abschnitt 5.6.3, Band 1 beschrieben, explizit vervollständigt, so kann in Abhängigkeit der gewählten Semantik diese Vervollständigung auch während der Codegenerierung durchgeführt

[14] Um aussagekräftige Testresultate zu erhalten, empfiehlt es sich, anstatt der von Java zu Verfügung gestellten `assert`-Anweisung das in Abschnitt 6.2.3 diskutierte Test-Framework zu verwenden.

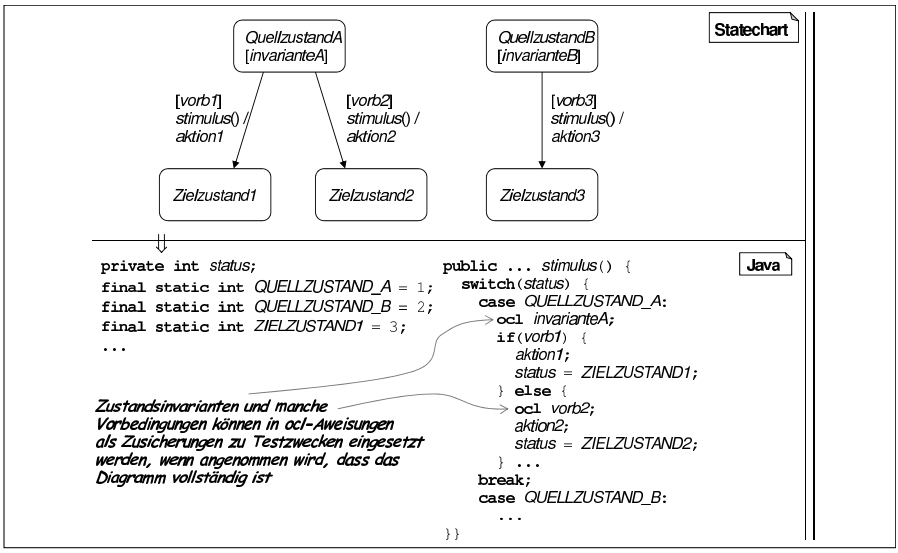


Abbildung 5.29. Verwendung von Zustandsinvarianten als Zusicherungen

werden. Dazu dient zum Beispiel die `default`-Anweisung, die alle nicht durch explizite Transitionen abgedeckten Situationen abfangen kann. Auch kann zum Beispiel ein weiterer Wert in der Aufzählung der Zustände in der Form `ERROR==-1` eingeführt werden, der in solchen Situationen angenommen wird.

Für das Abfangen von Exceptions aufgrund des Stereotyps ≪exception≫ kann eine umfassende `try-catch`-Anweisung verwendet werden. Dabei müssen allerdings je nach gewählter Umsetzungsstrategie mehrere Anweisungen in verschiedenen Methoden oder `case`-Alternativen eingesetzt werden.

Schwergewicht: State-Entwurfsmuster

Eine weitere Möglichkeit zur Umsetzung des Zustandskonzepts ist die Verwendung des State-Entwurfsmusters, zum Beispiel beschrieben in [GHJV94]. Dieses Entwurfsmuster wird als Schwergewicht bezeichnet, da es zu einer Reihe von zusätzlichen Klassen führt. Der Zustand des eigentlich modellierten Objekts wird ausgelagert in eine eigene Zustandsklasse. Diese besitzt mehrere Unterklassen, die je einem Zustand des eigentlich modellierten Objekts entsprechen. Der aktuelle Zustand des Objekts wird durch Referenz auf eines dieser Zustandsobjekte gespeichert. Das Verhalten wird nicht direkt in der modellierten Methode realisiert, sondern an dieses Zustandsobjekt *delegiert*. Dadurch wird das im Statechart auf den Transitionen verteilte Verhalten nicht innerhalb einer Methode zusammengefasst, sondern entsprechend

der Zustände gruppiert. Dies bietet zum Beispiel die Flexibilität, das Verhalten an einem Quellzustand durch Unterklassenbildung zu modifizieren. Der notationelle und operative Aufwand für das State-Entwurfsmuster ist allerdings immens. So muss ein Management der Zustandsobjekte realisiert werden, das entweder dynamisch solche Objekte erzeugt oder in einem Pool von Objekten speichert. Abbildung 5.30 zeigt daher nur einen kleinen Ausschnitt der Umsetzung in einem State-Entwurfsmuster. Die Codeteile *invarianteA'* und *aktion1'* sind dabei entsprechend anzupassen, da die darin enthaltenen Attribute und Methodenaufrufe auf das Ursprungsobjekt `k` statt `self` zugreifen müssen.

