Diplomarbeit

Evaluation der Praxistauglichkeit von OCL-Spezifikationen

bearbeitet von

Steffen Zschaler

geboren am 07. Dezember 1976 in Dresden

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Betreuer: Dr.-Ing. Birgit Demuth
Prof. Dr. Johannes Siedersleben (sd&m Research)
Oliver Juwig (sd&m Research)
Hochschullehrer: Prof. Dr. rer. nat. habil. Heinrich Hußmann

Eingereicht am 01. August 2002

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Object Constraint Language (OCL) ist eine formale Sprache zur Formulierung von Anfragen und Bedingungen für objektorientierte Systeme. Sie ist seit der ersten offiziell von der OMG anerkannten Version der Unified Modeling Language (UML) Teil dieses Standards. Damit wird versucht, formale Methoden stärker in die Praxis der Softwareentwicklung einzubringen und Hemmschwellen abzubauen.

Die Notwendigkeit eines sorgfältigen Entwurfs, unter Umständen sogar unter Verwendung formaler Methoden, ist im akademischen Bereich sowie in der Praxis allgemein anerkannt. Allerdings werden formale Methoden in der Praxis kaum angewandt. Gründe dafür sind möglicherweise im komplizierten mathematischen Hintergrund dieser Formalismen zu suchen. Deshalb wurde mit der OCL eine Sprache geschaffen, deren Syntax sich an die objektorientierter Sprachen anlehnt und dahinter einen Formalismus (prädikatenlogikbasiert) „versteckt“.

Bisher gibt es aber kaum Aufwand-Nutzen-Abwägungen, die helfen könnten, fundierte Entscheidungen zur Anwendung von OCL zu treffen. Hier soll diese Arbeit einen ersten Schritt darstellen. Außerdem soll auch grundsätzlich untersucht werden, inwiefern die relativ junge Sprache OCL geeignet ist, komplexe Systeme zu spezifizieren.

Dazu wurde ein Teil von JEFF, eines Frameworks von sd&m, mit Hilfe von OCL spezifiziert. In Zusammenarbeit mit sd&m wurden anschließend Kriterien definiert, anhand derer die entstandene Spezifikation evaluiert wur-

de. Bei der Erstellung der Spezifikation wurden einige Schwächen der OCL gefunden, für welche Lösungsansätze entwickelt wurden.

Der Logik dieses Vorgehens folgt die Gliederung der vorliegenden Arbeit. Zunächst wird eine kurze Einführung in die OCL (Kapitel 2) und in das sd&m-Framework JEFF (Kapitel 3) gegeben. Kapitel 4 stellt das Vorgehen, die Erfolge und die Probleme der erstellten Spezifikation dar. Außerdem werden die erarbeiteten Lösungsansätze erläutert. Die gefundenen Kriterien sowie die Evaluation der Spezifikation gegen diese Kriterien findet sich in Kapitel 5. Schließlich faßt Kapitel 6 die Ergebnisse der Arbeit zusammen und versucht einen Ausblick auf mögliche weitere Entwicklungen.

Kapitel 2

Die Object Constraint Language

Die Object Constraint Language (OCL) ist eine prädikatenlogikbasierte Sprache, die die grafischen Ausdrucksmittel der UML (Unified Modeling Language) durch formale textuelle Mittel ergänzt. Bestimmte komplexere Bedingungen lassen sich mit den grafischen Mitteln der UML nicht oder nur sehr schlecht darstellen. Die OCL wurde als Teil der UML eingeführt, um auch solche Bedingungen spezifizieren zu können.

Basierend auf Syntropy's „business modeling language“ ([CD94]), wurde 1995 die OCL in einem IBM-Projekt geboren. Im Rahmen der Bemühungen der Object Management Group (OMG) um eine standardisierte objektorientierte Modellierungssprache wurde die OCL in die UML ([UML01]) integriert. [WK99] gibt eine gute Einführung in die OCL.

Die Entwicklung der OCL wurde durch die folgenden Anforderungen gesteuert ([WK99], Seite 8):

1. OCL muß eine Sprache sein, die zusätzliche Aussagen über die Modelle und Artefakte der objektorientierten Entwicklung erlaubt.

2. OCL muß eine präzise und eindeutige Sprache sein, die von Software-Entwicklern und ihren Kunden leicht erlernt, gelesen und geschrieben werden kann. Das heißt insbesondere, es darf keine Sprache sein, die nur für Mathematiker und Informatiker verständlich ist.

3. OCL muß deklarativ sein, d.h. OCL-Ausdrücke haben keine Seiteneffekte. OCL definiert nur Bedingungen für zulässige Systemzustände, sie trifft keine Aussage bzgl. der Behandlung illegaler Zustände.

4. OCL muß getypt sein. Dadurch können OCL-Ausdrücke einer ersten (Typ-)Plausibilitätsprüfung unterzogen werden.

Im Verlauf der weiteren Entwicklung ergaben sich zusätzliche Anforderungen, insbesondere die der „Ausführbarkeit“. Das heißt, daß OCL-Ausdrücke über einem objektorientierten Modell zur Laufzeit der dem Modell entsprechenden Anwendung auswertbar sein sollten.

Ausgehend von diesen Anforderungen entstand eine Sprache, die auf Prädikatenlogik basiert, jedoch eine an objektorientierte Programmiersprachen angelehnte Syntax hat. Die Quantoren $\forall$ und $\exists$ sowie die verschiedenen Relationen und Operationen auf Mengen und Elementen ($\in$, $\subset$, $\cap$, $\cup$ ...) werden durch Operationen auf Kollektionen ausgedrückt (`->forAll()`, `->exists()`, `->includes()`, `->includesAll()` ...). Dies balanciert die widersprüchlichen Anforderungen unter 2. aus.

