Großer Beleg zum Thema

Java-Implementierung der OCL-Basisbibliothek

Bearbeitet von Frank Finger
geboren am 29. Mai 1975 in Obergünzburg

an der

Technischen Universität Dresden
Fakultät Informatik
Lehrstuhl Softwaretechnologie

Betreuerin: Dr. B. Demuth
Verantw. Hochschullehrer: Prof. Dr. H. Hussmann
Eingereicht am 30. Juli 1999

Ich erkläre, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe

Dresden, den 29.9.1999

Inhaltsverzeichnis

1 Einleitung 3

1.1 Motivation . . . 3

1.1.1 Formale Spezifikation . . . 3

1.1.2 OCL . . . 4

1.1.3 Werkzeugunterstützung . . . 5

1.1.4 Basisbibliothek . . . 5

1.2 Aufgabenstellung . . . 6

1.3 Aufbau dieser Arbeit . . . 6

2 Anmerkungen zur OCL 7

2.1 Einschränkungen der Implementierung . . . 7

2.1.1 Entwurfsmodelle . . . 7

2.1.2 Grenzen von Java . . . 8

2.1.3 Erzeugung von internen Repräsentationen . . . 8

2.2 Unklarheiten und Fehler in der Spezifikation . . . 9

2.2.1 Unklarheiten . . . 9

2.2.2 Fehler . . . 9

3 Analyse von Implementierungsvarianten 11

3.1 Basistypen . . . 11

3.1.1 Problem . . . 11

3.1.2 Varianten . . . 11

3.1.3 Lösung . . . 13

3.2 Iterierende Methoden . . . 13

3.2.1 Problem . . . 13

3.2.2 Iteratorvariante . . . 14

3.2.3 OCL-Ausdrücke als Objekte . . . 15

3.2.4 Implementierte Variante . . . 15

3.2.5 `iterate` . . . 16

3.3 Anwendungsklassen . . . 17

3.3.1 Problem . . . 17

3.3.2 Varianten . . . 17

3.4 Notwendigkeit einer gemeinsamen Wurzelklasse . . . 19

3.4.1 Problem . . . 19

3.4.2 Lösung . . . 20

4 Entwurf und Implementierung **21**
4.1 Anpaßbarkeit . . . 21
4.1.1 Erzeugung interner Repräsentationen . . . 21
4.1.2 Namensanpassung . . . 22
4.1.3 Weitere Anpassungsmöglichkeiten . . . 23
4.1.4 Zusammenfassung . . . 23
4.2 Hierarchie . . . 24
4.3 Undefinierte Werte . . . 24
4.4 Kollektionen . . . 25
4.4.1 Iterierende Methoden . . . 25
4.4.2 Spezifikationsabweichungen in `OclCollection` . . . 26
4.4.3 `OclUnsortedCollection` und Schnittstellen . . . 26
4.4.4 Zusammenfassung . . . 27
4.5 Basistypen . . . 27
4.5.1 Klassen und Operationen . . . 27
4.5.2 Schnittstellen . . . 27
4.5.3 Zusammenfassung . . . 29
4.6 Klassen des Anwendungsmodells . . . 29
4.6.1 Zugriff auf Attribute und Methoden . . . 29
4.6.2 Metamodell . . . 31
4.6.3 Zusammenfassung . . . 31
4.7 Beispiele . . . 33

5 Test **36**
5.1 Komponententest . . . 36
5.1.1 JUnit . . . 36
5.2 Integrationstest . . . 37

6 Zusammenfassung und Ausblick **42**
6.1 Zusammenfassung . . . 42
6.2 Kritik . . . 42
6.3 Ausblick . . . 43

Kapitel 1

Einleitung

1.1 Motivation

1.1.1 Formale Spezifikation

Es gibt bekanntlich viele Aspekte der Informatik, in denen die an Universitäten gelehrte Theorie und die in der Wirtschaft übliche Praxis deutliche Diskrepanzen aufweisen. Selbst die Verwendung von systematischen Methoden zur Softwareentwicklung ist nicht überall so selbstverständlich, wie man das, von der Perspektive von Forschung und Lehre aus betrachtet, vermuten könnte. Noch wesentlich deutlicher wird diese Ungleichheit aber im Gebiet der formalen Spezifikation von Softwaresystemen: Während sich auf der einen Seite Scharen von Wissenschaftlern mit dieser Thematik beschäftigen, ist eine praktische Anwendung ihrer Ergebnisse nur in einem sehr ausgesuchten Umfeld zu finden, in dem auf die Zuverlässigkeit von Software besonderer Wert gelegt wird.

Die Ursache für dieses Phänomen ist sicherlich zu einem großen Teil in dem hohen Aufwand zu suchen, der mit formaler Spezifikation verbunden ist. Die umfangreichen Texte, die zur vollständigen Spezifikation selbst relativ einfacher Softwaresysteme schnell notwendig werden, sind außerdem häufig so komplex und unübersichtlich, daß sie auch von Entwicklern, die in dem entsprechenden formalen Verfahren ausgebildet und geübt sind, nur mit Mühe durchdrungen werden können.

Hinzu kommt noch die mangelhafte Integration von formalen Elementen in vielen üblichen Methoden der Softwareentwicklung. Das führt zum einen dazu, daß die Ergebnisse einer formalen Spezifikation häufig isoliert von anderen Dokumenten der Analysephase stehen. Zum anderen ist auch die fehlende Unterstützung in verbreiteten Softwareentwicklungsumgebungen teilweise hierauf zurückzuführen.

Auf der anderen Seite würden einige Elemente formaler Methoden den Prozeß der Softwareentwicklung aber fraglos bereichern. Sicherlich ist jeder Programmierer bei der Umsetzung natürlichsprachlicher Spezifikationen schon über Unklarheiten gestolpert. Noch wesentlich kritischer sind Mehrdeutigkeiten und Widersprüche, die erst während des Einsatzes eines Softwaresystems zu Tage kommen und dann große Schäden verursachen.

1.1.2 OCL

Mit der Object Constraint Language (OCL) steht jetzt eine Sprache zur Verfügung, die sich von „herkömmlichen“ formalen Spezifikationssprachen deutlich unterscheidet. Die Betonung liegt hier weniger auf der mathematischen Fundierung als auf der praktischen Anwendbarkeit.

Mit Hilfe der OCL können präzise Einschränkungen und Bedingungen mit wenig Aufwand zu einem UML-Modell hinzugefügt werden. Die Ursache für diesen pragmatischen Ansatz ist sicher in der Herkunft dieser Sprache zu suchen: Sie ging aus einer Entwicklung der Firma IBM hervor und ist seit der Version 1.1 Bestandteil der Unified Modeling Language (UML) der Object Management Group (OMG) ([3], [1], [4]).

Die OCL hat gegenüber herkömmlichen Spezifikationssprachen den wesentlichen Vorteil, daß es mit ihr möglich ist, ein bestehendes UML-Modell inkrementell zu erweitern. Somit ist es von Dokumenten der objektorientierten Analyse ausgehend nicht mehr nötig, die Semantik des Modells, vor allem die in ihm definierte Typ- bzw. Klassenhierarchie, umständlich in eine äquivalente Darstellung in der Spezifikationssprache umzusetzen. Außerdem können gezielt diejenigen Aspekte präzise beschrieben werden, die aus den UML-Diagrammen nicht klar hervorgehen ([12]).

All dies kann in einer wesentlich kompakteren Schreibweise ausgedrückt werden, als das bei anderen Sprachen möglich ist. Das wird unter anderem durch eine Bibliothek von vordefinierten Typen erreicht.

Ein anderer Punkt, der OCL zur breiteren Anwendung tauglich macht, ist die Einfachheit der Sprache. Während bei anderen Spezifikationssprachen eine umfangreiche Vorbildung in mathematischer Logik zum Verständnis erforderlich ist, ist OCL für einen mit Programmiersprachen vertrauten Leser nahezu intuitiv verständlich, was man von anderen Spezifikationssprachen wie CafeOBJ oder Z kaum behaupten kann.

Diesen Vorteilen stehen auch einige problematische Aspekte gegenüber. Ein Punkt, der für die praktische Anwendung nur in wenigen Fällen ein Hindernis darstellt, ist die beschränkte Ausdrucksfähigkeit der OCL ([4]). Allerdings dürfte für diejenigen, die die Sprache aus der Blickrichtung der mathematischen Logik untersuchen, die nicht gegebene Turingmächtigkeit ein besonderes Manko sein.

Ein weiterer Nachteil der OCL ist ihre Definition, die an vielen Stellen unklar und gelegentlich auch widersprüchlich ist. Besonders störend fällt auf, daß für grundlegende Konzepte der Sprache, wie beispielsweise Objekte und Attribute, unterschiedliche Bezeichnungen verwendet werden. Eine Ursache für diese Probleme ist sicherlich die Definition der OCL über natürliche Sprache. Allerdings finden sich gerade an solchen Stellen zahlreiche Fehler in der Spezifikation, an denen eine Art „Kern-OCL“ zur Definition komplexerer Attribute verwendet wird. Als Beispiel sei hier nur die Nachbedingung erwähnt, die in [1] für die *Collection*-Operation *isUnique* angegeben wird.

Zusammenfassend kann aber trotzdem gesagt werden, daß die OCL durchaus geeignet ist, in realen Projekten Verwendung zu finden, ohne daß das durch die überdurchschnittlichen Sicherheitsansprüche eines solchen Projekts begründet werden müßte. Der wesentliche Vorteil liegt in dem vergleichsweise geringen

Aufwand, der hiermit verbunden ist. Bis mit der OCL formale Spezifikationsmethoden Einzug in den Alltag der Softwareentwicklung halten, sind aber noch einige Hürden zu bewältigen. Denn als reine Dokumentationssprache, die als Interpretationshilfe zu UML-Diagrammen hinzugefügt werden kann, wird sich diese Sprache sicherlich nicht durchsetzen.

1.1.3 Werkzeugunterstützung

Eine wesentliche Motivation, OCL in der Spezifikationsphase einzusetzen, wäre dadurch erreicht, daß der dafür nötige Aufwand durch Softwarewerkzeuge reduziert und in konkrete Ergebnisse überführt wird.

Die einfachste Form der Unterstützung wäre hierbei ein Parser, der OCL-Ausdrücke auf syntaktische Korrektheit untersucht. Hierbei könnten allerdings zahlreiche offensichtliche Fehler nicht erkannt werden, da die Grammatik von OCL nicht besonders restriktiv ist. Hilfreicher wäre hier ein darauf aufbauendes Werkzeug zur Typüberprüfung, das die Konsistenz eines OCL-Ausdrucks mit den vordefinierten Typen der OCL und dem Anwendungsmodell untersucht.

Wiederum auf diesen Werkzeugen aufbauend wäre eine Auswertung von OCL-Ausdrücken auf Instanzen des zugehörigen UML-Modells von Interesse. Dafür müßten die OCL-Ausdrücke entweder zur Laufzeit der Instanz durch einen OCL-Interpreter evaluiert werden, oder sie werden als Teil der Quelltextgenerierung aus dem UML-Modell mit einem OCL-Compiler in die entsprechende Zielsprache transformiert. In beiden Fällen könnte eine Modellinstanz, für objektorientierte Programmiersprachen also konkret eine Objektpopulation, auf Einhaltung der im Modell festgelegten Bedingungen überprüft werden.

Diese Überprüfung wäre für unterschiedliche Zielsetzungen interessant. Der wahrscheinlich naheliegenste Ansatz ist die Überprüfung von Programmen in der Testphase. Ob diese Auswertungen dann während des Einsatzes des Programms noch von Interesse sind, muß von Fall zu Fall entschieden werden. Eine andere Möglichkeit, diese Tests zu nutzen, ist die Überprüfung der in [2] definierten *Well-Formedness Rules* auf Instanzen des UML-Metamodells.

Eine ganz andere Möglichkeit ist die Nutzung der OCL als Konfigurationssprache für komplexe Softwaresysteme. Mit Hilfe eines OCL-Interpreters könnten hierbei zur Laufzeit des Systems sehr flexibel Einstellungen vorgenommen werden.

1.1.4 Basisbibliothek

Sowohl beim kompilierenden als auch beim interpretierenden Ansatz zur Auswertung von OCL-Ausdrücken ist eine Repräsentation des Typsystems der OCL notwendig. Dieses Typsystem wird zum einen durch den Abschnitt *Predefined OCL Types* in [1] definiert, zum anderen auch durch das UML-Modell.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist es, eine Javaimplementierung der vordefinierten Typen von OCL zu erstellen. Da die Ergebnisse für einen Quelltextgenerator, der OCL in Java transformiert, verwendet werden sollen, ist auf eine Anpassung für diesen Einsatz besondere Rücksicht genommen worden. Letztendlich soll jeder OCL-Ausdruck in eine Reihe von Aufrufen dieser Klassenbibliothek umgewandelt werden können. Basis der Implementierung ist Java 2 ([7], [8], [9]).

Für die Zugriffe auf die Modellinstanz ist eine geeignete Kaspelung zu entwerfen, um die Bibliothek in dieser Hinsicht leicht anpassbar zu halten. Eine Standardimplementierung dieser Anpassung ist zu realisieren, die auf den Codegenerator der Softwareentwicklungsumgebung Argo/UML ([5]) zugeschnitten sein sollte.