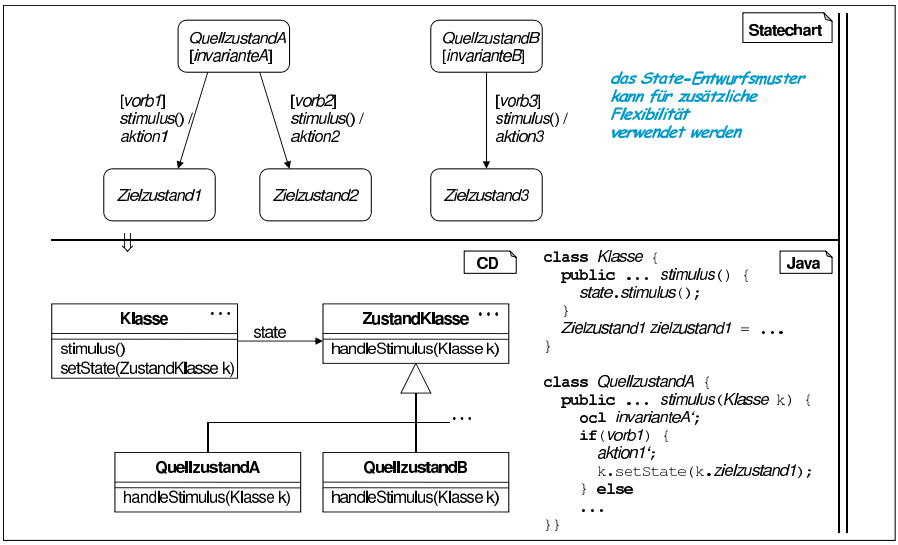


Abbildung 5.30. Schwergewicht: State-Entwurfsmuster

Die Verwendung des State-Entwurfsmusters ist nur zu empfehlen, wenn die dadurch entstehende Flexibilität den Zusatzaufwand bei der Generierung rechtfertigen kann. Da die Umsetzung eines Statecharts in Java-Code normalerweise automatisiert erfolgt und ein manueller Eingriff in den generierten Code im Allgemeinen nicht sinnvoll ist, sollte dementsprechend auch das State-Entwurfsmuster nur selten adäquat sein.

5.4.3 Umsetzung der Transitionen

Stimuli

Wie Abbildung 3.37 zeigt, werden drei Arten von Stimuli unterschieden. Spontane Transitionen und Empfang von return-Ergebnissen treten nur in-

nerhalb von Methoden-Statecharts auf. Die Übertragung asynchroner Nachrichten und der Methodenaufruf werden im Statechart nicht unterschieden. Erst bei der Umsetzung in Code ist die Unterscheidung zwischen einem Methodenaufruf und der Verwendung von Nachrichtenobjekten zur asynchronen Übertragung relevant. Handelt es sich um einen Methodenaufruf, so wird dieser, wie bereits im letzten Abschnitt beschrieben, durch eine entsprechende Methode implementiert.