Aus der Anforderung der „Ausführbarkeit“ ergibt sich eine Beschränkung der Quantoren und Iteratoren auf endliche Mengen. Das wird erreicht, indem das Erzeugen unendlicher Mengen syntaktisch unterbunden wird. So ist die Operation `::allInstances()`, die die Menge aller Instanzen eines Typs liefert, nur für solche Typen definiert, die zur Laufzeit eine endliche Instanzenmenge haben (also z.B. nicht für `Integer`).

Der letzte Unterschied zur klassischen Prädikatenlogik ist die Möglichkeit undefinierter Ausdrücke. Dazu wird ein spezieller Wert (`OclUndefined`) eingeführt, der Teil der Instanzenmenge jedes Typs ist. Dieser Wert entspricht dem `null`-Wert aus Java, kann aber auch Fehlerbedingungen bei der Ausführung von Operationen des Modells markieren. Die meisten OCL-Operationen sind strikt, d.h. wenn einer der Parameter undefiniert ist, so ist auch das Ergebnis undefiniert. Die Booleschen Operatoren und Relationen bilden dabei eine Ausnahme, für sie gelten besondere Regeln, die im Standard ausführlich dargestellt sind.

OCL wird z.B. in den folgenden Projekten/Produkten eingesetzt:

- Die in [DHL01] und [Löc01] vorgestellten Methoden zur Anwendung der OCL zur Formulierung von Konsistenzbedingungen über Datenbanken wurden bei der EDF (Electricité de France) auf die Kundenstammdatenbank angewendet.

- Die Bold!-Architektur von BoldSoft ([Bol]) verwendet OCL als Anfragesprache, um darzustellende Objektmengen zu beschreiben. Aus einem mit OCL angereicherten Modell wird Quelltext generiert, der die entsprechenden Mengen aufsammelt.

Diese Arbeit beruht auf dem OCL 2.0 Vorschlag von BoldSoft, Rational und IONA ([W+01]). Teilweise werden in der vorliegenden Arbeit Erweiterungen vorgeschlagen, die nicht mehr in den OCL 2.0 Vorschlag aufgenommen wurden.

Kapitel 3

Das Komponenten-Framework JEFF

sd&m stellte das Java Enterprise Foundation Framework (JEFF, [MS01], [MU01]) für die Spezifikation zur Verfügung.

3.1 Überblick über das Framework

JEFF ist ein von sd&m entwickeltes Java-Framework zur Entwicklung von modularen, interaktiven Geschäftsanwendungen. Es besteht aus voneinander unabhängigen Komponenten, die die verschiedenen Teile einer Geschäftsanwendung realisieren.

Abbildung 3.1 gibt einen Überblick über die einzelnen Komponenten. Das Framework unterteilt sich in einen Server- und einen Client-Teil. Der Client-Teil umfaßt die Rahmenanwendung (das **Portal**), welche einen konfigurierbaren Rahmen für die Darstellung der Funktionen bietet. Diese Darstellung erfolgt durch **View**s, die — teilweise modellbasiert — die Umsetzung der GUI übernehmen. Die client-seitigen **View**s können auf der Server-Seite durch einen **ViewServer** unterstützt werden.

Die Kommunikation über das – möglicherweise vorhandene – Netzwerk wird durch den **CommunicationManager** erleichtert. Proxies (die vom

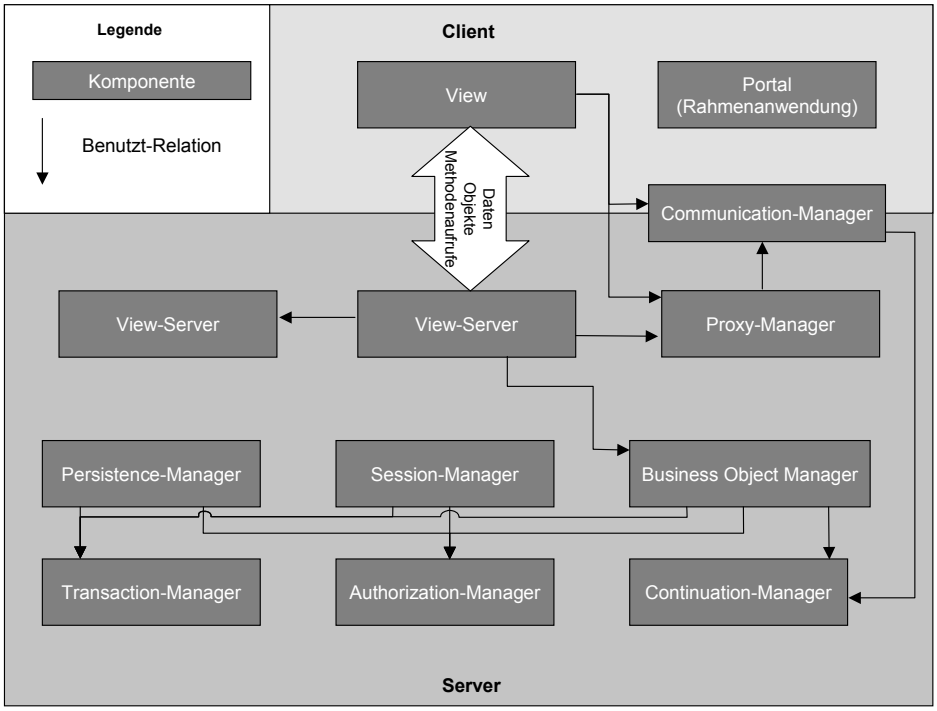

Abbildung 3.1: Komponenten des JEFF-Frameworks. ([MS01], [MU01])

Proxy-Manager verwaltet werden) sind eine Optimierung, um die Netzwerklast zu senken.