Außerdem sollte besonderen Wert auf Test und Dokumentation gelegt werden, um sicherzustellen, daß die erstellte Klassenbibliothek ohne unnötigen Einarbeitungsaufwand zuverlässig einsetzbar ist.

1.3 Aufbau dieser Arbeit

Kapitel 2 enthält einige Einschränkungen bezüglich der Umsetzbarkeit der OCL und andere Anmerkungen zur Sprache.

Dann wird die Grobarchitektur der Klassenbibliothek vorgestellt, diese weiter verfeinert, und schließlich wird der Test des Systems besprochen.

Hierbei werden in Kapitel 3 grundsätzliche Entscheidungen zur Architektur erläutert, mit Alternativen verglichen und die getroffene Auswahl begründet.

Die gewählten Implementierungsvarianten werden in Kapitel 4 zu einer Gesamtarchitektur zusammengefügt und detaillierter beschrieben. Außerdem werden Implementierungsentscheidungen zu weiteren Aspekten von OCL erläutert, die geringeren Einfluß auf die Architektur der Klassenbibliothek hatten.

Kapitel 5 beschreibt den Komponenten- und Integrationstest der Bibliothek.

Um Begriffe der OCL-Spezifikation, insbesondere Typen, und ihre Repräsentationen in der Klassenbibliothek zu unterscheiden, werden erstere *kursiv* und letztere in `Maschinenschrift` dargestellt. Quelltexte werden generell in `Blocksatz` geschrieben, wobei auch OCL-Ausdrücke als Quelltext betrachtet werden.

Als Anlage zu dieser Ausarbeitung findet sich eine CD, die neben der Implementierung der Klassenbibliothek auch eine umfangreiche HTML-Dokumentation dazu enthält. Für Anwender der Klassenbibliothek ist dies die geeignetste Informationsquelle.

Kapitel 2

Anmerkungen zur OCL

Die Klassenbibliothek setzt die OCL-Spezifikation ([1]) um, die Teil der UML 1.3 ([2]) ist. Einige Aspekte ließen sich dabei nicht realisieren. Diese sollen in diesem Kapitel dargestellt werden. Außerdem sollen in Abschnitt 2.2 Probleme erläutert werden, die bei der Arbeit mit der Spezifikation auftraten. Auf eine Einführung in die Sprache OCL, wie sie beispielsweise in [3], [1], [12] und [4] zu finden ist, wird an dieser Stelle verzichtet.

2.1 Einschränkungen der Implementierung

2.1.1 Entwurfsmodelle

Die Klassenbibliothek und auch der im weiteren zu entwickelnde Codegenerator sollen für verschiedene CASE Plattformen verwendbar sein. Daraus ergibt sich eine wesentliche Beschränkung: Es können nur OCL-Ausdrücke bearbeitet bzw. umgesetzt werden, die zu einem Entwurfsmodell gehören, also einem UML-Modell, das möglichst direkt in eine Programmiersprache, in diesem Fall Java, überführt werden kann. Zwar sind CASE-Umgebungen häufig auch in der Lage, Quelltexte aus Analysemodellen zu erzeugen, allerdings ist dann stark vom jeweiligen Codegenerator abhängig, wie Modellelemente in Konstrukte der Programmiersprache umgesetzt werden. Eine Anpassung der Klassenbibliothek könnte hier nur für einen speziellen Codegenerator erfolgen.

Eine Einschränkung, die sich hieraus ergibt, ist, daß keine Namenskonflikte durch Vererbung auftreten. In der UML ist es zulässig, daß Attribute oder Assoziationsenden einer Klasse die gleiche Bezeichnung tragen wie Attribute oder Assoziationsenden einer ihrer Oberklassen. In der OCL können die daraus resultierenden Namenskonflikte mit Hilfe der Operation `oclAsType` aufgelöst werden. In einer Javaimplementierung ist das generell nicht möglich. Solche Namenskonflikte sind daher nicht zulässig, und die Operation `oclAsType` wurde aus diesem Grund nicht implementiert.

Desweiteren verfügt die Sprache Java nicht über Aufzählungstypen, wie sie in der OCL durch den Typ `Enumeration` verfügbar sind. Diese müßten daher bei der Überführung in ein Entwurfsmodell entfernt werden. Der OCL-Typ `Enumeration` wird in der Bibliothek nicht repräsentiert.

2.1.2 Grenzen von Java

Einige Ausdrucksmöglichkeiten der OCL können nicht in dieser Klassenbibliothek umgesetzt werden, da die Sprache Java bzw. das dazugehörige Laufzeitsystem hierfür notwendige Informationen nicht zur Verfügung stellen. Hierzu zählen beispielsweise das Sprachelement *@pre* und die Operation *oclIsNew*. Beide dürfen nur in Nachbedingungen von Methoden verwendet werden und bieten Informationen, die die Änderung des Modellzustands im Vergleich zum Zeitpunkt der Vorbedingung betreffen.

Die Operation *allInstances* kann ebenfalls nicht in ein Java-Äquivalent überführt werden. Die Java-Laufzeitumgebung bietet die hierfür nötigen Grundlagen nicht. In speziellen Anwendungsfällen könnte eine Implementierung möglich sein, wenn z.B. bei Auswertungen von Invarianten auf Instanzen des Metamodells auf das Repository einer CASE-Umgebung zurückgegriffen werden kann. Eine Schnittstelle für entsprechende Anpassungen wurde vorgesehen. In diesem Zusammenhang soll auch darauf hingewiesen werden, daß in der OCL-Spezifikation von der Verwendung von *allInstances* abgeraten wird.

Die Sprache Java unterstützt im Gegensatz zu C++ keine parametrisierbaren Typen, obwohl eine entsprechende Erweiterung seit langem in Betracht gezogen wird ([7]). Damit ist eine genaue Umsetzung der OCL-Kollektionen nicht möglich. Es wurde zwar erwogen, das Verhalten solcher Typen dadurch nachzuahmen, daß Elemente bei der Aufnahme in die Kollektion auf ihren Typ überprüft werden, allerdings würde das zu wesentlich schwerer handhabbaren Kollektionen führen und wurde daher nicht realisiert.

Ein weiteres prinzipielles Problem besteht beim Zugriff auf nichtöffentliche Attribute. Da die Auswertung der OCL-Ausdrücke in einer anderen Klasse als der erfolgt, die die zu überprüfenden Attribute enthält, kann nur auf solche Attribute und Methoden zugegriffen werden, die als `public` deklariert sind.

Die Einschränkung, daß in der OCL nur auf solche Operationen der Anwendungsobjekte zugegriffen werden darf, die reine Anfragen an die Objekte darstellen und deren Zustand nicht verändern, kann in der Klassenbibliothek nicht durchgesetzt werden.

2.1.3 Erzeugung von internen Repräsentationen

Wie in Abschnitt 4.1.1 näher erläutert wird, verwendet die Klassenbibliothek eine abstrakte Fabrik ([6]), um interne Repräsentationen von Anwendungsobjekten und -werten zu erzeugen. Hierbei wird eine Abbildung von der Menge der Anwendungsdatentypen auf die Menge der Repräsentationstypen vorausgesetzt. Diese Abbildung kann durch Anpassung der Fabrik geändert werden. Problematischer wird es, wenn beispielweise die Klasse `java.util.ArrayList` explizit im Modell verwendet wird und gleichzeitig zur Repräsentation von Assoziationen dient, da dann im ersten Fall eine Abbildung auf `OclAny` erfolgen müßte, im zweiten Fall aber auf `OclSet` oder `OclSequence`. Da die Existenz einer Abbildung auf OCL-Typen vorausgesetzt wird, könnten bei einer solchen Konstellation OCL-Ausdrücke gar nicht oder nur über eine komplizierte Anpassung der Fabrik ausgewertet werden.

Ein weiteres Problem ergibt sich aus der genannten Fabrikarchitektur, wenn `null`-Referenzen ausgewertet werden sollen. Solche Referenzen haben in Java

keinen zugeordneten Typ, und somit kann der korrekte Repräsentationstyp nicht bestimmt werden. Allerdings dürfen `null`-Referenzen bei exakter Umsetzung eines UML-Modells auch nur bei optionalen Assoziationen auftauchen. Ein Zugriff auf eine solche Assoziation kann in der Klassenbibliothek erkannt werden und resultiert in einer Instanz von `OclSet`, die keine Elemente enthält (siehe 3.3.2).

2.2 Unklarheiten und Fehler in der Spezifikation

Im Laufe dieser Arbeit sind einige Probleme im Zusammenhang mit der OCL-Spezifikation aufgetreten. Diese beruhen teilweise auf Unklarheiten, teilweise aber auch auf Fehlern in der Spezifikation.

2.2.1 Unklarheiten

Allgemein kann gesagt werden, daß die Verwendung natürlicher Sprache zur Definition der OCL die Ursache vieler dieser Probleme ist. Insbesondere die inkonsequente Verwendung von Begriffen wie *object*, *expression* und *value* sollte in einer überarbeiteten Version vermieden werden.

Eine konkrete Unklarheit, die darauf zurückzuführen ist, ist die Beziehung von *OclAny* zu den Basistypen (*String*, *Real*, *Integer* und *Boolean*) in der Typhierarchie. Hier würde eine deutlichere Aussage das Verständnis erleichtern.

Auch zu den undefinierten Werten wären einige Klarstellungen wünschenswert. Daß neben *and* und *or* auch *implies* trotz undefinierter Teilausdrücke ein definiertes Ergebnis haben kann, muß man sich aus der Nachbedingung von *implies* ableiten. Bezüglich *xor* wird diese Fähigkeit auch nach der Untersuchung der Nachbedingung nicht klar. Die Formulierung der Nachbedingung „`(b or b2) and not (b=b2)`“ legt durch ihre Umständlichkeit nahe, daß auch *xor* bei undefinierten Argumenten ein definiertes Ergebnis haben soll (sonst hätte „`not (b=b2)`“ ausgereicht). Bei genauerer Untersuchung muß man allerdings das Gegenteil folgern, da auch der Vergleich (=) mit einem undefinierten Wert ein undefiniertes Ergebnis liefert.

2.2.2 Fehler

Gerade an den Stellen, an denen ein natürlichsprachlich definierter Kern der OCL verwendet wird, um komplexere Ausdrücke zu definieren, und man also größere Präzision erwarten sollte, finden sich einige klare Fehler. Die folgenden sind im Laufe dieser Arbeit aufgefallen und wurden der UML-RTF ([10]) mitgeteilt.

- Die Postfix-Notation der Infix-Operatoren wird inkonsistent verwendet.
- Der Rückgabetyp der Kollektionsoperation *sortedBy* ist laut der natürlichsprachlichen Beschreibung *Sequence*, nach der angegebenen Signatur aber *Boolean*. Die entsprechende Methode in der Klassenbibliothek wurde mit dem Rückgabetyp `OclSequence` vereinbart.
- Die Nachbedingung, die für die Operation *round* des Typs *Real* angegeben wird, ist nicht syntaktisch korrekt, da sie mehr schließende als öffnende Klammern enthält.

- Die Nachbedingung der für Kollektionen definierten Operation *isUnique*,
```
post: result=collection->collect(expr)->forAll(
    e1, e2 | e1<>e2
),
```
ist nur für leere Mengen erfüllt, da die Iteratorvariablen `e1` und `e2` auf dasselbe Element „zeigen" können. Richtig wäre zum Beispiel:
```
post: result=collection->collect(expr)->forAll(
    e|collection->collect(expr)->count(e)=1
)
```

Kapitel 3

Analyse von Implementierungsvarianten

In diesem Kapitel sollen verschiedene Implementierungsvarianten einzelner Aspekte der Klassenbibliothek auf ihre Eignung untersucht werden. Diese Aspekte betreffen die Basistypen, iterierende Methoden und die Einführung eines gemeinsamen Obertyps.

Hier werden auch verschiedene Möglichkeiten beleuchtet, die schließlich nicht realisiert wurden. Für Leser, die sich ausschließlich für die tatsächlich implementierte Lösung interessieren und weniger für deren Begründung, ist dieses Kapitel daher eventuell weniger relevant und kann übersprungen werden.

3.1 Basistypen

3.1.1 Problem

Zur Realisierung der Basistypen von OCL, also `Integer`, `Real`, `Boolean` und `String`, sind verschiedene Ansätze denkbar. Die wesentliche Frage ist, in welchem Umfang die Basistypen in der Klassenbibliothek umgesetzt werden. Die Möglichkeiten liegen dabei zwischen einer vollen Repräsentation in der Bibliothek, bei der zur Laufzeit für jede Instanz eines Basistyps eine Instanz der entsprechenden Bibliotheksklasse angelegt wird, und der vollständigen Umsetzung der Basistypen durch primitive Java-Typen. Bei letzterer Variante bleibt die Umsetzung also dem Generator überlassen.

3.1.2 Varianten

Wrapperobjekte

Für die Basistypen werden eigene Klassen in der Basisbibliothek eingerichtet. Das ist der naheliegende Ansatz. Die einstelligen Operationen werden als parameterlose Instanzmethoden implementiert, die zweistelligen als Instanzmethoden mit einem Parameter.

Ein erstes Problem taucht bei der einzigen dreistelligen Operation auf: if-then-else. Der Rückgabetyp ist laut Spezifikation der Typ des zweiten (then-Zweig) und dritten (else-Zweig) Operanden. Das ließe sich in Java nur so rea-

lisieren, daß der Rückgabewert der if-then-else-Methode die Wurzelklasse der OCL-Bibliothek ist (siehe 3.4). Allerdings muß dann in vielen Fällen eine Typumwandlung auf eine konkrete Unterklasse eingefügt werden. Das widerspricht dem Ziel, den Bau des Generators möglichst einfach zu halten, und macht außerdem Typüberprüfungen durch den Java-Compiler im generierten Quelltext unmöglich.