Wird eine Objektifizierung der Stimuli vorgenommen, so werden Events typischerweise in Form einer Klassenhierarchie mit einer abstrakten Oberklasse `Event` umgesetzt, deren Unterklassen jeweils einzelnen Nachrichtentypen entsprechen. Nachrichten werden durch Aufruf des entsprechenden Konstruktors erzeugt und durch einen geeigneten Übertragungs- und Scheduling-Mechanismus beim Zielobjekt zur Ausführung gebracht. Für den Transport dieser Nachrichten stehen eine Reihe von Frameworks und Middleware-Komponenten, wie zum Beispiel Corba [OH98], zur Verfügung. Weitere Varianten sind die Serialisierung der Nachrichtenobjekte zum Beispiel mit XML [W3C00] oder die Übertragung durch ein selbstdefiniertes, typischerweise effizienteres Protokoll. Aufgrund der hohen Auswahl an zur Verfügung stehenden Lösungen soll dieser technische Aspekt der Kommunikation hier nicht weiter vertieft werden. Für die Umsetzung des Statecharts ist letztendlich nur wesentlich, dass das Nachrichtenobjekt an das zu verarbeitende Objekt übergeben wird, indem eine geeignete Methode aufgerufen wird. Abbildung 5.31 skizziert eine mögliche Realisierung auf Basis einer doppelten `switch`-Anweisung. Durch Definition von Methoden, die jeweils ein spezifisches Nachrichtenobjekt verarbeiten oder durch Auslagerung der Verarbeitung einer Methode auf abhängige Objekte können auch in diesem Fall die bereits früher diskutierten Varianten zur Codegenerierung angewandt werden.

Ein bereits früher diskutierter Vorteil der Verwendung asynchron kommunizierter Nachrichten ist die Vermeidung von rekursiven Objektaufrufen. Die Verarbeitung eines Nachrichtenobjekts findet immer unter exklusivem Zugriff auf den Objektzustand statt. Eine Parallelverarbeitung mehrerer Nachrichten ist ausgeschlossen. Um das sicherzustellen, werden geeignete Synchronisationsmechanismen der Programmiersprache Java eingesetzt. In einem Statechart können auch die Verarbeitung von Nachrichten und Methodenaufrufe gemischt vorkommen. Für das Statechart ist letztendlich unerheblich, ob der Stimulus durch die spezielle Methode `receive` zur Nachrichtenverarbeitung oder durch einen normalen Methodenaufruf bearbeitet wird. Zu beachten ist nur, dass innerhalb einer Aktion des Statecharts keine weiteren Methodenaufrufe an dasselbe Objekt stattfinden, die bezüglich des Statecharts eine zustandsverändernde Wirkung besitzen. Diese, als Rekursionsfreiheit bezeichnete Bedingung, wurde bereits in den Abschnitt 5.1, Band 1 ausführlich diskutiert.

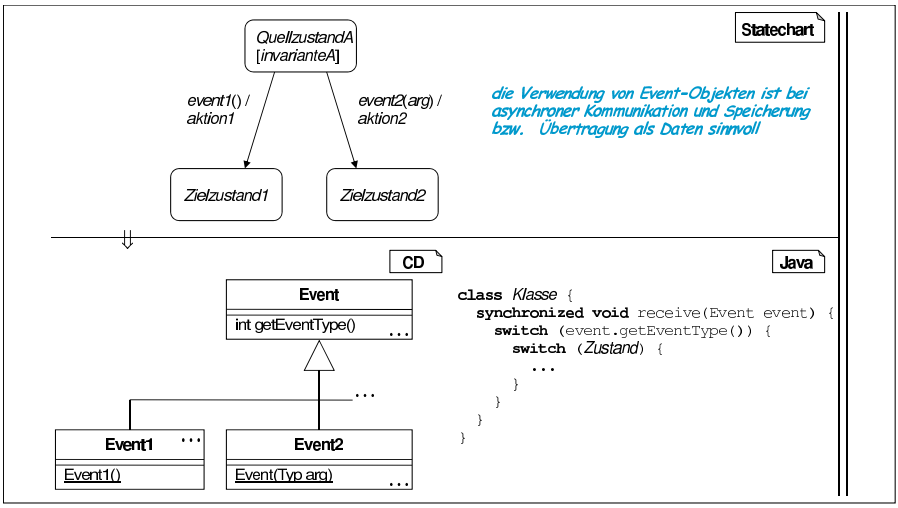


Abbildung 5.31. Verwendung von Event-Objekten als Stimuli

Aktionen

Die Aktionsbeschreibungen eines Statecharts bestehen aus zwei möglichen Komponenten. Die prozedural formulierten Aktionen können nach Anpassung von Attibutzugriffen, etc. gemäß Abschnitt 5.1 in den generierten Code übernommen werden. Wurde ein zusätzliches Attribut zur Speicherung des Diagrammzustands eingeführt, so ist zum Ende der Aktion eine zusätzliche Zuweisung des neuen Zustands notwendig. Die Umsetzung in Abbildung 5.28 illustriert dies.