Der **ValidationManager** unterstützt die Überprüfung der strukturellen Korrektheit und semantischen Plausibilität der vom Anwender eingegebenen Daten.

Die Basis des Frameworks bilden zwei Schichten von Komponenten, die grundlegende Dienste zur Verfügung stellen. Die erste Schicht bietet Dienste auf einer höheren Abstraktionsebene, nämlich:

PersistenceManager Persistenz von Objekten und Objektstrukuren,

SessionManager Verwaltung von **Sessions**, die Benutzern zugeordnet werden können,

BusinessObjectManager Registratur von **Business Objects** (BOs) und anderen Diensten.

Die zweite Schicht bietet grundlegende Dienste:

TransactionManager Verwaltung von Transaktionen,

AuthorizationManager Verwaltung von Berechtigungen und Benutzern,

ContinuationManager Optimierung, die das Multiplexen eines physischen Threads in mehrere logische Threads erlaubt.

Die einzelnen Komponenten des Frameworks sind vollständig voneinander abgekoppelt. Sie bieten ihre Funktionen ausschließlich über wohldefinierte Schnittstellen (Java-Interfaces) an und verwenden die Funktionen anderer Komponenten ausschließlich über *Stützschnittstellen*, die sie selbst definieren. Diese Konzepte entsprechen den *offered* und *used interfaces*, wie sie in [CD01] diskutiert werden. Dadurch definiert die Komponente selbst, welche Operationen sie von anderen Komponenten benötigt sowie die Bedingungen, unter denen diese Operationen aufgerufen werden.

Da die Komponenten von außen nur über ihre Schnittstellen zu erreichen sind, kann die Implementation jederzeit ausgetauscht werden. Durch die Verwendung von *Stützschnittstellen* für den Zugriff auf andere Komponenten wird Unabhängigkeit von der Implementation externer Funktionen erreicht.

Das gesamte Framework ist ausschließlich natürlichsprachlich dokumentiert. Im Quelltext selbst werden zwar Assertions verwendet, diese sind jedoch in den Dokumenten nicht aufgeführt. Neben der Javadoc existieren für jede Komponente eine **Design Rationale** (Architekturbeschreibung), ein **Nutzungskonzept** (eine Art kurzes Tutorial) sowie ein **Testkonzept** (Beschreibung der Testfälle der Komponente) als Word-Dokumente. Für manche Komponenten gibt es zusätzlich eine kleine Beispielanwendung, die die wesentlichen Punkte präsentiert.

3.2 Der Persistenz-Manager von JEFF

Die praktische Spezifikationsarbeit konzentrierte sich auf die Persistenz-Manager-Komponente ([OJ01]), welche deshalb genauer vorgestellt wird.

Persistente Daten können in JEFF durch die Komponente **Persistence-Manager** verwaltet werden. Diese Komponente verhält sich analog zu einer objektorientierten Datenbank und kapselt die tatsächlich verwendete Persistenz-Technologie. Eine – mitgelieferte – Möglichkeit ist dabei die Anbindung an eine relationale Datenbank mittels JDBC und unter Verwendung einer objekt-relationalen Abbildungsvorschrift.

3.2.1 Pakete

Die Klassen, die den Persistenz-Manager implementieren, sind in Paketen mit dem Präfix `com.sdm.jeff.persistence` zusammengefaßt. Die Komponente gliedert sich (wie alle JEFF-Komponenten) in eine Komponentenschnittstelle (in `com.sdm.jeff.persistence`) und eine Komponentenimplementation (in `com.sdm.jeff.persistence.implementation`, `com.sdm.jeff.persistence.model` und `com.sdm.jeff.persistence.datasource`). Im Bereich der Komponentenimplementation werden auch die verwendeten *Stützschnittstellen* definiert.

Weiterhin finden sich Klassen zur Implementation einer objekt-relationalen Abbildungsvorschrift (`com.sdm.jeff.persistence.mapping`) sowie zur Formulierung und Ausführung von Abfragen (`com.sdm.jeff.persistence.query` und `com.sdm.jeff.persistence.query.implementation`). Die-

se sind in ihrer Implementation teilweise recht speziell auf die Standardimplementation des Persistenz-Managers zugeschnitten.

Zusätzlich gibt es eine Beispielanwendung (`com.sdm.jeff.persistence.sample`) sowie Testklassen (`com.sdm.jeff.persistence.test`).

Bei der Spezifikation lag der Fokus auf der Schnittstelle der Komponente (`com.sdm.jeff.persistence`) sowie auf den hauptsächlichen Implementationsklassen (`com.sdm.jeff.persistence.implementation`).