Klassenmethoden

Als Alternative ist zu erwägen, ob auf eine eigene bibliotheksinterne Repräsentation verzichtet werden kann, indem direkt die Attribute der Anwendungsobjekte verwendet werden. In diesem Fall müßte auf jede Implementierungsmöglichkeit eines Modellelements Rücksicht genommen werden.

OCL-Typ	Implementierungsmöglichkeiten
`Integer`	`int`, `long`, `short` bzw. Wrapperklassen (java.lang.Integer..)
`Boolean`	`boolean`, java.lang.Boolean
`Real`	`float`, `double` bzw. Wrapperklassen
`String`	`java.lang.String`

Die Operationen auf diese Attribute können durch Klassenmethoden realisiert werden. Diese müssen für jede denkbare Kombination der Implementierungsmöglichkeiten zur Verfügung stehen.

```
class Ocl {
  long add(int x, long y) {...}
  long add(int x, int y)  {...}
  ...
}
```

Vorteilhaft ist, daß der OCL-Compiler einerseits nicht wissen muß, um welche Typen es sich konkret handelt. Solange es einer der vorgesehenen Typen ist, erledigt das der Java-Compiler oder die Laufzeitumgebung. Andererseits werden auch keine eigenen Wrapperklassen wie `tudresden.ocl.Integer` benötigt, von denen dann Instanzen angelegt werden müßten. Besonders interessant erscheint die Möglichkeit, if-then-else unter Bewahrung der Typinformationen durch „conditional expressions“ umzusetzen.

Ein großes Problem stellt allerdings die Anzahl der notwendigen Klassenmethoden dar, die sehr schnell mit der Anzahl der Implementierungsmöglichkeiten steigt. Zwar sind diese alle so kurz, daß der Aufwand eventuell vertretbar wäre, im Hinblick auf eine spätere Erweiterung um neue Typen ist dieser Ansatz aber nicht geeignet.

Zu einem weiteren Problem werden hier die undefinierten Werte von OCL, die sich nicht auf die primitiven Javatypen abbilden lassen.

Primitive Datentypen von Java

Auf den ersten Blick erscheint die Möglichkeit, die Umsetzung der Basistypen vollständig dem Generator zu überlassen, sehr verlockend. Es würden also nicht nur wie im vorigen Beispiel die primitiven Datentypen von Java genutzt, sondern außerdem die Operationen der OCL-Typen direkt auf Operationen der primitiven Java-Typen abgebildet. Komplexere OCL-Operationen müßten auf Kombinationen von Java-Operationen abgebildet werden. Der Vorteil ist weniger, daß

die Implementierung der Klassen gespart werden kann, als das laufzeiteffiziente und typsichere Ausnutzen der Sprachmöglichkeiten von Java.

Leider funktioniert dieser Lösungsansatz nur, wenn die Basistypen in den Anwendungsklassen als primitive Typen repräsentiert werden. Wird z.B. eine Kapselklasse wie etwa `java.lang.Integer` zur Implementierung verwendet, wird die Umsetzung durch den Generator sehr schwierig oder unmöglich, da auf diese Kapselobjekte die entsprechenden Operationen nicht anwendbar sind. Wenn diese Einschränkung auch zu verschmerzen wäre, so scheitert der Ansatz doch daran, daß undefinierte Werte nicht repräsentiert werden können.

3.1.3 Lösung

Die Repräsentation der Basistypen durch eigene Klassen scheint sowohl hinsichtlich der Erweiterbarkeit und Anpaßbarkeit als auch wegen der Möglichkeit, undefinierte Werte zu repräsentieren, am ehesten geeignet. If-then-else ließe sich auch bei dieser Lösung auf „conditional expressions" abbilden: (<cond>)?<then-Zweig>:<else-Zweig>. Dadurch bleibt eine Typüberprüfung durch den Java-Compiler möglich, und Typumwandlungen sind hoffentlich seltener nötig.

3.2 Iterierende Methoden

3.2.1 Problem

Als iterierende Methoden werden in dieser Arbeit diejenigen Operationen von Kollektionen bezeichnet, die eine *OclExpression* als Parameter erwarten. Dieser OCL-Ausdruck muß für alle Elemente der Kollektion ausgewertet werden. In Java-Implementierungen ist dafür eine Iteration über diese Elemente notwendig. Beispiele für solche Methoden sind *forAll* und *iterate*.

Für die iterierenden Methoden wurden verschiedene Lösungsansätze untersucht. Hier sollen nur diejenigen dargestellt werden, die sich prinzipiell als tauglich erwiesen haben, die iterierenden Methoden umzusetzen. Als besondere Hürde erwies sich eine Schachtelung von derartigen Operationen wie im folgenden Beispiel. Es bezieht sich wie die meisten Beispiele in dieser Ausarbeitung auf das Klassendiagramm in Abbildung 4.7 auf Seite 33.

```
context Company inv:
self.employee->forAll(e1| self.employee->forAll
          (e2| e1<>e2 implies e1.forename<>e2.forename)
)
```

Die Tauglichkeit der Varianten soll daher an diesem OCL-Ausdruck gezeigt werden.

3.2.2 Iteratorvariante

Wenn Kollektionen eine Möglichkeit bieten, einen Iterator abzufragen, und der parametrisierende OCL-Ausdruck in eine anonyme Innere Klasse („Inner Class“, siehe [8]) überführt wird, läßt sich die oben dargestellte Invariante durch den folgenden Java-Quelltext auswerten. Die im Quelltext vorausgesetzte Klassenbibliothek ist darunter skizziert. Es wird deutlich, daß bei geschachtelten iterierenden Methoden ein Bezug auf Iteratorvariablen beliebiger Schachtelungsebenen möglich ist.

```
OclSet employee=...;
final OclIterator e1=employee.getIterator();
final OclIterator e2=employee.getIterator();
OclBoolean constraintValid=employee.forAll(
  e1,
  new OclBooleanEvaluatable() {
    public OclBoolean evaluate() {
      return employee.forAll(
        e2,
        new OclBooleanEvaluatable() {
          public OclBoolean evaluate() {
            /* ... berechne Ergebnis von
             * "e1<>e2 implies e1.forename<>e2.forename"
             */
          }
        }
      );
    }
  }
);
```

Die Deklaration der Iteratoren als `final` ist notwendig, damit in der Inneren Klasse auf diese Variablen zugegriffen werden darf.

Benutzte Schnittstelle

Die für diesen Lösungsansatz notwendige Schnittstelle der Klassenbibliothek soll nun kurz zusammengefaßt werden. Die Notwendigkeit des hier verwendeten Typs `OclRoot` wird in Abschnitt 3.4 begründet. Vorerst genügt, daß hier die Bibliotheksrepräsentation eines beliebigen Objekts oder Wertes übergeben werden kann.

```
class OclSet:
  OclBoolean forAll(OclIterator, OclBooleanEvaluatable)
  OclIterator getIterator()

class OclIterator:
  OclRoot getValue()

interface OclBooleanEvaluatable:
  OclBoolean evaluate()
```

Alternative mit implizitem Iterator

Eine Alternative wäre, die Verwendung einer *Collection*-Instanz nur mit einem einzigen Iterator zuzulassen. Für das Beispiel oben müßte dann erst eine Kopie von „employee“ angelegt werden. Der Vorteil hier wäre, daß der Iterator nicht an „forAll“ übergeben werden müßte, sondern implizit vorhanden und direkt aus der Kollektion abrufbar wäre.

3.2.3 OCL-Ausdrücke als Objekte

Vergleichend zum Iteratorentwurf wird hier eine Lösung mit als Objekten repräsentierten *OclExpressions* dargestellt. Der Syntaxbaum des OCL-Ausdrucks muß hierfür in eine Baumstruktur aus OCL-Ausdrucksobjekten überführt werden. Zur besseren Übersicht voran nochmals der OCL-Ausdruck.

```
context Company inv:
self.employee->forAll(e1| self.employee->forAll(
                e2| e1<>e2 implies e1.forename<>e2.forename))

OclSet employee=...;
OclExpression iter1=
              new OclIteratingExpression(Ocl.FOR_ALL, employee);
OclExpression iter2=
              new OclIteratingExpression(Ocl.FOR_ALL, employee);
OclExpression exp1=new OclSimpleExpression(
  Ocl.NOT_EQUAL,
  ((Person)iter1.value()).forename,
  ((Person)iter2.value()).forename
);
iter1.setSubExpression(iter2);
iter2.setSubExpression(exp1);
boolean constraintValid=iter1.evaluate();
```

Dieser Lösungsansatz würde eine deutlich aufwendigere Klassenbibliothek erfordern, da neben den vordefinierten Typen auch Elemente der Grammatik repräsentiert werden müßten.

3.2.4 Implementierte Variante

Von den beschriebenen Ansätzen wurde die zuerst dargestellte Iteratorvariante in der Klassenbibliothek umgesetzt. Die Variante mit Ausdrucksobjekten wäre mit deutlich größerem Aufwand verbunden gewesen, und obwohl sie nicht im Detail untersucht wurde, läßt sich abschätzen, daß Probleme aus der Vermischung von Typsystem und Grammatik entstehen würden.

Von der Möglichkeit, bei der jede Kollektion implizit genau einen Iterator enthält, wurde ebenfalls Abstand genommen. Sie erscheint weniger intuitiv und hätte daher bei geschachtelten Aufrufen von iterierenden Methoden einer Kollektion eine potentielle Fehlerquelle dargestellt.

3.2.5 iterate

Abschließend soll die gewählte Implementierungsvariante für die mächtigste der *Collection*-Operationen untersucht werden: für `iterate`.

```
collection->iterate( elem: Type1; acc : Type2 = <expression> |
                     expression-with-element-and-acc )
```

Die Funktionalität von `iterate` läßt sich mit dem beschriebenen Iteratorkonzept realisieren. Allerdings muß die `evaluate`-Methode in diesem Fall nicht `OclBoolean`, sondern ein Objekt vom Typ `OclRoot` zurückliefern. Außerdem wird eine zusätzliche Klasse (`OclContainer`) benötigt, die die Akkumulatorvariable repräsentiert.

```
class OclContainer:
  OclContainer(OclRoot value)
  protected void setValue(OclRoot value)
  OclRoot getValue()

class OclCollection:
  OclRoot iterate(OclIterator iter, OclContainer acc,
                                  OclRootEvaluateable expr) {
    // vereinfacht:
    while (iter.hasNext()) {
      acc.setValue(expr.evaluate());
    }
    return acc.getValue();
  }
```

Die Klasse `OclContainer` hat nur die Aufgabe, eine Referenz auf ein Objekt zu halten und diese verändern zu lassen. Ohne diese Indirektion würde bei jedem neuen Aufruf von `expr.evaluate()` wieder der Initialisierungswert der Akkumulatorvariable verwendet.

In der `evaluate`-Methode von `OclRootEvaluateable` kann sowohl auf die Akkumulator- als auch auf die Iteratorvariable zugegriffen werden.

Zur Verdeutlichung hier ein Beispiel:

```
collection->iterate(iter: Type1; acc: Type2 = Bag{} |
                    acc->including(iter.property))

final OclIterator iter;
final OclContainer acc;
Bag b=collection.iterate(
  iter=collection.getIterator(),
  acc=new OclContainer(new Bag()),
  new OclRootEvaluatable() {
    public OclRoot evaluate() {
      /* Java-Äquivalent für acc->including(iter.property)
       * berechnen und zurückgeben
       */
    }
  }
);
```

3.3 Anwendungsklassen

3.3.1 Problem

Nach den Kollektionen und den Basistypen bleibt noch eine Kategorie von OCL-Typen offen: Die in OCL verwendeten Klassen des Anwendungsmodells.

Die Realisierung dieser Klassen muß zwei grundsätzliche Ansprüche erfüllen: Einerseits müssen die Attribute der Anwendungsklassen zugänglich sein, beispielsweise im Standardbeispiel das Attribut „forename“ der Klasse „Person“. Andererseits sollen auch die in der OCL-Spezifikation für *OclAny* definierten Eigenschaften wie z.B. *oclIsKindOf* auswertbar sein. Außerdem wäre es für die automatische Generierung von Javarepräsentationen für OCL-Ausdrücke eine erhebliche Erleichterung, wenn keine Typkonvertierungen nötig sind, für deren Erstellung Informationen über das Anwendungsmodell notwendig sind, die nicht aus dem OCL-Ausdruck abgelesen werden können. In jedem Fall ist zu beachten, daß es unmöglich ist, für jede Anwendungsklasse eine eigene OCL-Variante vorzusehen (also eine Klasse `OclPerson` für `Person` usw.), es sei denn, auch diese Klassen werden automatisch generiert. Diese Möglichkeit erscheint nicht praktikabel und wurde nicht näher untersucht.

3.3.2 Varianten

Direkte Verwendung der Anwendungsklassen

Die naheliegenste Variante ist, die Java-Implementierungen der Anwendungsklassen, die sowieso vorhanden sind, direkt zu verwenden. Auf Attribute der Anwendungsklassen kann dann direkt zugegriffen werden. Problematischer sind die *OclAny*-Eigenschaften. Diese könnten über Klassenmethoden einer festen Klasse realisiert werden, denen das betreffende Anwendungsobjekt als Parameter übergeben wird.