Wurden bei den Aktionen zusätzlich oder ausschließlich OCL-Nachbedingungen verwendet, so kann daraus im Allgemeinen kein operationeller Code generiert werden. Der Einsatz eines solchen Statecharts zur Programmierung ist daher nicht möglich. Solche Nachbedingungen werden deshalb typischerweise nur zur abstrakten Spezifikation von Verhalten verwendet, das zum einem bei einer Implementierung manuell in ablauffähigen Code umgesetzt oder zum anderen bei Tests eingesetzt wird. Die Nachbedingungen können deshalb nur mithilfe von `ocl`-Anweisungen in den Code umgesetzt werden.

Existente Codegenerierungen für Statecharts

Natürlich ist die oben beschriebene Form nicht die erste Form der Umsetzung von Statecharts in Code. Mehrere Werkzeuge, wie zum Beispiel Statemate oder Rhapsody [HN96] sowie einige der neueren UML-basierten Werkzeuge und Ansätze wie [SZ01, BS01a, BLP01] übersetzen ihre Version der Statecharts in produktiven Code. Dabei werden teilweise auch Konzepte wie

parallele Zustände, ein History-Mechanismus, Pseudozustände oder echte Parallelität innerhalb eines Zustands umgesetzt, die hier nicht eingeführt wurden. Dies liegt unter anderem an der bisher dominierenden Verwendung von Statecharts zur Modellierung verteilter und eingebetteter Systeme, die meist einen höheren Kontrollanteil besitzen als die datenlastigen Geschäftssysteme [Dou98, Dou99].

Statecharts wurden bereits in [Har87] eingeführt und in [vdB94] wurde die bis dahin entstandene Vielzahl von Semantiken verglichen sowie in [vdB01] die UML-Semantik der Statechart in den Vergleich einbezogen. Ein interessantes Merkmal dieser Semantiken ist, dass sie teilweise unterschiedliches Verhalten beschreiben und damit zu unterschiedlichen Implementierungen der Statecharts führen, teilweise aber auch nur verschiedene Mechanismen nutzen, um den Statecharts die essentiell gleiche Semantik zuzuweisen.

Von besonderer Aufmerksamkeit ist die Behandlung der Priorisierung und Unterbrechbarkeit von Transitionen verschiedener Hierarchieebenen und die damit eng verbundene „Run to Completion"-Problematik für Statecharts. Während für eingebettete Systeme die äußeren Transitionen (bezogen auf den Quellzustand) bevorzugt werden, wird zum Beispiel in [HG97] für objektorientierte Statecharts diese Priorisierung umgekehrt und inneren Transitionen der Vorzug gegeben. Der hier verfolgte Ansatz, dies über Stereotypen dem Modellierer selbst entscheiden zu lassen, führt zu mehr Flexibilität. „Run to Completion" spielt bei den UML/P-Statecharts keine Rolle, denn wegen dem zugrunde liegenden, auf Java basierendem Maschinenmodell wird davon ausgegangen, dass die Exceptions, die als Stimuli auftreten, durch das Statechart selbst oder eine aus einer Aktion des Statecharts heraus aufgerufenen Methode verursacht sind und daher eine Weiterführung der Transition zu einem natürlichen Ende nur beschränkt sinnvoll ist. Die Möglichkeit paralleler Verarbeitung von Transitionen in demselben Objekt wird durch die grundsätzlich verwendete Synchronisation auf zustandsbehafteten Objekten ausgeschlossen.