3.2.2 Komponentenschnittstelle

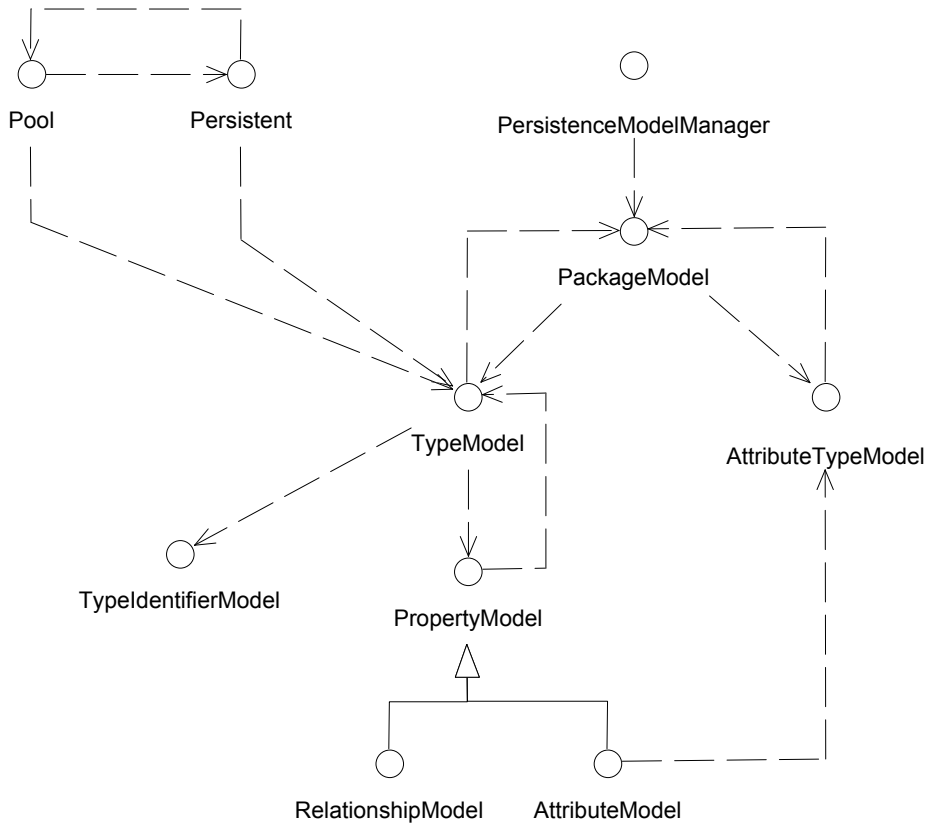

Abbildung 3.2: Die wesentlichen Interfaces der Komponentenschnittstelle des JEFF-Persistenz-Managers. Bei den Abhängigkeiten handelt es sich um ≪use≫-Dependencies, die von der Implementation als Assoziationen umgesetzt werden.

Ein UML-Klassendiagramm für die Komponentenschnittstelle des Persistenz-Managers zeigt Abbildung 3.2.

Persistente Objekte sind Instanzen von Klassen, die das Interface `Persistent` implementieren. Dabei handelt es sich um Klassen der Anwendung und nicht um Frameworkklassen. `Persistent` stellt in diesem Sinn eine *Stützschnittstelle* dar. Im Gegensatz zu anderen *Stützschnittstellen* kann die Implementierung jedoch direkt entsprechende Operationen in `Pool` nutzen.

Ein `Pool` verwaltet mehrere persistente Objekte. Nur solche Objekte, die mit einem Pool verbunden sind, sind auch persistent. Der Pool stellt damit die Schnittstelle zur Datenbank dar.

Die Struktur der zu verwaltenden Daten wird dem Persistenz-Manager durch entsprechende Modelle bekanntgemacht. Ein Modell wird vom `PersistenceModelManager` verwaltet und in `PackageModel`s, `TypeModel`s und `PropertyModel`s strukturiert. `PackageModel`s dienen der weiteren Strukturierung des Modells. Ein `PackageModel` enthält `TypeModel`s und `AttributeTypeModel`s. Ein `TypeModel` beschreibt eine einzelne persistente Klasse durch Auflistung ihrer Eigenschaften (`PropertyModel`s) und der zu verwendenden Schlüssel bzw. Schlüsselkandidaten (`TypeIdentifierModel`).

Jede Eigenschaft einer einzelnen persistenten Klasse wird durch ein `PropertyModel` beschrieben, speziell durch ein `RelationshipModel` für eine Beziehung zu anderen persistenten Klassen und ein `AttributeModel` zur Beschreibung eines Attributs. `AttributeModel` verwendet zusätzlich ein `AttributeTypeModel`, um den Typ des Attributs zu kapseln.

Auch die Schnittstellen für Modelle stellen in gewisser Weise *Stützschnittstellen* des Persistenz-Managers dar. Die Anwendung stellt diese Modelle entweder durch Nutzung der Standardimplementation in `com.sdm.jeff.persistence.model` oder durch direkte Implementation der Schnittstellen zur Verfügung.

3.2.3 Komponentenimplementation

Abbildung 3.3 zeigt die wichtigsten Klassen, die an der Implementation des Persistenz-Managers beteiligt sind. Die Implementation des Datenmodells entspricht direkt der Schnittstellendefinition und wird deshalb hier

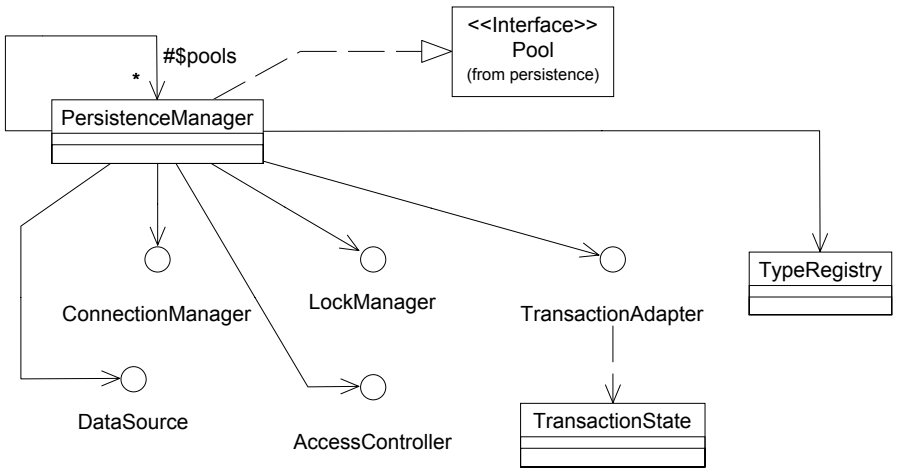

Abbildung 3.3: Die wesentlichen Klassen der Implementation des JEFF-Persistenz-Managers.