Das wesentliche Problem, daß bei dieser Variante auftritt, ist die Notwendigkeit von Typkonvertierungen. In vielen Fällen ist der genaue Typ eines Objekts nicht bekannt, beispielsweise wenn es aus einer Kollektion entnommen wurde. Um auf ihre Attribute zuzugreifen, muß eine Typkonvertierung auf die entsprechende Anwendungklasse eingefügt werden. Da der Typname nur in Einzelfällen aus dem OCL-Ausdruck ersichtlich ist, sind somit Informationen über das UML-Modell notwendig.

Kapselung über Instanzen von `OclAnyImpl`

Bei diesem Ansatz wird eine Klasse `OclAnyImpl` verwendet, deren Instanzen im wesentlichen die Aufgabe haben, eine Referenz auf das Anwendungsobjekt zu halten. Außerdem bietet diese Klasse die in der OCL-Spezifikation für den gleichnamigen Typ festgelegten Methoden wie z.B. `oclIsKindOf`. Für den Zugriff auf Attribute der Anwendungsklassen wird eine Methode `getFeature(String name)` vereinbart, die das entsprechende Attribut mit Hilfe von Java Reflection ([8]) abfragt.

```
public class OclAnyImpl {
  public OclBoolean isEqualTo(OclAnyImpl any) { ... }
  public OclBoolean isNotEqualTo(OclAnyImpl any) { ... }
  public OclBoolean oclIsKindOf(OclType type) { ... }
  public OclBoolean oclIsTypeOf(OclType type) { ... }
  public OclRoot getFeature(String name) { ... }
}
```

Der große Vorteil dieser Variante ist, daß keine Typinformation zur Generierung der Aufrufe benötigt wird, da der Typ der Anwendungsklassen nicht bekannt sein muß. Daher wurde diese Implementierungsvariante vorgezogen.

Beispiel

Der schon oben verwendete Beispielausdruck soll nun um einen derartigen Zugriff über `OclAny` erweitert werden:

```
context Company inv:
self.employee->forAll(e1 | self.employee->forAll
          (e2 | e1<>e2 implies e1.forename<>e2.forename))

final OclSet employee=...;
final OclIterator e1=employee.getIterator();
final OclIterator e2=employee.getIterator();
OclBoolean constraintValid=employee.forAll(
  e1,
  new OclBooleanEvaluatable() {
    public OclBoolean evaluate() {
      return employee.forAll(
        e2,
        new OclBooleanEvaluatable() {
          public OclBoolean evaluate() {
            return e1.getValue().isNotEqualTo(e2.getValue()).
              implies(
                e1.getValue().getFeature("forename").
                  isNotEqualTo(
                    e2.getValue().getFeature("forename")
                  )
              );
          }
        } // end inner class
      );
    }
  } // end inner class
);
```

Ein besonderes Problem stellt in diesem Zusammenhang die von OCL angebotene Möglichkeit dar, einfache Assoziationen als *Set* zu verwenden. Im folgenden Beispiel ist daher der Teilausdruck `self.husband` einmal vom Typ *Set* und einmal vom Typ *Person*.

```
context Person inv:
self.husband->size=1 implies self.husband.age>=18
```

Die Unterscheidung ist nur dadurch möglich, daß in dem Fall, in dem die Navigation über die Assoziation in ein *Set* resultieren soll, ein durch „`->`“ gekennzeichneter Zugriff auf eine Kollektionseigenschaft auf den Teilausdruck folgt. Daraus ergibt sich die Notwendigkeit, eine Methode `getFeatureAsCollection` zur Klasse `OclAnyImpl` hinzuzufügen. Diese Methode wird statt `getFeature` aufgerufen, falls für das Ergebnis eine Kollektionseigenschaft ausgewertet werden soll. Sie stellt sicher, daß das zurückgegebene Objekt vom Typ `OclCollection` ist.

3.4 Notwendigkeit einer gemeinsamen Wurzelklasse

3.4.1 Problem

In der OCL-Spezifikation wird keine gemeinsame Wurzelklasse für die Kollektionen, die Basistypen und die Anwendungsklassen festgelegt[1]. An dem folgenden Beispiel, das absichtlich nicht dem sonst in dieser Arbeit verwendeten Kontext entnommen ist, wird aber deutlich, daß eine solche Wurzelklasse notwendig ist, wenn der Generator nicht unnötig von Informationen über das Anwendungmodell abhängig sein soll:

```
context Machine inv:
self.control.button->size < 20
```

Diese Invariante sagt aus, daß eine Maschine nicht mehr als 20 Knöpfe haben darf. Ohne das dazugehörige Klassendiagramm ist aber nicht entscheidbar, ob die Maschine nur ein `control`-Element hat und `button` damit ein Zugriff auf ein Assoziationsende der einzigen Kontrolleinheit ist, oder ob die Maschine viele `control`-Elemente hat und der obige Ausdruck eine Kurzschreibweise für den folgenden Ausdruck darstellt:

```
context Machine inv:
self.control->collect(button)->size < 20
```

Unklar bleibt auch, ob ein `control`-Element in diesem Fall über einen oder mehrere `Buttons` verfügt.

Somit kann nicht bestimmt werden, ob der Rückgabewert von `getFeature("control")` eine Kollektion oder ein Anwendungsobjekt sein wird. Eine Entscheidung dieser Frage ist nur mit Hilfe von Kontextwissen möglich, die ein automatisierter Generator aus dem Anwendungsmodell abfragen müßte.

[1] `OclAny` ist ausdrücklich nicht als Supertyp der Kollektionen spezifiziert.

3.4.2 Lösung

Um die Kopplung zwischen Generator und Anwendungsmodell gering zu halten, darf die Entscheidung zwischen Kollektion und Anwendungsobjekt erst zur Laufzeit gefällt werden. Als Rückgabetyp der Methode `getFeature(String name)` wird `OclRoot` vereinbart, die gemeinsame abstrakte Oberklasse von Kollektionen, Basistypen und `OclAnyImpl`. Zum Zugriff auf `button` im Beispiel oben wird die Methode `getFeatureAsCollection(String name)` verwendet, die sowohl für Kollektionen als auch für Anwendungsobjekte verfügbar sein muß. Für Instanzen von `OclAnyImpl` wird diese Methode wie schon beschrieben implementiert, für Kollektionen wird ein `collect` ausgeführt. Alle oben beschriebenen Interpretationsvarianten des OCL-Ausdrucks können somit wie folgt in Java realisiert werden:

```
OclAny self=...;
OclBoolean constraintValid=
  self.getFeature("control").getFeatureAsCollection("button").
    size().isLessThan( new OclInteger(20) );
```

Um bestimmte Implementierungsmöglichkeiten der Basisbibliothek nicht von vornherein auszuschließen, ist es vorteilhaft, `OclRoot` nicht wirklich als Klasse, sondern als Schnittstelle (*Interface*) zu vereinbaren.

Es ist ein großer Vorteil dieser Lösung, daß es bei der automatischen Generierung möglich ist, jeden Ausdruck der Form `.<attr>` in `.getFeature("<attr>")` oder `.getFeatureAsCollection("<attr>")` umzuwandeln. Auch die Unterscheidung, welche dieser beiden Möglichkeiten zu wählen ist, kann rein auf Basis des OCL-Ausdrucks getroffen werden.

Kapitel 4

Entwurf und Implementierung

Die im vorangegangenen Kapitel dargestellten und begründeten Entscheidungen legen die Grobarchitektur der Klassenbibliothek weitgehend fest. Im weiteren sollen die dargestellten Lösungen von Einzelaspekten zu einem Gesamtbild zusammengefügt und detaillierter erläutert werden.

4.1 Anpaßbarkeit

Ein wesentliches Entwurfsziel ist die Anpaßbarkeit der OCL-Bibliothek an Java-Quelltexte, die von verschiedenen Codegeneratoren erzeugt werden. Diese Anpaßbarkeit wird bezüglich drei getrennter Aspekte ermöglicht: Die Erzeugung von Repräsentationen für Anwendungsobjekte bzw. -werte, die Umwandlung von Namen von Modellelementen in Namen von Implementierungselementen sowie der Zugriff auf Objektzustände.

4.1.1 Erzeugung interner Repräsentationen

Verschiedene Codegeneratoren verwenden unterschiedliche Klassen, um bestimmte Aspekte eines UML-Modells zu repräsentieren. Insbesondere bei der Umsetzung von Assoziationen sind etliche Lösungen denkbar, wie z.B. Arrays, Vektoren oder auch proprietäre Klassen. Wie im vorangegangenen Kapitel beschrieben worden ist, werden Anwendungsobjekte in der Bibliothek nicht direkt verwendet, sondern jeweils eigene, interne Repräsentationen erzeugt. Die korrekte Erzeugung dieser Repräsentationen setzt eine Anpassung auf den jeweils verwendeten Codegenerator voraus.

Diese Anpassung wird über eine abstrakte Fabrik ermöglicht, die die Schnittstelle `OclFactory` implementiert. Sie muß Methoden namens `getOclRepresentationFor` für alle möglichen Parameter bereitstellen, also für primitive Java-Typen und für `java.lang.Object`. Innerhalb der Methode `getOclRepresentationFor(Object)` sollte das übergebene Objekt mit Hilfe des Java-Operators `instanceof` auf seinen Typ untersucht werden und die diesem Typ entsprechende Repräsentation erzeugt werden. Das Überladen der Methode für Unterklassen von `Object` führt dagegen nicht zum gewünschten Er-

gebnis, da die Java-Laufzeitumgebung unter Umständen auch für einen Parameter, der einer solchen Unterklasse angehört, die Implementierung für `Object` aufrufen würde.

Wenn in einem OCL-Ausdruck der Rückgabewert einer Methode eines Anwendungsobjekts ausgewertet wird, so muß in dem Java-Quelltext, der diesem Ausdruck entspricht, ein Aufruf dieser Methode eingefügt werden. Da die Parameter der Methode im OCL-Ausdruck selbst wieder OCL-Ausdrücke sind, ist eine Rekonvertierung von bibliotheksinternen Repräsentationen in Anwendungsobjekte notwendig. Von der Existenz einer eindeutigen Abbildung von Repräsentationstypen auf Anwendungstypen kann hier nicht ausgegangen werden. Daher wird in `OclFactory` eine Methode `reconvert` vereinbart, die außer dem umzuwandelnden `OclRoot`-Objekt auch noch eine Instanz von `java.lang.Class` übergeben bekommt, die den gewünschten Zieltyp anzeigt. Dieser Zieltyp kann über *Reflection* aus der Signatur der Methode abgelesen werden.

Mit `DefaultOclFactory` enthält die Klassenbibliothek eine Implementierung von `OclFactory`, die auf den Codegenerator von Argo/UML abgestimmt ist. Sie sollte aber auch für einige andere Softwareentwicklungsumgebungen ausreichen, sofern diese Assoziationen auf Objekte der Klasse `java.lang.Vector` abbilden.

Zugriff auf die Fabrik

Aus der Bibliothek wird über Klassenmethoden der Klasse `Ocl` auf die Fabrik zugegriffen, wenn eine neue Repräsentation erzeugt werden soll. Dadurch wird beim Zugriff auf Attribute von Anwendungsobjekten über die Methode `getFeature(String)` jeweils diejenige Repräsentationsklasse gewählt, die in der `OclFactory` vorgesehen wurde. Die Klasse `Ocl` bietet außerdem eine Methode, über die eine neue Fabrik registriert werden kann.

4.1.2 Namensanpassung

Codegeneratoren können bei der Umsetzung von Modellelementen deren Namen verändern. So repräsentiert der Codegenerator von Argo/UML nicht explizit benannte Assoziationsenden, die in OCL mit dem kleingeschriebenen Klassennamen angesprochen werden, durch Instanzattribute namens `my<Klassenname>`. Um eine Anpassung dieser Namensänderung zu ermöglichen, kann die Klasse `Ocl` eine Referenz auf ein Objekt halten, die die Schnittstelle `NameAdapter` implementiert. In dieser Schnittstelle ist eine Methode `getNames(String)` definiert, die ein Stringarray mit allen möglichen Repräsentationsnamen zurückgibt. In der Klassenbibliothek wird beim Zugriff auf ein Assoziationsende eines Objekts dieses Array abgefragt und dann über *Java Reflection* untersucht, für welchen der möglichen Namen ein Feld des Objekts existiert.

Wie schon bei der Fabrik ist auch hier wieder eine inverse Abbildung notwendig. Sie wird verwendet, wenn über die OCL-Metaebene die Namen der Assoziationsenden abgefragt werden. Mit Hilfe der Methode `getPossibleAssociationNames(String)` können die Modellnamen abgefragt werden, die in einen über *Reflection* gefundenen Implementierungsnamen resultiert haben könnten.

Die Klasse `ArgoNameAdapter` stellt eine Implementierung der Schnittstelle `NameAdapter` für den Codegenerator von Argo/UML dar.