5.5 Übersetzung von Sequenzdiagrammen

Ein Sequenzdiagramm ist in seiner Natur exemplarisch. Es zeigt einen einzelnen möglichen Ablauf eines Systems, das typischerweise abhängig von der aktuellen Objektstruktur, den Inhalten der Attribute und der Systemumgebung alternative Abläufe erlaubt. Die Beschreibung eines exemplarischen Ablaufs ist für die konstruktive Codegenerierung nur schlecht geeignet. Die einzige Möglichkeit, einen exemplarischen Ablauf konstruktiv einzusetzen, ergibt sich, wenn eine damit modellierte Methode nur einen Ablauf besitzt. Sie darf dann keine Verzweigungen des Kontrollflusses oder Iterationen beinhalten. Eine derartig einfache Struktur haben typischerweise Testtreiber und Testbeobachtungen.

Da Sequenzdiagramme jedoch vor allem für die Modellierung von Tests eingesetzt werden, ist eine Codegenerierung für die Prüfung von Testabläufen sinnvoll.

5.5.1 Sequenzdiagramm als Testtreiber

Abbildung 5.32 enthält ein typisches Sequenzdiagramm, von dem für das Objekt `t` eine Methode generiert werden soll, die die mit dem Stereotyp «trigger» markierten Aufrufe durchführt.

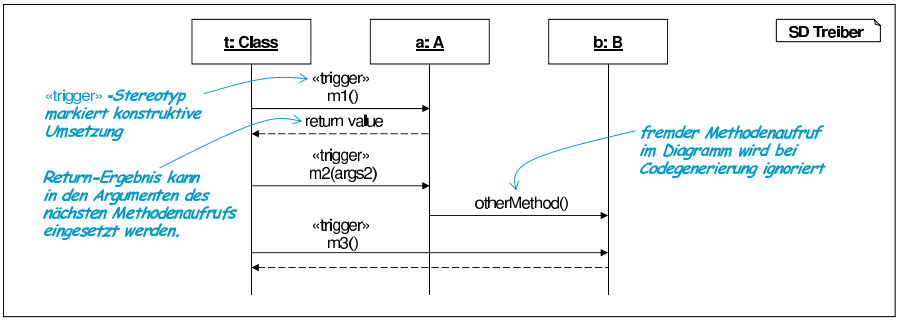


Abbildung 5.32. Sequenzdiagramm mit Treiber

Die zu generierende Methode benötigt einen Namen, der aus dem Diagrammnamen extrahiert werden kann. In Kombination mit einem standardmäßig festgelegten Präfix `runSD` ergibt sich damit der in der Klasse `Class` generierte Methodenname `runSDTreiber` in der Klasse.

Die Signatur dieser Methode ist wie bei der konstruktiven Umsetzung von Objektdiagrammen geprägt durch die freien Variablen des Sequenzdiagramms. Im Beispiel werden daher zumindest die beiden anderen Objekte `a` und `b` als Parameter eingesetzt. Freie Variablen, die zuerst als Argumente eines «trigger»-Aufrufs auftreten, werden ebenfalls als Parameter aufgenommen. Demgegenüber werden freie Variablen, die in Returns das erste Mal verwendet werden, durch diesen Return-Wert belegt. Aufrufe und Returns zwischen anderen Objekten des Sequenzdiagramms sowie Aufrufe an Treiber-Objekte werden bei der Codegenerierung für die angegebene Methode ignoriert. Als Ergebnis entsteht der in Abbildung 5.33 angegebene Code.

Die Typen der im Sequenzdiagramm angegebenen Variablen können, soweit nicht aus dem Sequenzdiagramm ersichtlich, aus dem Kontext, also zum Beispiel der Signatur der Klassen `A` und `B`, bestimmt werden. Als Erweiterung könnte es von Interesse sein, zwischendurch und am Ende zusätzliche Aktionen in Form von Java-Code anzugeben, die zum Beispiel ein explizites Return-Ergebnis der Methode berechnen. Für den Einsatz als Testtreiber ist allerdings die angegebene Form der Codegenerierung ausreichend. Eine Vereinfachung entsteht zum Beispiel, wenn die Variable `a` bereits als Attribut der

```
class Class { ...
  public void runSDTreiber(A a, B b, Type2 args2) {
      Type value = a.m1();
      a.m2(args2);
      b.m3();
  }
}
```

Java

Abbildung 5.33. Aus einem Sequenzdiagramm generierter Treiber

Klasse `Class` deklariert ist. Dann wird auf die Verwendung des Parameters `a` verzichtet.