nicht nochmals erläutert. Die entsprechenden Klassen finden sich im Paket `com.sdm.jeff.persistence.model` und erscheinen der besseren Übersichtlichkeit halber nicht im UML-Klassendiagramm.

Die Implementation des Persistenz-Managers besteht aus weiteren Klassen, deren Darstellung im Detail den Rahmen dieser Arbeit sprengen würde. Die Beschreibung konzentriert sich im folgenden vor allem auf die Darstellung der Anbindung anderer Komponenten sowie auf die direkte Untersetzung der Komponentenschnittstelle.

Die Funktionen von `Pool` werden von `PersistenceManager` unter Verwendung mehrerer anderer Klassen umgesetzt. Die Anbindung des Datenmodells erfolgt über die `TypeRegistry`, welche auch die Anbindung der Klassen in `com.sdm.jeff.persistence.model` übernimmt.

Verschiedene andere Funktionen werden über *Stützschnittstellen* genutzt. Für diese *Stützschnittstellen* gibt es stets Standardimplementationen, die entweder eine andere JEFF-Komponente anbinden, oder die benötigten Funktionen selbst implementieren. So gibt es `ConnectionManager` und `DataSource` (in `com.sdm.jeff.persistence.datasource`), die die Anbindung einer Datenbank übernehmen. Benutzer- und Rechteverwaltung wer-

den mit Hilfe von `AccessController` genutzt, eine Sperrenverwaltung steht über `LockManager` zur Verfügung. Zur Zeit gibt es nur eine Implementation der Sperrenverwaltung, die ausschließlich Sperren im selben Prozeß zuläßt.

Ein Transaktionsmanager kann über die Schnittstelle `TransactionAdapter` angebunden werden. Der interne Zustand aller innerhalb einer Transaktion geänderten persistenten Objekte wird mit Hilfe eines `TransactionState`-Objektes verwaltet.

Das Interface `Persistent` aus der Komponentenschnittstelle ist eigentlich eine Schnittstelle, die von Nutzern des Persistenz-Managers implementiert werden muß. Daher gibt es auch nur eine Implementation der grundlegenden Funktionen, die als Baustein für die eigene Implementation genutzt werden kann (`AbstractPersistent`).

Kapitel 4

Die Spezifikation

In diesem Kapitel werden das bei der Spezifikation des Persistenz-Managers gewählte Vorgehen, aufgetretene Probleme sowie gefundene Lösungsansätze dargestellt.

4.1 Vorgehen

Die Spezifikation konzentrierte sich auf die Komponente Persistenz-Manager des JEFF. Als Grundlage dienten verschiedene englisch- und deutschsprachige Dokumente, wie sie von sd&m bisher zur Dokumentation der Komponenten des JEFF verwendet werden, sowie der Quelltext der Komponente.

Die in der Spezifikation verwendete OCL basiert auf dem OCL 2.0 Vorschlag von Boldsoft, Rational und IONA ([W+01]). Im Verlauf der Arbeit an der Spezifikation fanden sich allerdings einige Probleme, deren Lösungsansätze nicht mehr in den Vorschlag Eingang fanden. Dies betrifft insbesondere die Vorschläge zur Spezifikation von Exceptions (s. Abschnitt 4.4.1).

Es wurden sowohl die Klassen der Komponentenschnittstelle als auch die Klassen der Komponentenimplementation spezifiziert. Die Spezifikation der Komponentenschnittstelle ist so formuliert, daß eine beliebige Implementation verwendet werden könnte, d.h. es wurden keine Referenzen auf implementationseigene Klassen/Operationen aufgenommen. Bei der Spezifikation der Implementation handelt es sich hingegen beinahe um eine Übersetzung der Implementation in OCL. Diese „Spezifikation“ (oder vielmehr Beschrei-

bung in OCL) wurde durchgeführt, um zu untersuchen, inwiefern sich dieser Stil für die Überprüfung von Implementationsbeziehungen eignet.

Viele Eigenschaften des Persistenz-Managers ließen sich gut in OCL spezifizieren. Dennoch ergaben sich einige Probleme. Bei der Erarbeitung von Lösungsansätzen entstanden verschiedene Versionen der Spezifikation, welche sich auf der beigefügten CD-ROM befinden (s. Anhang B). Die Syntax und Semantik von `OclMessage` hat sich von Revision 1.4 zu Revision 1.5 des OCL 2.0-Vorschlags stark verändert. Diese Veränderung wird zwar im Text dieser Arbeit, jedoch nicht in der Spezifikation berücksichtigt. Die Spezifikation auf der CD-ROM verwendet weiterhin die Fassung der Revision 1.4 ([W+02a]).