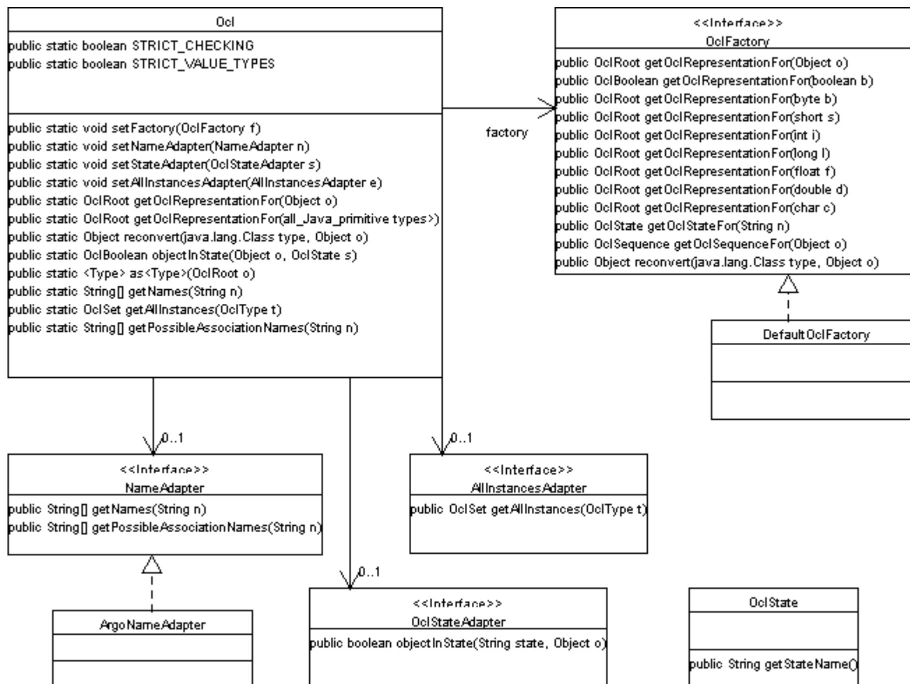

Abbildung 4.1: Die Klasse `Ocl`

4.1.3 Weitere Anpassungsmöglichkeiten

In der OCL 1.3 ist es möglich, Zustandinformationen von Objekten über die Operation `oclInState(OclState)` abzufragen. Wenn eine generatorspezifische Implementierung der Schnittstelle `OclStateAdapter` bei der Klasse `Ocl` registriert wird, kann diese Auswertung durch die Klassenbibliothek durchgeführt werden. Die genannte Schnittstelle definiert die Methode `objectInState(String, Object)`, der der Name eines Zustands sowie das fragliche Objekt übergeben werden.

Die schon in Abschnitt 2.1 angesprochene Anpassungsmöglichkeit, die die Auswertung der Operation *allInstances* ermöglichen soll, besteht in der Schnittstelle `AllInstancesAdapter`. Hier kann eine Methode implementiert werden, die für ein Objekt vom Typ `OclType` ein `OclSet` mit allen Instanzen zurückgibt.

4.1.4 Zusammenfassung

Das Klassendiagramm in Abbildung 4.1 zeigt die Klasse `Ocl` sowie die Fabrik- und Adapterklassen.

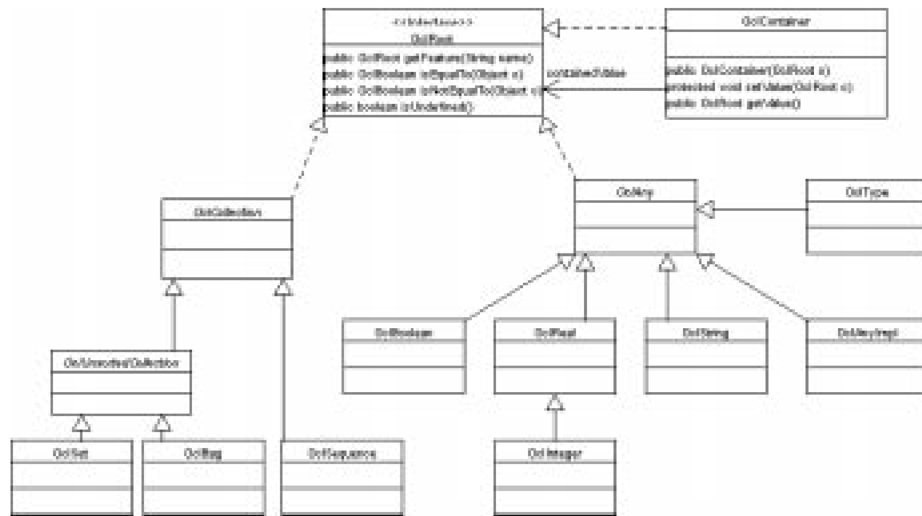

Abbildung 4.2: Übersicht über die Basisbibliothek

4.2 Hierarchie

Abbildung 4.2 zeigt die Vererbungshierarchie der Repräsentationen der vordefinierten OCL-Typen in der Javabibliothek. Die Schnittstelle `OclRoot`, die von allen abgebildeten Klassen direkt oder indirekt implementiert wird, definiert die Methoden `getFeature(String)` und `getFeatureAsCollection(String)`. Je nach der implementierenden Klasse realisiert diese Methode entweder den Zugriff auf Attribute (bei `OclAny`) oder die abkürzende Schreibweise für *collect*. Außerdem werden die Vergleichsmethoden `isEqualTo(Object)` und `isNotEqualTo(Object)` vereinbart. Diese Methoden sind der in `Object` definierten Methode `equals` sehr ähnlich. Es ist jedoch nicht möglich, die bestehende Methode `equals` für die Umsetzung des OCL-Operators „=" zu verwenden, da sie `boolean` und nicht `OclBoolean` als Rückgabewert hat.

Die abstrakte Klasse `OclAny` definiert entsprechend dem Typ *OclAny* Methoden, die in allen OCL-Objekten bis auf den Kollektionen verfügbar sind. `OclAnyImpl` implementiert diese dann für die Kapselung von Anwendungsobjekten.

4.3 Undefinierte Werte

OCL-Ausdrücke können undefinierte Abfragen enthalten. Die OCL-Spezifikation spricht in diesem Fall von undefinierten Ausdrücken, in der Implementierung ist die Bezeichnung undefinierter Wert für das Ergebnis eines solchen Ausdrucks treffender. Die meisten OCL-Ausdrücke, die einen undefinierten Teilausdruck enthalten, sind selbst undefiniert. Für die Javabibliothek bedeutet das, daß jede Methode einen undefinierten Wert zurückgeben kann. Es ist somit nicht möglich, undefinierte Werte als eigene Unterklasse von `OclRoot` oder als Instanz einer beliebigen bestehenden Unterklasse zu realisieren, da sonst die Repräsentation des undefinierten Wertes dem Rückgabetyp jeder einzelnen Methode entsprechen müßte.

Andererseits können diese undefinierten Werte auch nicht durch Ausnahmen (*Exceptions*) ausgedrückt werden, da die Booleschen Operationen *and* und *or* auch bei undefinierten Operatoren definierte Werte zurückgeben können.

Als Lösungsmöglichkeit bleibt, zu `OclRoot` eine Methode `boolean isUndefined()` hinzuzufügen. Diese soll `true` zurückgeben, falls es sich um einen undefinierten Wert handelt. Ungünstig ist, daß in jeder Methode der Klassenbibliothek explizit abgefragt werden muß, ob es sich bei der aufgerufenen Instanz oder einem der Parameter um einen undefinierten Wert handelt. Dem Anwender der Klassenbibliothek bleibt dieses Problem aber verborgen.

Typkonvertierung

Wenn `null`-Referenzen in Bibliotheksrepräsentationen umgewandelt werden müssen, ist es nicht immer möglich, den undefinierten Wert des korrekten Typs zurückzugeben. Darum kann es bei Typkonvertierungen zu Fehlern kommen, die `ClassCastExceptions` auslösen. Als Beispiel soll der folgende OCL-Ausdruck betrachtet werden, der sich auf das Klassendiagramm in Abbildung 5.1 auf Seite 39 bezieht:

```
context Membership
inv: loyaltyAccount.points>=0 or loyaltyAccount->isEmpty
```

Falls das optionale Assoziationsende `loyaltyAccount` nicht belegt ist, wird für den Zugriff auf `points` ein undefinierter Wert vom Typ `OclAnyImpl` zurückgegeben. Die Typkonvertierung auf `OclComparable`, die für den Vergleich nötig ist, würde eine `Exception` auslösen. Somit wäre diese Invariante, die eigentlich erfüllt ist, nicht auswertbar.

Die Lösung dieses Problems besteht in den in Abbildung 4.1 dargestellten Konvertermethoden `to<Typname>(OclRoot)`, die versuchen, das übergebene Objekt in den Zieltyp zu konvertieren und im Fall eines Fehlers den undefinierten Wert des entsprechenden Typs zurückgeben.

4.4 Kollektionen

4.4.1 Iterierende Methoden

Zur Realisierung der iterierenden Methoden (*forAll*, *iterate*,...) hat sich ein Iteratorkonzept als am ehesten geeignet erwiesen (siehe Abschnitt 3.2). Iteratoren können von Kollektionen abgefragt werden, und werden dann derselben Kollektion als ein Parameter einer iterierenden Methode übergeben. Jeder Iterator darf dabei nur einmal verwendet werden. Sein Wert kann mit der Methode `getValue()` abgefragt werden.

Der OCL-Ausdruck, der an die iterierende Methode als weiterer Parameter übergeben wird, wird durch eine Methode `evaluate` einer `Evaluatable`-Schnittstelle realisiert. Der Java-Quelltext, der den OCL-Ausdruck umsetzt, bildet dabei den Körper der Methode. Da der Rückgabewert von `evaluate()` meistens `OclBoolean` sein muß (für `forAll`, `exists`, `select`, `reject`), manchmal aber auch `Object` (für `isUnique`, `iterate`, `collect`) und in einem Fall auch `java.lang.Comparable` (für `sortedBy`), muß es drei solche Schnittstellen geben: `OclBooleanEvaluatable`, `OclObjectEvaluatable` und

`OclComparableEvaluatable`, mit dem entsprechenden Rückgabetyp für die jeweils einzige Methode `evaluate()`. Diese Schnittstellen können bei der Generierung jeweils durch eine anonyme *Inner Class* implementiert werden.

Für die Akkumulatorvariable der Operation *iterate* wird außerdem eine Klasse `OclContainer` bereitgestellt (siehe Abschnitt 3.2.5).

4.4.2 Spezifikationsabweichungen in `OclCollection`

In der OCL-Spezifikation werden die Operationen *collect*, *select*, *reject*, *excluding* und *including* für *Set*, *Bag* und *Sequence* definiert, aber nicht für die abstrakte Oberklasse *Collection*. Diese Vorgabe wird in der Basisbibliothek nicht eingehalten: Schon in `OclCollection` werden diese Methoden vereinbart. Eine ähnliche, noch weitergehende Veränderung betrifft die Operationen *asSet*, *asBag* und *asSequence*. Diese sind in der OCL Spezifikation für alle konkreten Kollektionen definiert, jedoch fehlt jeweils diejenige Operation, die eine Kollektion auf eine Kollektion des gleichen Typs abbildet (z.B. `set->asSet`). In der Klassenbibliothek werden diese Operationen hinzugefügt und sämtlich in `OclCollection` vereinbart.

Durch diese beiden Abweichungen von der Spezifikation wird ermöglicht, die genannten Methoden bei Instanzen beliebiger Kollektionen aufzurufen. Andernfalls müßte eine Typkonvertierung auf die konkrete Kollektion durchgeführt werden, die nicht aus dem Namen der Operation abgeleitet werden kann und damit nicht ohne weiteres bekannt ist.

Eine entsprechende Abweichung von der Spezifikation ergibt sich auch für die Operation *union*, allerdings mit einem zusätzlichen Problem. Diese Operation ist in *Set* und *Bag* für einen Parameter vom Typ *Set* oder *Bag* definiert, in *Sequence* dagegen nur für einen Parameter vom Typ *Sequence*. Also ist beispielsweise `set->union(sequence)` nicht zulässig. Wenn `union` als Methode in der Oberklasse `OclCollection` vereinbart werden soll, kann das allgemein nur für einen Parameter vom Typ `OclCollection` geschehen. Damit ist `oclSet.union(oclSequence)` für den Java Compiler ein gültiger Aufruf. Eine Fehlermeldung kann demnach erst zur Laufzeit erfolgen. Dennoch wird die Methode `union` wie beschrieben in `OclCollection` deklariert, da sonst die automatische Generierung an den Stellen sehr erschwert würde, an denen für eine Kollektion unbekannten Typs die Methode `union` aufgerufen werden soll. Die unspezifizierten Methoden werden so implementiert, daß sie eine `Exception` auslösen.

4.4.3 `OclUnsortedCollection` und Schnittstellen

Die Klasse `UnsortedCollection` wird eingeführt, um zu verhindern, daß sich an einer anderen Stelle das gleiche Problem ergibt, wie eben für die Methode `union` geschildert. Die Operation *intersection* ist für die Typen *Set* und *Bag* definiert und akzeptiert jeweils Objekte beider Typen als Parameter. Wenn auf `OclUnsortedCollection` verzichtet und `intersection` in `OclCollection` deklariert würde, könnte ein fehlerhafter Aufruf der Methode, an dem eine Instanz von `OclSequence` beteiligt ist, erst zur Laufzeit entdeckt werden. Durch die Einführung von `OclUnsortedCollection` werden solche Fehler vom Compiler erkannt.