Das Verfahren zur konstruktiven Umsetzung von Teilen eines Sequenzdiagramms kann auch eingesetzt werden, wenn der Methodenaufruf der generierten Methode im Sequenzdiagramm selbst angegeben ist. Abbildung 5.34 beschreibt ein Dummy-Objekt, das eine einfache Implementierung der Methode `foo()` benötigt. Diese wird nach demselben Verfahren generiert und steht damit für den in Abbildung 5.34 angegebenen Test zur Verfügung. Das Zielobjekt `b` der durch `foo()` aufzurufenden Methode ist als Attribut in der Klasse `Dummy` festgelegt. Die Besetzung der Objektstruktur wird typischerweise durch ein Objektdiagramm vorgenommen.

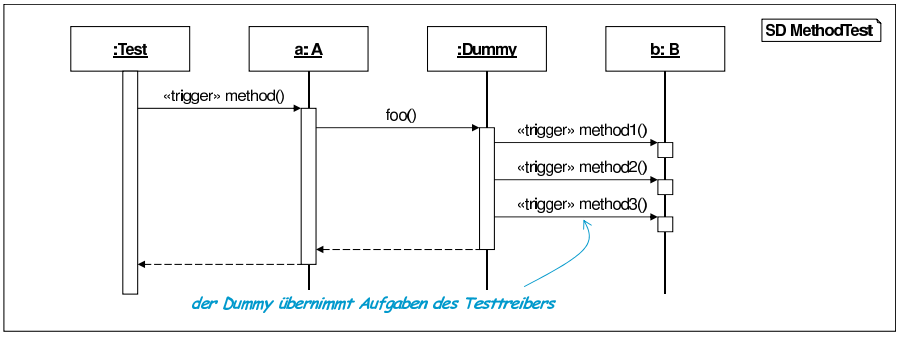


Abbildung 5.34. Sequenzdiagramm mit Treiber und Dummy

5.5.2 Sequenzdiagramm als Prädikat

Ein Sequenzdiagramm beschreibt genau wie ein Objektdiagramm eine exemplarische Eigenschaft des Systems. Im Gegensatz zu einem Objektdiagramm kann diese aber nicht an einem Snapshot eines Systems, sondern muss während eines Systemablaufs geprüft werden. Um die während eines Ablaufs auftretenden Interaktionen prüfen zu können, ist eine entsprechende Instrumentierung des Codes notwendig. Dabei muss zu Beginn und

am Ende jeder beobachteten Methode eine Mitteilung über deren Aufruf beziehungsweise Terminierung erfolgen. Dabei ist auch die anormale Terminierung durch eine Exception zu protokollieren. Dies kann innerhalb der instrumentierten Methode, aber auch durch das Adapter-Entwurfsmuster [GHJV94] erfolgen. Ein solcher Adapter kann zum Beispiel durch Redefinition der Methode in einer Unterklasse gebildet werden. Das Prinzip für die Klasse `A` aus Abbildung 5.34 ist in Abbildung 5.35 dargestellt.

```
class Ainstrumented extends A { ...                    Java
  public Type method() {
    // Protokolliere Methodenaufruf (Objekt, Methode, leere Argumentliste)
    SDlog.call(this, "method", new Object[] {});
    Type result;
    try{
      // Eigentlicher Aufruf
      result = super.method();
    catch (Exception ex) {
        // Protokolliere Exception
        SDlog.exceptionReturn(this, "method", ex);
        throw ex;
    }
    // Protokolliere Return + Ergebnis
    SDlog.normalReturn(this, "method", result);
    return result;
  }
}
```

Abbildung 5.35. Adapter zur Codeinstrumentierung

Es ist allerdings in der Praxis sinnvoll, diese Informationen statt an ein globales Objekt `SDlog` zu übergeben, aus dem Aufrufkeller über die Virtual Machine auszulesen.