Die Ausführbarkeit von OCL-Ausdrücken ist stets ein Grundziel bei der Entwicklung von OCL gewesen (s. z.B. [CKM+02], Seite 118). Daher ist eine OCL-Spezifikation bereits sehr nah an der Programmiersprache. Bei der Abbildung Java – UML/OCL gibt es jedoch einige Punkte zu beachten bzw. Entscheidungen zu treffen:

- OCL definiert seine eigenen Kollektionstypen (`CollectionType` und abgeleitete Typen ([W+01])), um Sprachunabhängigkeit zu gewährleisten. In Java gibt es ebenfalls Kollektionsklassen (im Paket `java.util`) sowie das Sprachkonstrukt des Arrays. Der OCL-Standard definiert nicht, in welcher Beziehung solche sprachspezifischen Kollektionen zu den OCL-Kollektionen stehen.

  Für diese Arbeit wurde entschieden, Java-Kollektionen und OCL-Kollektionen gleichzusetzen. Insbesondere gilt das Mapping aus Tabelle 4.1.

Java-Kollektionstyp	*OCL-Kollektionstyp*
`java.util.Collection`	`CollectionType`
`java.util.List`	`SequenceType`
Java array	`SequenceType`
`java.util.Set`	`SetType`

Tabelle 4.1: Abbildung von Java-Kollektionen auf OCL-Kollektionen

Mit dieser Entscheidung sind natürlich auch Probleme verbunden. Die beiden hauptsächlichen Punkte, die hier genannt werden sollen, sind:

1. Java-Kollektionen sind ungetypt, während OCL-Kollektionen stets getypt sind. Ein formales, automatisierbares Mapping müßte daher alle Java-Kollektionen in OCL-Kollektionen mit dem `elementType OclAny` übersetzen. Dieses Vorgehen versagt jedoch für geschachtelte Java-Kollektionen, da keine Instanz von `CollectionType` ein Subtyp von `OclAny` ist.

   Die Typfreiheit von Java-Kollektionen ist aber eher sprachbedingt, tatsächlich wird in den meisten Fällen mit homogenen Kollektionen[1] gearbeitet. Daher wurde eine Übersetzung gewählt, die die Verwendung der entsprechenden Kollektion berücksichtigt und daraus den tatsächlichen `elementType` ableitet.

2. Java-Kollektionen und OCL-Kollektionen bieten jeweils verschieden mächtige Zugriffsschnittstellen. Die Semantik von OCL-Kollektionen ist daher nicht vollständig präzise in Java darstellbar bzw. umgekehrt. Dadurch wird ein formales Mapping schwierig. Allerdings sind die Unterschiede gering und werden deshalb in dieser Arbeit ignoriert.

   Ein Beispiel ist die OCL-Operation `->count()`, die zählt, wie oft ein bestimmtes Element in der Kollektion enthalten ist. Diese Operation läßt sich zwar prinzipiell auch in Java umsetzen, diese Umsetzung erfordert aber die Erzeugung eines `Iterator`s und ist somit potentiell nicht seiteneffektfrei.

- UML-Pakete und Java-Pakete unterscheiden sich leicht. Elemente eines UML-Pakets sind implizit für alle enthaltenen Pakete sichtbar ([UML01], Seite 2-197). Das heißt, eine Klasse `a::b::c::A`, die sich in einem Paket `a::b::c` befindet, „sieht“ implizit alle Klassen in den Paketen `a::b` und `a`.

  Demgegenüber sind Klassen in einem Java-Paket `a.b` nur dann für

[1] d.h. solchen, bei denen alle enthaltenen Objekte eine gemeinsame Oberklasse unterhalb von `java.lang.Object` haben

Klassen im Paket `a.b.c` sichtbar, wenn dies durch ein explizites `import`-Statement deklariert wird. Java-Pakete sehen zwar aus, als wären sie verschachtelt, verhalten sich aber wie völlig unabhängige UML-Pakete ([JSGB00], Abschnitt 7.1).

Das Java-Paket `a.b.c` müßte also korrekt als top-level UML-Paket mit dem Namen „a.b.c“ dargestellt werden.

Die meisten OCL-Werkzeuge haben Probleme mit Namen, die Punkte enthalten, da dies zu Zweideutigkeiten in der Grammatik führt. Da überdies die OCL normalerweise unabhängig von Sichtbarkeit agiert ([W+02b], Abschnitt 4.2.2), wurden für diese Arbeit Java-Pakete wie UML-Pakete behandelt.

4.2 Ein Beispiel

Abbildung 4.1 zeigt die JEFF-Javadoc der Operation `become` des Interfaces `Pool`. In Quelltext 4.1 findet sich die daraus abgeleitete OCL-Spezifikation. Dieses Beispiel soll ein Gefühl für die geleistete Spezifikationsarbeit vermitteln.

Besonders interessant ist an diesem Beispiel, daß der recht kurze Satz „Note: Any existing references to the supplied instance must be replaced with references to the returned object.“ in der OCL-Spezifikation zu einer recht umfangreichen Nachbedingung gerinnt. Dies ist jedoch nicht per se ein Problem der OCL, sondern vielmehr ein Ergebnis der höheren Präzision. Zusätzlich ist es interessant zu bemerken, daß die OCL-Spezifikation Aussagen trifft, die in der Javadoc nicht dargestellt sind (z.B. über den Transfer der Sperren). Auch dies ist ein Ergebnis der höheren Präzision bei der Spezifikation mit OCL, wodurch man auf Unklarheiten hingewiesen wird.

4.3 Aufgetretene Probleme

Während der Spezifikation des Persistenz-Managers traten einige Probleme auf. Diese werden im folgenden beschrieben.