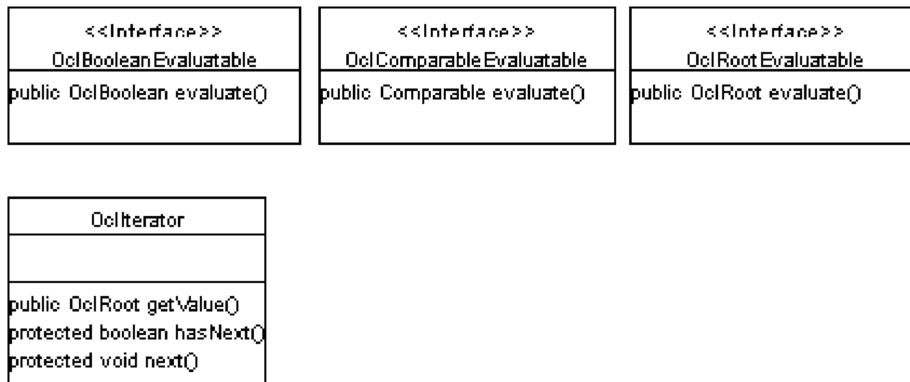

Abbildung 4.3: Klassendiagramm für die `Evaluatable`-Schnittstellen und `OclIterator`

`OclSizable` ist nötig, da die Operation *size* für die OCL-Typen *String* und *Collection* definiert ist. Durch Einführung dieser Schnittstelle, die von `OclString` und `OclCollection` implementiert wird, kann bei der automatischen Generierung in jedem Fall, in dem im OCL-Ausdruck ein Aufruf von *size* vorgefunden wird, eine Typkonvertierung auf `OclSizable` eingefügt werden.

Ähnliches gilt für die Schnittstelle `OclSubtractable`: Sie definiert die Methode `subtract`, die der OCL-Operation „-“ entspricht. Diese Operation ist außer für *Set* auch für *Real* und *Integer* spezifiziert.

4.4.4 Zusammenfassung

In den Abbildungen 4.3 und 4.4 ist der komplette Entwurf der Kollektionen zusammengefaßt. Dabei zeigt Abbildung 4.3 das Klassendiagramm für die Hilfsklassen, Abbildung 4.4 das für die Kollektionen selbst.

4.5 Basistypen

4.5.1 Klassen und Operationen

In der OCL-Spezifikation werden einfache Basistypen und dazugehörige Operationen festgelegt. Diese Basistypen sind *Integer*, *Real*, *Boolean* und *String*. Die Überlegungen haben gezeigt, daß auch für diese Basistypen eigene Klassen in der Basisbibliothek nötig sind.

Die Operationen auf die Basistypen werden durch Instanzmethoden der Klassen realisiert. Eine Sonderregelung wird für die *if-then-else*-Operation, die für *Boolean* definiert ist, getroffen: Diese Operation läßt sich am geeignetsten durch den `<cond>?<then>:<else>`-Ausdruck in Java umsetzen, da so der Typ des OCL-Ausdrucks erhalten bleibt. Der Vollständigkeit halber wird die Operation außerdem als Methode von `OclBoolean` implementiert.

4.5.2 Schnittstellen

Einige Operationen wie z.B. „`+`“ und „`*`“ können sowohl *Real*- als auch *Integer*-Werte als Parameter haben. Falls diese Werte einer Kollektion entnommen

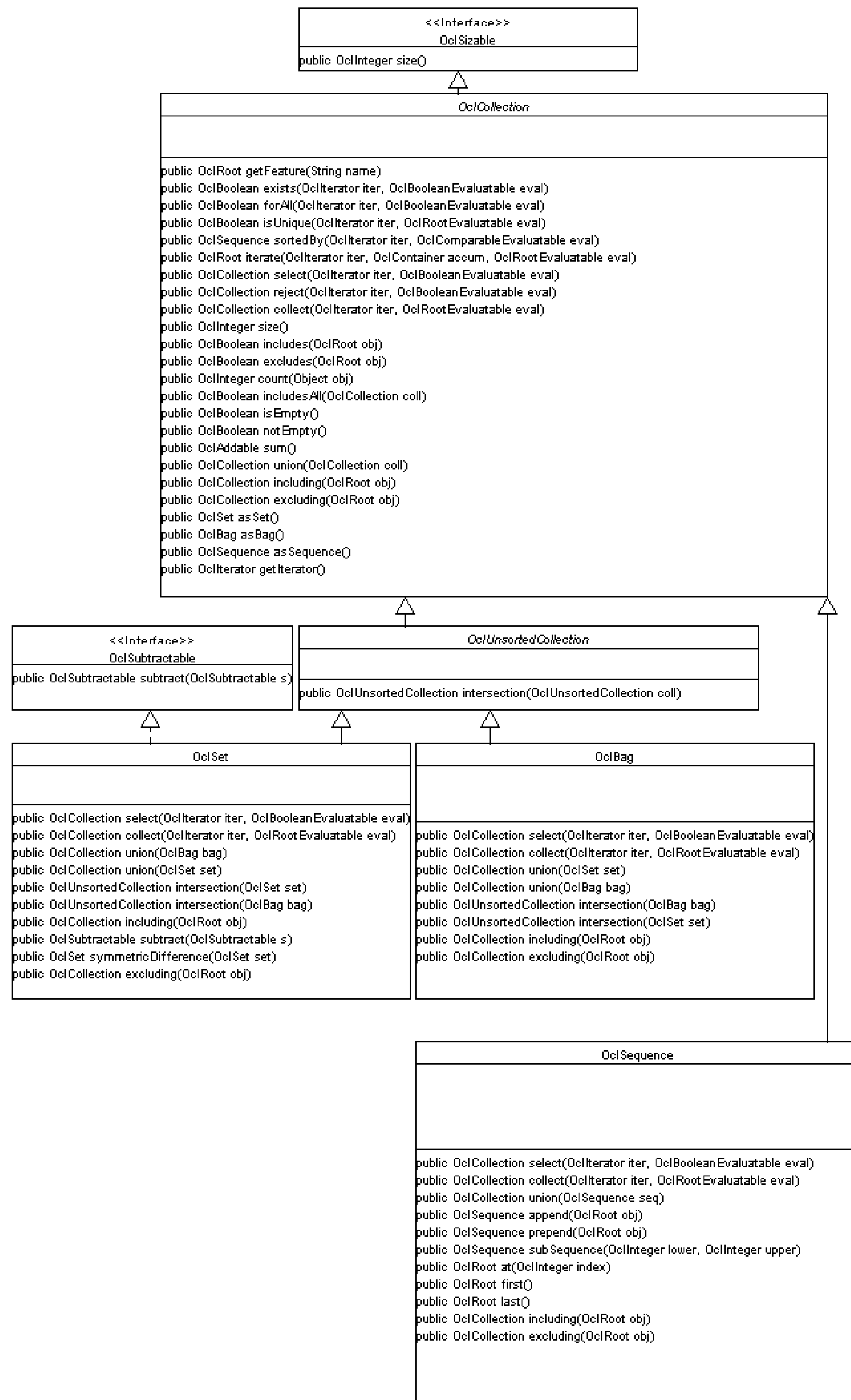

Abbildung 4.4: Klassendiagramm für die Kollektionen

werden, wird eine Typkonvertierung nötig. Obwohl eine Typkonvertierung auf `OclReal` ausreichend wäre, werden Schnittstellen für addierbare (`OclAddable`) bzw. multiplizierbare (`OclMultiplyable`) Werte eingeführt. Das kann sich als vorteilhaft erweisen, falls in einer folgenden Version der OCL Spezifikation beispielsweise der Operator „+“ auch für *Strings* zugelassen wird. Für subtrahierbare Klassen ist eine getrennte Schnittstelle nötig, da für den Typ *Set* zwar „-“, aber nicht „+“ definiert ist.

Eine Schnittstelle für vergleichbare Werte ist mit `java.lang.Comparable` schon in der Standardbibliothek von Java enthalten. Obwohl `OclReal` auch diese implementiert, wird dennoch eine eigene Schnittstelle `OclComparable` vereinbart, die besser der OCL-Spezifikation entspricht. `java.lang.Comparable` wird in der Bibliothek aber an anderer Stelle verwendet, nämlich als Parameter der Methode `sortedBy` von `OclCollection`, wo diese Schnittstelle eine größere Flexibilität in Hinsicht auf eine OCL-Erweiterung um vergleichbare Anwendungsobjekte bietet (also solche, die über Operationen „<“ und „>“ verfügen).

Die Operation *size* ist sowohl für Kollektionen als auch für *String* definiert. Wie schon in Abschnitt 4.4.3 beschrieben, wird daher die Schnittstelle `OclSizable` eingeführt.

Bei der automatischen Generierung kann vor jedem Aufruf einer solchen Operation eine Typkonvertierung auf die entsprechende Schnittstelle eingefügt werden, also z.B. Konvertierung zu `OclComparable` für den Parameter von `isGreaterThan`. Allerdings können dabei vom Compiler einige Fehler nicht erkannt werden. So würde bespielsweise eine Subtraktion eines `OclSets` von einem `OclInteger` erst zur Laufzeit erkannt und dann mit einer *Exception* gemeldet.

4.5.3 Zusammenfassung

Die Schnittstelle der Klassenbibliothek bezüglich der Basistypen ist in Abbildung 4.5 dargestellt.

4.6 Klassen des Anwendungsmodells

4.6.1 Zugriff auf Attribute und Methoden

Anwendungsobjekte werden in Instanzen von `OclAnyImpl` gekapselt. Auf ihre Attribute wird mit `getFeature(String)` bzw. `getFeatureAsCollection(String)` zugegriffen.

Wenn Methoden von Anwendungsobjekten in OCL-Ausdrücken ausgewertet werden, müssen zusätzlich Parameter übergeben werden. Dafür ist in `OclAnyImpl` die Methode `getFeature(String, Object[])` vorgesehen. Die Parameter werden als Elemente des Feldes von Objekten übergeben. Diese Methode braucht nicht in `OclRoot` oder `OclAny` definiert zu werden, da eine Typkonvertierung auf `OclAnyImpl` eingefügt werden kann, wenn ein Methodenaufruf erkannt worden ist.

Um aus der OCL-Basisbibliothek heraus Methoden von Anwendungsobjekten aufrufen zu können, müssen die Parameter des Aufrufs, die eigentlich als OCL-Repräsentationen zur Verfügung stehen, wieder in ihre ursprüngliche Repräsentation zurückverwandelt werden. Die Klasse `Ocl` bietet hierfür die

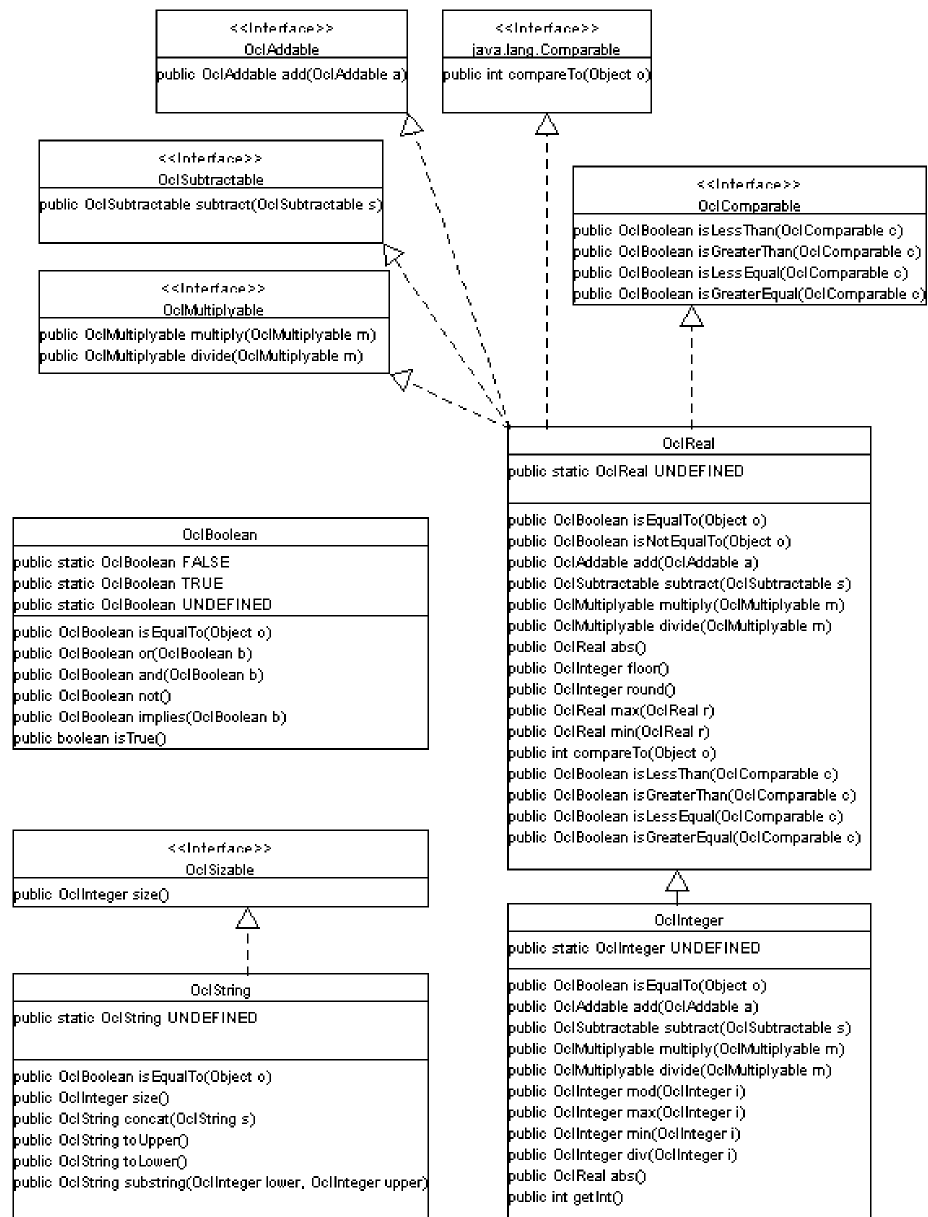

Abbildung 4.5: Klassendiagramm für die Basistypen

Klassenmethode `reconvert` an. Aufrufe dieser Methode werden auch an das `OclFactory`-Objekt weitergeleitet.