Der Algorithmus zur Erkennung, ob ein Sequenzdiagramm in der angegebenen Form abgearbeitet wurde, basiert auf der in Abbildung 6.14, Band 1 angegebenen Form zur Interpretation eines Sequenzdiagramms als regulärer Ausdruck. Der reguläre Ausdruck wird zunächst in einem nichtdeterministischen endlichen Automaten[15] realisiert, der es erlaubt, diesen regulären Ausdruck zu erkennen. Abbildung 5.36 demonstriert dies an einem Beispiel.

Im Sequenzdiagramm können prototypische Objekte auftreten, für die zunächst keine Zuordnung zu echten Objekten existiert. Diese Zuordnung findet bei der jeweils ersten Interaktion mit einem passenden Objekt statt. Deshalb werden die Zustände des nichtdeterministischen Automaten um die Konfiguration dieser Objekte sowie weiterer freier Variablen erweitert.

[15] Der Automat dient zur Erkennung von Eingabesequenzen und hat mit den in Kapitel 5, Band 1 beschriebenen Statecharts formal nichts zu tun.

Deshalb kann ein Automat mehrfach denselben Zustand mit verschiedenen Objektkonfigurationen einnehmen.

Nach jeder Interaktion werden eventuell im Sequenzdiagramm nachfolgende OCL-Bedingungen geprüft und die Konfigurationen, welche die Bedingung nicht erfüllen, entfernt. Dies kann im Automaten durch eine zusätzliche Transition dargestellt werden, die keine Interaktion verarbeitet, aber eine Vorbedingung besitzt. Beginnend mit einer Konfiguration im Startzustand wird unter Umständen eine leere Menge von gültigen Konfigurationen erreicht.

Ein Sequenzdiagramm gilt als *erfüllt*, wenn nach der Testdurchführung der Endzustand in den erreichten Konfigurationen enthalten ist. Als Nebenerzeugnis der Prüfung entsteht dabei auch die Belegung der freien Variablen und der prototypischen Objekte.

Die verschiedenen durch den Stereotyp «match» wählbaren Semantiken für Sequenzdiagramme werden durch die in Abbildung 6.14, Band 1 beschriebene Beschränkung von Interaktionen, die ignoriert werden dürfen, umgesetzt.

Der beschriebene Algorithmus kann anhand des in Abbildung 5.36 beschriebenen Sequenzdiagramms illustriert werden. Dabei wird von konkreten Werten der Parameter abstrahiert. Diese können genauso behandelt werden, wie die OCL-Bedingungen, also zusätzlich geprüft werden. Abbildung 5.36 beinhaltet auch den Automaten, der zur Erkennung des Sequenzdiagramms verwendet wird.