Die Probleme lassen sich im wesentlichen in zwei große Kategorien einteilen:

```
context Pool::become (p : Persistent, c : java::lang::Class)
            : Persistent
pre: not p.oclIsUndefined()
pre: not c.oclIsUndefined()
pre: -- the type associated to the target class must not be abstract
     -- not specified explicitly using an exception, as this is
     -- implemented with a RuntimeException
     not getTypeModel (c).oclIsUndefined() and
     not getTypeModel (c).isAbstract()
post: -- the old object has been replaced by the new
     self.lookup (p.getClass(), p.getPrimaryKey()) = result
post: -- the new object can be looked up under the new class
     self.lookup (c, p.getPrimaryKey()) = result
post: -- the type of the result is the type asked for
     result.oclIsNew() and (result.getClass() = c)
post: -- all locks for the original object have been transferred to
     -- the new object
     getLockedBy (p)->isEmpty() and
     transactions->forAll (th | th.getLockMode (p) = LockMode::NONE)

     and

     getLockedBy (result) = getLockedBy@pre (p) and
     getLockedBy (result)->forAll (th | th.getLockMode (result) =
                                        th.getLockMode@pre (p))
post: -- all links have been changed to use the new object
     getTypeModel (p.getClass()).getRelationshipModels (true)
       ->forAll (rm |
           Let inverse : RelationshipModel = rm.
               getInverseRelationshipModel() in
             rm.getValue (result) = rm.getValue@pre (p) and
             not inverse.oclIsUndefined() implies
               (
                 Set{rm.getValue (p)}->flatten()
                 .oclAsType (Set(Persistent))
                 ->forAll (p1 |
                     Set{inverse.getValue (p1)}->flatten()
                     .oclAsType (Set(Persistent)) =
                       inverse.getValue@pre (p1)->excluding (p)
                       ->including (result)
                   )
               )
         )
post: -- all attribute values have been copied
     getTypeModel (p.getClass()).getAttributeModels (true)
     ->forAll (am |
         am.getValue (result) = am.getValue@pre (p)
       )
exception (PersistenceNotSupportedException):
     -- the class of the persistent object must be a superclass of
     -- the new object
     not p.getClass().isAssignableFrom (c)
exception (PersistentDeletedException):
     -- the persistent object must be an object of this pool
     self.lookup (p.getClass(), p.getPrimaryKey()).oclIsUndefined()
     or self.lookup (p.getClass(), p.getPrimaryKey()) <> p
exception (PersistentNotLockedException):
     -- the persistent object must have been locked for modification
     not Set{LockMode::MODIFY, LockMode::OPTIMISTIC}
         ->includes (getCurrentTransaction().getLockMode (p))
```

Quelltext 4.1: Beispiel: OCL-Spezifikation für `Pool::become()`.

become

```
public Persistent become(Persistent persistent,
                         java.lang.Class instanceType)
                  throws PersistenceException
```

The supplied instance of a persistent class is turned into an instance of the supplied persistent class. The class of the supplied instance must be a superclass of the new class.

Note: Any existing references to the supplied instance must be replaced with references to the returned object.

Parameters:
`persistent` - the persistent object to assume the new class
`instanceType` - the new persistent class for the object to assume
Returns:
the persistent object as an instance of the supplied class
Throws:
`PersistenceException` - if the cast cannot be performed or if an error occurred

Abbildung 4.1: Beispiel: Javadoc für `Pool::become()`.

- **Probleme mit der Ausdrucksmächtigkeit von OCL:** Dies sind Sachverhalte, die sich mit der OCL nicht ausdrücken lassen (Abschnitt 4.3.1).
- **Konzeptuelle Probleme:** Diese Kategorie umfaßt Probleme, die nicht direkt mit der Verwendung der OCL zusammenhängen, sondern allgemein bei Spezifikationen mit Vor- und Nachbedingungen zu erwarten sind (Abschnitt 4.3.2).

Schließlich folgt ein Abschnitt mit weiteren Problemen, die vor allem Mehrdeutigkeiten im Standard betreffen.

Zu einigen dieser Probleme gibt es bereits Lösungsansätze, entweder in der Literatur oder in dieser Arbeit. Die bereits in der Literatur beschriebenen Ansätze werden hier nur kurz zitiert. Ansätze, die erprobt wurden oder deren Erläuterung sehr umfangreich ist, werden in Abschnitt 4.4 dargestellt.

Die beobachteten Probleme traten zwar im Kontext der Spezifikation einer JEFF-Komponente auf, sie sind aber prinzipiell auch bei anderen OCL-Spezifikationen zu erwarten.

4.3.1 Mächtigkeit von OCL

Für die folgenden Punkte gibt es in OCL kein adäquates Ausdrucksmittel.

Exceptions

Die OCL bietet momentan keinerlei Unterstützung für die Spezifikation von Exceptions (bzw. allgemeiner von Signalen).

Das Problem zerfällt in zwei Bereiche:

1. **Exception-auslösende Bedingungen:** Es ist oft notwendig zu spezifizieren, unter welchen Bedingungen eine Operation eine Exception auslöst und in welchem Zustand sie das System in diesem Fall hinterläßt.

2. **Reaktionen auf ausgelöste Exceptions:** Wenn eine Operation andere Operationen aufruft und diese Exceptions auslösen können, so ist es gelegentlich wichtig, die entsprechende Reaktion zu spezifizieren.

Für beide Problembereiche existieren Lösungsvorschläge, die genauer im Abschnitt 4.4.1 erläutert werden.

Threads

OCL kennt das Konzept des Threads bisher nicht, so daß Vor- oder Nachbedingungen, die davon abhängen, in welchem Thread eine Operation aufgerufen wird, nicht (sauber) spezifiziert werden können.