4.6.2 Metamodell

Über Attribute von `OclAny` kann auf die Metaebene von OCL zugegriffen werden. Die Klasse `OclType` entspricht hierbei dem Typ *OclType*, der in der OCL-Spezifikation festgelegt ist. Wie schon in 2.1 beschrieben wurde, läßt sich die Operation *allInstances* in Java nicht mit angemessenem Aufwand realisieren und wird daher nicht in die Klassenbibliothek übernommen.

In OCL-Ausdrücken ist es möglich, Werte vom Typ *OclType* durch direkte Benennung anzugeben. Wenn solche Ausdrücke in Java-Quelltext umgewandelt werden, ist es aber nicht möglich, aus dem einfachen Klassennamen den genauen Typ zu bestimmen. In den für solche Aufgaben vorgesehenen Methoden der Java-Standardbibliothek ([9]) ist dafür ein vollqualifizierter Klassenname, d.h. inklusive Packagebezeichnung, notwendig. Darum verfügt die Klasse `OclType` über eine Fabrikmethode, der neben dem einfachen Klassennamen auch noch ein beliebiges Objekt übergeben wird. In der Fabrikmethode wird dann versucht, eine Klasse des gegebenen Namens in der Package des übergebenen Objekts zu finden. In vielen Fällen kann für dieses Bezugsobjekt die Referenz auf das aktuelle Objekt (in Java `this`) übergeben werden. Eine Abfrage des Packagenamens aus dem Modell ist mit dieser Lösung nicht notwendig.

Eine Unterscheidung zwischen Attributen und Assoziationen, wie sie für die Methoden `attributes` und `associationEnds` eigentlich notwendig wäre, ist ohne Zugriff auf Modelldaten zur Laufzeit nicht möglich. Die Methoden werden so implementiert, daß beide alle Attribute der Javaklasse zurückgeben. Das entspricht also sowohl Assoziationsenden als auch Attributen im Modell. Bei der Abfrage der Assoziationsenden wird außerdem der bei der Klasse `Ocl` registrierte Namensadapter verwendet, um Implementationsnamen in Modellnamen zurückzuverwandeln. Bei der Verwendung von `attributes` oder `associationEnds` sollte daher berücksichtigt werden, daß diese Methoden eine Obermenge der laut OCL-Spezifikation vorgesehenen Menge zurückgeben.

4.6.3 Zusammenfassung

Die Klassen `OclAnyImpl` und `OclType` sind in Abbildung 4.6 dargestellt.

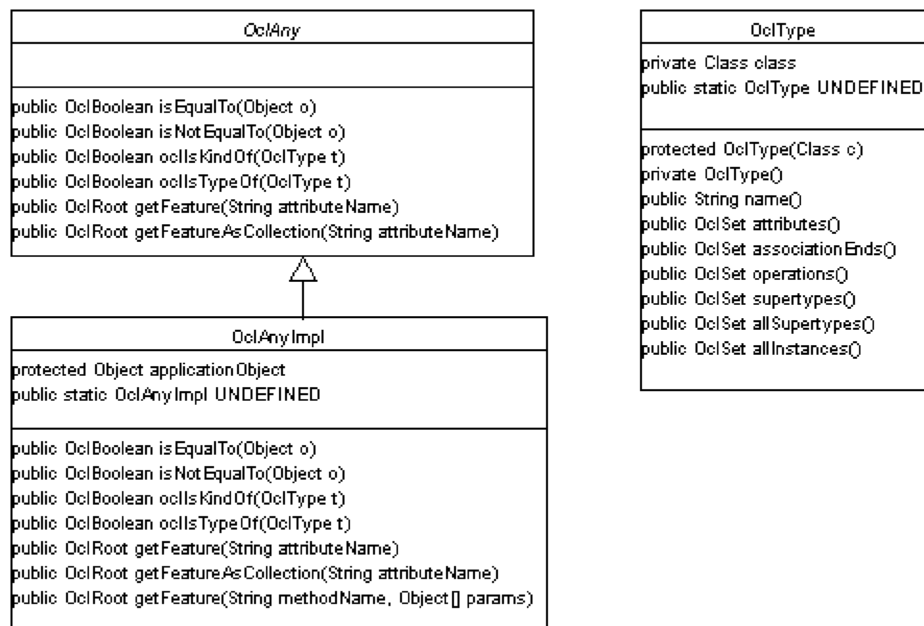

Abbildung 4.6: Klassendiagramm zu `OclAnyImpl` und `OclType`

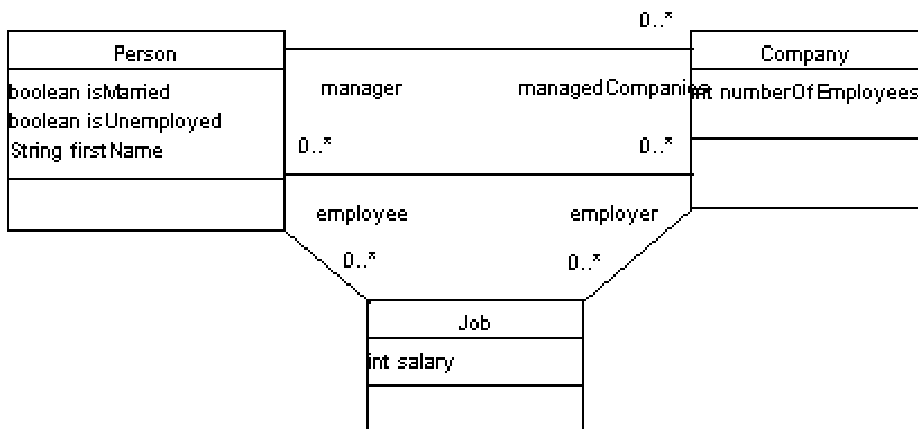

Abbildung 4.7: Klassendiagramm zu den OCL-Beispielen

4.7 Beispiele

Alle Beispiele in diesem Abschnitt beziehen sich auf das Klassendiagramm in Abbildung 4.7. Dieses UML-Modell ist ein Ausschnitt aus dem Beispielmodell in der OCL-Spezifikation. Da das Modell so, wie es dort verwendet wurde, in Java nicht generierbar ist, wurden hinsichtlich der Assoziationsklasse `Job` Änderungen vorgenommen. Ein Ziel hierbei war, die OCL-Ausdrücke, die sich auf das alte Modell bezogen, direkt in das veränderte übernehmen zu können. Daher wurde die verwendete Umsetzung der Assoziationsklasse gewählt.

Beispiel 1

```
context Company inv:
self.numberOfEmployees > 50
```

```
OclAny self=(OclAny)Ocl.getOclRepresentationFor(this);
OclBoolean constraintFulfilled=
  Ocl.toOclComparable(
    self.getFeature("numberOfEmployees")
  ).isGreaterThan(
    Ocl.toOclComparable(
      Ocl.getOclRepresentationFor(50)
    )
  );
```

Beispiel 2

```
context Person inv:
let income:Integer = self.job.salary->sum in
if self.isUnemployed then
  income < 100
else
  income >= 100
endif
```

```
OclAny self=(OclAny)Ocl.getOclRepresentationFor(this);
OclIterator iter;
OclAddable income=self.getFeature("job").
  getFeatureAsCollection("salary").sum();
OclBoolean constraintFulfilled=
  (Ocl.toOclBoolean(self.getFeature("isUnemployed")).isTrue()) ?
  (
    Ocl.toOclComparable(income).isLessThan(
      Ocl.toOclComparable(Ocl.getOclRepresentationFor(100))
    )
  ) :
  (
    Ocl.toOclComparable(income).isGreaterEqual(
      Ocl.toOclComparable(Ocl.getOclRepresentationFor(100))
    )
  );
```

Das Objekt, das von `self.getFeature("job")` zurückgegeben wird, ist in diesem Fall eine Instanz von `OclSet`. `getFeature("salary")` wird deswegen in seiner Implementierung für Kollektionen ausgeführt, d.h. es wird ein `OclBag` mit den Ergebnissen von `getFeature("salary")` für alle Elemente des `OclSet`s zurückgegeben.

Bei diesem Beispiel zeigt sich, daß Zwischenergebnisse wie in diesem Fall `income` gelegentlich mehrfach weiterverwendet werden. Die Methoden der Basisbibliothek müssen also tatsächlich rückwirkungsfrei implementiert werden. Das führt leider in einigen Fällen zu deutlich laufzeitintensiveren Realisierungen, wie man am Beispiel der Methode `OclSet including(OclRoot new)` der Klasse `OclSet` sehen kann. Die Instanz von `OclSet`, deren Methode aufgerufen wird, muß erst kopiert und dann um das neue Objekt `new` erweitert werden. Es ist nicht möglich, einfach die aktuelle Instanz um `new` zu vergrößern und zurückzugeben.

Beispiel 3

```
context Company inv:
self.employee->exists(
  p|p.managedCompanies->exists(
    c|c=self
  )
)
```

Diese Invariante ließe sich eleganter formulieren, allerdings soll hier die Umsetzung der verwendeten OCL-Sprachmittel in Java gezeigt werden.

```
final OclAny self=(OclAny)Ocl.getOclRepresentationFor(this);
OclCollection employee=self.getFeatureAsCollection("employee");
final OclIterator iterP=employee.getIterator();
OclBoolean contraintFulfilled=employee.exists(
  iterP,
  new OclBooleanEvaluatable() {
    public OclBoolean evaluate() {
      OclCollection managedComp=iterP.getValue().
        getFeatureAsCollection("managedCompanies");
      final OclIterator iterC=managedComp.getIterator();
      return managedComp.exists(
        iterC,
        new OclBooleanEvaluatable() {
          public OclBoolean evaluate() {
            return iterC.getValue().isEqualTo(self);
          }
        } // end inner class
      );
    }
  } // end inner class
);
```

Viele weitere Beispiele sind auf der beiliegenden CD in Form der Komponenten- und Integrationstests enthalten.

Kapitel 5

Test

In diesem Kapitel soll kurz beschrieben werden, wie die Klassenbibliothek getestet wurde. Zuerst wird auf den Komponententest eingegangen, der parallel zur Implementierung durchgeführt wurde. Dann wird der Integrationstest erläutert.

5.1 Komponententest

Für den Komponententest wurde das JUnit-Verfahren verwendet ([11]). Für diese Aufgabenklasse, nämlich die Entwicklung einer Klassenbibliothek, erwies sich diese Methode als sehr geeignet und hilfreich. Diese Erfahrung kann aber nicht notwendigerweise generalisiert werden, da der Umgang mit Datenbankzugriffen und Ausnahmebehandlungen relativ umständlich ist.

Dieses Verfahren soll nun kurz beschrieben werden. Die erstellten Testklassen sind auf der beiliegenden CD im Verzeichnis `classes/tudresden/ocl/test` enthalten. Voraussetzung für die Abarbeitung ist eine Javaumgebung wie das SUN-JDK 1.2. Das Verzeichnis `classes` sowie das Archiv `classes/junit.jar` müssen in die Umgebungsvariable `CLASSPATH` aufgenommen werden. Der Komponententest kann dann mit dem Kommando
`java test.ui.TestRunner tudresden.ocl.test.TestAll` gestartet werden.

5.1.1 JUnit

Das Prinzip, das JUnit ([11]) zugrunde liegt, ist, daß ein Programmierer parallel zur eigentlich zu schreibenden Klasse eine separate Testklasse erstellt. Für jede programmierte Methode wird dabei eine Testmethode geschrieben, die aus Aufrufen der zu testenden Methode und Vergleichen mit Sollwerten besteht.

Zu JUnit gehört ein einfaches Test-Framework, das allerdings laut Aussage der Autoren ohne großen Aufwand durch eine Eigenentwicklung ersetzt werden kann. Die zu schreibende Testklasse wird von der Klasse `TestCase` abgeleitet und erbt dabei eine Methode `assert(boolean b)`. Diese Methode wird dann in der Testklasse genutzt, um sie mit dem Vergleich aus Sollwert und Istwert als Parameter aufzurufen. Jede Testklasse implementiert außerdem eine Klassenmethode namens `suite`, die ein Objekt vom Typ `Test` zurückgeben muß. In dieser Methode wird gewöhnlich eine neue `TestSuite` erzeugt und dieser mehrere Instanzen der Testklasse hinzugefügt. Diese Instanzen arbeiten beim Aufruf

ihrer `run`-Methode jeweils eine Testmethode ab.