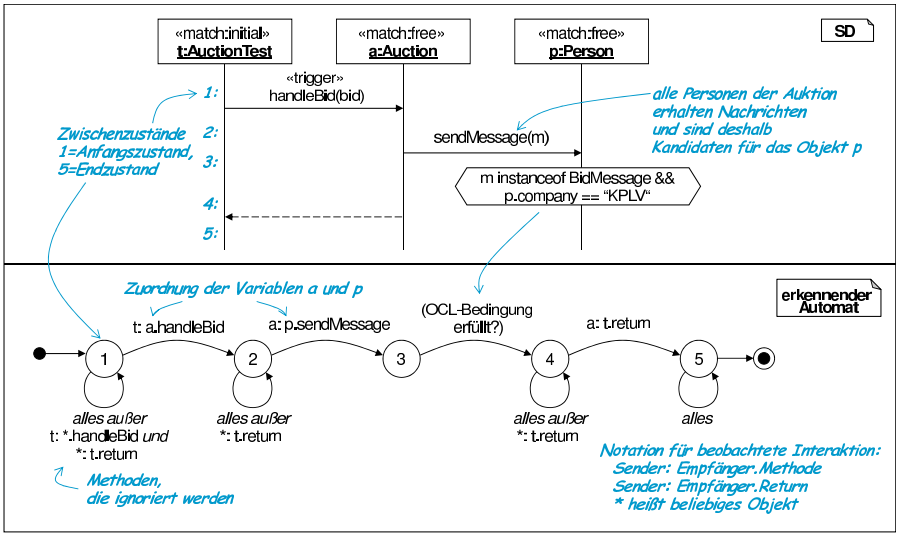


Abbildung 5.36. Erkennender Automat aus einem Sequenzdiagramm

Die übliche Konstruktion, um den Automaten deterministisch zu machen und zu minimieren [HU90], kann aufgrund der Konfigurationen, die Auswirkungen auf die Schaltbereitschaft weiterer Transitionen und die Evaluierung der OCL-Bedingungen haben, im Allgemeinen nicht durchgeführt werden. Dies wäre zum Beispiel möglich, wenn die im Sequenzdiagramm dargestellten Objekte im Voraus auf echte Objekte zugeordnet werden könnten. Andererseits hat zum Beispiel ein Automat, der aus einem mit dem Stereotyp «match:complete» markierten Sequenzdiagramm entsteht, keine Schleifen und ist damit bereits deterministisch.

Ein Sequenzdiagramm, das bereits teilweise zur konstruktiven Generierung von Treibern benutzt wurde, kann zusätzlich zu einer Prüfung verwendet werden. Dass dabei auch der Testtreiber instrumentiert wird, erzeugt nur wenig Zusatzaufwand, ist aber notwendig, um die Reihenfolge der auftretenden Nachrichten und die OCL-Bedingungen prüfen zu können.

5.6 Zusammenfassung zur Codegenerierung

Kapitel 4 und dieses Kapitel diskutieren zunächst grundsätzliche Fragestellungen zur Codegenerierung, beginnend mit der Ausdrucksmächtigkeit der UML/P bis hin zu einer sinnvollen Architektur eines Codegenerators, der die notwendige Flexibilität und Parametrisierbarkeit besitzt, um verschiedenen Zielplattformen und unterschiedlichen Einsatzgebieten der UML/P gerecht zu werden. Statt einer tatsächlichen Darstellung der *Skripte* und *Templates* wurde eine abstrakte, an den Darstellungen von Regelkalkülen orientierte Form gewählt, um Transformationen darzustellen. Dabei wurde der pragmatische Ansatz unvollständiger und verkürzender Transformationsregeln in Kombination mit erklärendem Text gewählt. Die Umsetzung der einzelnen UML/P-Notationen in Code werden auf Basis dieser Transformationsregeln in diesem Kapitel beschrieben.

Ausblick zur Codegenerierung

Die Ansätze der ersten Generation von CASE-Werkzeugen zeigen, dass bei der Verwendung von Codegenerierung neben den in diesem Kapitel beschriebenen Vorteilen auch einige Nachteile in Kauf zu nehmen sind. So ist die Wartbarkeit eines mit Codegenerierung entwickelten Systems davon abhängig, ob der Codegenerator in der verwendeten oder einer aufwärtskompatiblen Form während des Einsatzzeitraums des Produktionssystems zur Verfügung steht. Ist das nicht der Fall, ist der einzige Ausweg, den generierten Code manuell weiterzubearbeiten. Wird zum Beispiel die technische Plattform migriert, so ist es darüber hinaus notwendig, dass Entwickler mit Fähigkeiten zur Anpassung („Programmierung“) des Generators zur Verfügung stehen.

Wird ein Codegenerator daher im eigenen Projekt entwickelt, dann sollte er sehr einfach sein geschrieben und seine Verwendung wenn bei Wartung und Evolution des Systems notwendig wieder rekonstruierbar sein. Die Komplexität der heutigen Quellsprachen wie UML steht aber im Widerspruch zu diesem Wunsch. Als vernünftige Alternative steht daher ein extern entwickelter, produktreifer Codegenerator zur Diskussion, der eine ausreichende Stabilität aufweist, um auch in der Wartungsphase noch zur Verfügung zu stehen. Um Inkompatibilitäten bei Versionswechseln zu verhindern oder wenigstens zu minimieren, ist hier eine Standardisierung ähnlich der Standardisierung von Compilern und der Semantik der zugrunde liegenden Sprachen geboten. Eine für diese Zwecke ausreichend detaillierte Standardisierung ist in der UML-Standardisierungsdiskussion derzeit nicht zu finden. Dennoch könnte die Interoperabilität heutiger UML-Werkzeuge einen gewissen Standardisierungseffekt für die Codegenerierung bewirken.