Sowohl `TestCase` als auch `TestSuite` implementieren die Schnittstelle `Test`. Eine `TestSuite` ist eine Ansammlung von `Tests`, die nacheinander ausgeführt werden. Somit entspricht diese Architektur dem Kompositum-Muster ([6]). Es ist daher auch möglich, zu einer `TestSuite` wiederum weitere Instanzen von `TestSuite` hinzuzufügen, und so aus den verschiedenen Klassentests einen Komponententest für ein ganzes Projekt zusammenzubauen. Für die OCL-Klassenbibliothek existiert mit der Klasse `tudresden.ocl.test.TestAll` ein solcher Projekttest.

Ebenfalls zum Framework gehört eine grafische Oberfläche, aus der heraus Testklassen aufgerufen werden können. Ein Fortschrittsbalken informiert den Benutzer darüber, wieviele der Testmethoden bereits abgearbeitet wurden. Im Fall eines Fehlers wird dem Benutzer der *Stack Trace* zum Fehlerzeitpunkt angezeigt. Ein Fehler kann dabei entweder aus einem Aufruf der `assert`-Methode mit `false` als Parameter oder aus einer unbehandelten Ausnahme (`Exception`) in der getesteten Klasse bestehen.

Als Kritikpunkte zu diesem Verfahren ist außer den schon genannten Problemen mit Ausnahmebehandlung und der Überprüfung von Nebenwirkungen, z.B. auf Datenbanken, die fehlende Trennung zwischen Methoden- und Klassentest zu nennen, die in der objektorientierten Programmierung eigentlich möglich ist ([13]). Auch die methodisch vorgesehene zeitliche Nähe zwischen der Programmierung der Methode und dem Schreiben des dazugehörigen Tests wirkte sich negativ aus, da so Fehleinschätzungen bestimmter Verhaltensweisen des Programms nicht noch einmal überdacht wurden.

5.2 Integrationstest

Um einen möglichst anwendungsnahen und alle OCL-Konstrukte abdeckenden Integrationstest der Klassenbibliothek zu ermöglichen, wurde das Beispiel „royals and loyals“ aus [3] implementiert. Alle dazu angegebenen OCL-Invarianten wurden in entsprechende Java-Ausdrücke überführt. Lediglich triviale Ausdrücke, deren korrekte Abarbeitung aufgrund vorheriger Tests außer Zweifel stand, wurden ausgelassen, sowie solche, die das Modell nicht sinnvoll ergänzt hätten (z.B. `context Customer inv: name='Edward'`). Außerdem wurden OCL-Ausdrücke formuliert und umgesetzt, die aus dem UML-Diagramm abgeleitet werden können, wie z.B.:

```
context LoyaltyProgram
inv: customer->forAll(
  program->includes(self)
)
```

Diese überprüfen die konsistente Implementierung von Assoziationen durch Java-Referenzen. Die Tatsache, daß erstaunlich viele der als Beispiele aus [3] entnommenen OCL-Ausdrücke kleine Ungenauigkeiten vor allem in ihrem Bezug auf das Anwendungsmodell enthielten, belegt die Notwendigkeit von Werkzeugunterstützung für OCL.

Um das Beispiel in die verwendete Softwareentwicklungsumgebung Argo/UML eingeben zu können, mußten Assoziationsklassen und Enumerationen in dem Modell ersetzt werden. Dies stellt eine Überführung von einem

Analyse- in ein Entwurfsmodell dar, wie sie ansonsten zur Generierung von Java-Quelltexten manuell oder automatisiert notwendig gewesen wäre. Das resultierende UML-Diagramm ist in Abbildung 5.1 dargestellt. Die im generierten Programm überprüften OCL-Ausdrücke sind in Tabelle 5.1 aufgelistet. Aus Gründen der Übersichtlichkeit fehlen in dieser Tabelle diejenigen OCL-Bedingungen, die aus dem Diagramm abgeleitet wurden.

Um Instanzen dieses UML-Modells erzeugen und bearbeiten zu können, wurde eine einfache grafische Oberfläche (siehe Abbildung 5.2) erstellt. Über die im oberen Fensterbereich gelegenen Schaltflächen können neue Objekte zur Beispielpopulation hinzugefügt werden, die dann in der Liste in der Fenstermitte dargestellt werden. Über das Textfeld am unteren Rand können einfache Manipulationsanweisungen eingegeben werden, die dann bei Drücken der `ENTER`-Taste oder der Schaltfläche „`EXECUTE`" für das gerade markierte Objekt in der Liste ausgeführt werden. Einige Beispiele für mögliche Anweisungen sind in Tabelle 5.2 aufgeführt. Über die Schaltfläche „`ASSERT`" kann die Überprüfung der OCL-Bedingungen für alle Anwendungsobjekte gestartet werden. Sämtliche Ausgaben erfolgen auf die Konsole. Um die aufwendig eingegebene Objektpopulation über einen Testlauf hinaus nutzen zu können, kann sie in einer Datei gespeichert und wieder geladen werden.

Die Quelltexte und ausführbaren `class`-Dateien zu diesem Beispiel sind auf der beiliegenden CD im Verzeichnis `classes/royloy` enthalten. Um die grafische Oberfläche zu starten, muß eine Javaumgebung wie z.B. das SUN-JDK 1.2 installiert sein und die Umgebungsvariable `CLASSPATH` auf das Verzeichnis `classes` auf der CD gesetzt werden. Der Aufruf erfolgt dann mit dem Kommando `java royloy.RLObject` aus der Konsole heraus. Als Parameter kann der Pfad zu einer Objektpopulationsdatei übergeben werden, wie sie mit `rlpop` im Verzeichnis `classes/royloy` auf der CD zu finden ist.

Abbildung 5.1: Das Beispielmodell „royals and loyals“

```
context Burning
inv: self.oclType=Burning
inv: self.oclIsKindOf(Transaction)=true
inv: self.oclIsTypeOf(Transaction)=false
inv: Customer.name=´Customer´ -- in beliebigem Kontext möglich
inv: Transaction.attributes->includesAll(Set{´points´, ´date´})
inv: Transaction.associationEnds->includesAll(
  Set{´serviceLevel´, ´loyaltyAccount´, ´card´}
)
inv: Burning.supertypes=Set{Transaction}
```

```
context Customer
inv: title=(if isMale=true then ´Mr´ else ´Ms´ endif)
```

```
context LoyaltyProgram
inv: self.customer->forAll(
  c1, c2 | c1<>c2 implies c1.name<>c2.name
)
inv: self.customer->isUnique(name)
```

```
context Membership
inv: customer.cards.membership->includes(self)
inv: program.serviceLevel->includes(actualLevel)
inv: loyaltyAccount.points>=0 or loyaltyAccount->isEmpty
```

```
context ProgramPartner
inv: self.deliveredServices.transactions->iterate(
      t:Transaction;
      result:Integer = 0 |
      if t.oclType=Burning then result+points else result endif
    )
    <=
    self.deliveredServices.transactions->iterate(
      t:Transaction;
      result:Integer = 0 |
      if t.oclType=Earning then result+points else result endif
    )
```

```
context Service
inv: self.pointsEarned>0 implies not self.pointsBurned=0
```

```
context ServiceLevel
inv: loyaltyProgram.partners->includes(service.programPartner)
```

```
context Transaction
inv: self.program()=customerCard.membership.program
```

```
context Transaction::program()
post: result=self.card.membership.program
```

Tabelle 5.1: Umgesetzte OCL-Ausdrücke

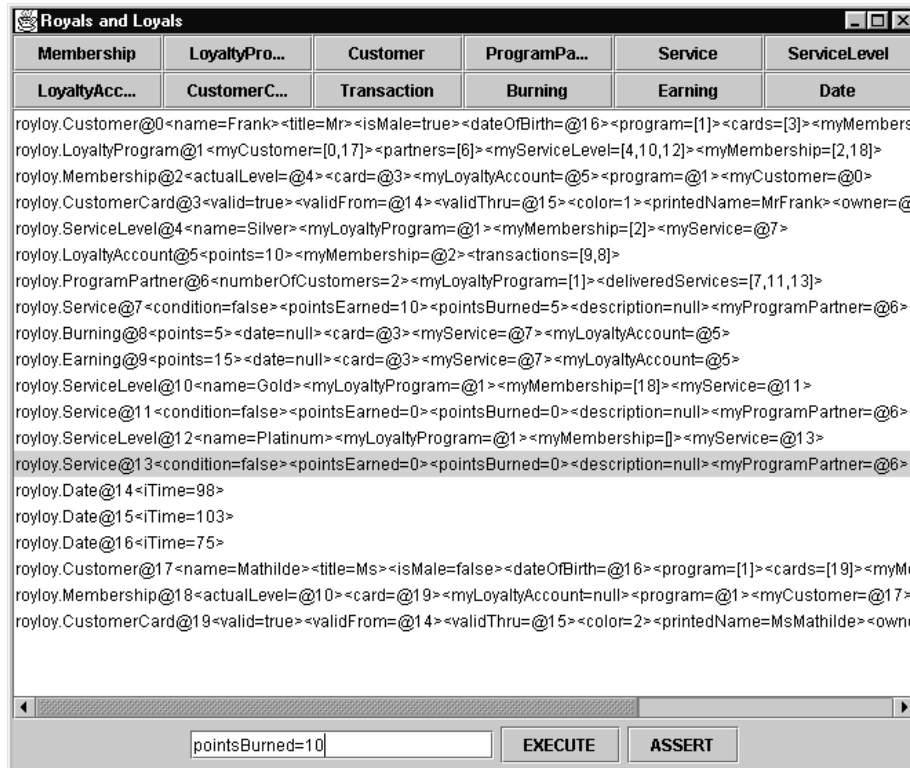

Abbildung 5.2: Die Benutzungsoberfläche zu „royals and loyals“

delete	markiertes Objekt aus Populationsliste austragen
points=10	Attribut `points` des markierten Objekts auf den Wert 10 setzen
myMembership=@10	Attribut `myMembership` des markierten Objekts auf eine Referenz auf das Objekt mit der Identifikationsnummer 10 setzen
cards.add(@7:Object) cards.remove(@7:Object)	dem Vector `cards` des markierten Objekts eine Referenz auf das Objekt mit der Identifikationsnummer 7 hinzufügen; der Typ des Parameters der aufzurufenden Methode muß mit angegeben werden diese Referenz wieder entfernen

Tabelle 5.2: Beispiele für Manipulationsanweisungen

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

In dieser Arbeit wurde eine Java-Klassenbibliothek erstellt, die die vordefinierten Typen der OCL in der Version 1.3 implementiert. Es wurde darauf geachtet, daß Aufrufe der Klassenbibliothek mit einem Generator erstellt werden können, der seine Informationen soweit wie möglich aus dem OCL-Ausdruck selbst und nicht aus dem dazugehörigen UML-Modell bezieht. Die Klassenbibliothek wurde mit Hilfe des Werkzeugs `javadoc` dokumentiert und mit JUnit komponentengetestet. Außerdem wurde ein Beispielmodell erstellt, auf dessen Instanzen verschiedene OCL-Bedingungen überprüft wurden.

An solchen Stellen, an denen sich von verschiedenen Softwareentwicklungsumgebungen generierter Quelltext typischerweise unterscheidet, sind Schnittstellen zur Adaption vorgesehen. Diese betreffen die Umsetzung von Assoziationen und Modelldatentypen in Anwendungsrepräsentationen, die Veränderungen an Namen von Assoziationsenden im Vergleich zur OCL-Notation sowie das Abfragen von Zustandsinformationen.

6.2 Kritik

Leider ist es mit dieser Lösung nicht möglich, sämtliche OCL-Konstrukte in Java umzusetzen. Die Einschränkungen wurden in Abschnitt 2.1 dargestellt.

Generell ist die Anpaßbarkeit der Klassenbibliotek über Adapter und Fabriken auf die Aspekte beschränkt, in denen sich Codegeneratoren für gewöhnlich unterscheiden. Darüber hinaus gehende Adaptionen würden Eingriffe in den Quelltext der Klassenbibliothek erfordern.

6.3 Ausblick

Mit Hilfe dieser Bibliothek kann ein OCL-Compiler erstellt werden, der OCL-Ausdrücke in Java-Quelltext umwandelt. Dieser Quelltext besteht im wesentlichen aus Aufrufen der Klassenbibliothek. Obwohl beim Entwurf der Bibliothek auf die Verwendung in einem OCL-Interpreter keine besondere Rücksicht genommen worden ist, ist auch ein solcher Einsatz denkbar.

Literaturverzeichnis

[1] Object Constraint Language, Kapitel 7 in [2]

[2] Unified Modeling Language 1.3. http://uml.shl.com/artifacts.htm

[3] J. Warmer und A. Kleppe: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, 1999.

[4] L. Mandel und M. C. Cengarle: On the expressive power of the object constraint language OCL. Forschungsinstitut für angewandte Softwaretechnologie (FAST e.V.), http://www.fast.de, 1999

[5] Argo/UML. http://www.ics.uci.edu/pub/arch/uml

[6] E. Gamma u.a.: Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, 1996

[7] J. Gosling u.a.: Java Language Specification. Addison-Wesley, 1996

[8] Java Language Specification, Update 1.1.
http://java.sun.com/docs/books/jls

[9] JDK 1.2 API Specification.
http://www.javasoft.com/products/jdk/1.2/docs/api

[10] UML Revision Task Force Homepage. http://uml.shl.com

[11] JUnit. http://members.pingnet.ch/gamma/junit.htm

[12] H. Hussmann: Lehrmaterialien zur Vorlesung Formale Spezifikation von Softwaresystemen

[13] H. Hussmann: Lehrmaterialien zur Vorlesung Softwaretechnologie 2