Speziell bei der Spezifikation von Java-Programmen ist es stets möglich, auf die Java-eigene Thread-Unterstützung zurückzugreifen. Ein solcher Spezifikationsstil löst jedoch das prinzipielle Problem nicht, da man zwar die gewünschte Aussage nun aufschreiben kann, Beweise oder Ableitungen aber weiterhin nicht möglich sind. Vor- bzw. Nachbedingungen können zur Laufzeit nur in demselben Thread überprüft werden, in dem die Operation ausgeführt wird bzw. wurde.

Um eine saubere Behandlung von Threads zu ermöglichen, scheint es notwendig, OCL mit einem eigenen Thread-Konzept auszustatten. Eine andere Möglichkeit wäre, die OCL an das UML-Konzept *aktiver Klassen* ([UML01], Seite 2-26) anzukoppeln.

timeout-Parameter

Beim Sperren eines persistenten Objekts ist in JEFF stets die Angabe eines timeout-Wertes möglich. Die Bedeutung ist, daß die Operation abgebrochen wird, wenn die gewünschte Sperre innerhalb des durch das timeout vorgegebenen Zeitrahmens nicht erworben werden konnte.

Die Semantik dieser Operationen ist mit OCL sehr schwer zu spezifizieren. Zum einen kennt OCL kein Konzept von Zeit, zum anderen ist es schwierig auszudrücken, daß es nach Ablauf des timeouts sein *kann*, jedoch nicht sein *muß*, daß die Sperre nicht erworben wurde.

Das erste dieser zwei Probleme ist tatsächlich ein Problem in der OCL. Ohne ein Zeit-Konzept ist eine saubere Spezifikation von zeitabhängigem Verhalten nicht möglich. [SS01] zeigen einen Weg auf, die in der UML angelegten Zeitkonzepte auszubauen und für OCL-Spezifikationen nutzbar zu machen.

Das zweite Problem ließe sich (zumindest im Fall von JEFF) durch eine Möglichkeit zur Spezifikation von Exceptions lösen (s. Abschnitt 4.4.1). Das hängt damit zusammen, daß die entsprechenden JEFF-Operationen bei Überschreiten des timeouts eine Exception auslösen.

Ausschluß von Änderungen zwischen Operationsaufrufen

Für die Sperroperationen in JEFF ist es wichtig, daß sie der einzige Weg sind, um Sperren zu erwerben bzw. wieder abzugeben. Insbesondere verschwindet eine einmal erworbene Sperre nicht einfach, sondern sie kann nur durch (explizites) Beenden der Transaktion wieder aufgelöst werden.

Diese Aussage kann mit der OCL nicht ausgedrückt werden. Dazu wäre eine Erweiterung um temporal-logische Konstrukte notwendig. [CT01] stellt eine solche Erweiterung vor. Der Vorschlag zeigt die prinzipielle Machbarkeit, auch wenn man anhand der dort beispielhaft angegebenen Constraints sieht, daß der Vorschlag noch weiter untersucht werden muß. Die Hauptdiskussionspunkte betreffen:

- die verwendete Syntax,
- die Verwendung von Aktionen und

- den Umgang mit undefinierten Ausdrücken.

Die für eine solche Erweiterung zu verwendende Syntax ist natürlich zu weiten Teilen Geschmackssache. Dennoch kann man einige grundlegende Forderungen, wie stilistische Konformität zur OCL und leichte Lesbarkeit, an sie stellen. Die stilistische Konformität ist gegeben, da es sich einfach um eine Ergänzung um weitere grundlegende logische Operatoren (`always`, `sometime` ...) handelt, die auch nicht in eine Operationssyntax (wie etwa `->forAll()` oder `->exists()`) verpackt werden können. Die leichte Lesbarkeit ist eher problematisch, da die Namen der Operatoren sehr lang sind (beispielsweise `sometime_since_last`). Hier wäre es möglicherweise sinnvoll, nochmals über die Namensgebung nachzudenken.

Substantiellere Probleme bereitet hingegen die Verwendung von Aktionen. Die Autoren verwenden in ihren Beispielen die Namen von Operationen gefolgt von aktuellen Parametern in runden Klammern, um das Ereignis zu bezeichnen, das eintritt, wenn eine bestimmte Operation mit bestimmten Parametern aufgerufen wird. In UML-Termini handelt es sich dabei um eine `CallAction`. In [KW02] wird eine OCL-Klausel vorgestellt, die die Spezifikation von UML-`Actions` ermöglicht. Dieser Vorschlag wird in [W+01] aufgenommen und weiterentwickelt. In beiden Fällen ist die Semantik einer solchen Aktionsklausel für eine Operation jedoch, daß die entsprechende `Action` zwischen Vorbedingungszeit und Nachbedingungszeit einer Operation ausgelöst wurde. Für die Verwendung wie sie in [CT01] vorgeschlagen wird, wäre eine Anpassung der Semantik notwendig, so daß auch über `Actions` geredet werden kann, die zu einem beliebigen Zeitpunkt in der Vergangenheit auftraten bzw. zu einem beliebigen Zeitpunkt in der Zukunft auftreten werden. Gerade im zweiten Fall muß genauer über die Implikationen für die Ausführbarkeit von OCL-Ausdrücken nachgedacht werden.

Die OCL ist eine Sprache mit dreiwertiger Logik. Jeder OCL-Ausdruck kann neben definierten Werten auch den Wert `OclUndefined` annehmen. Jeder OCL-Operator muß daher auch mit undefinierten Werten umgehen können. Gewöhnlich werden alle Operatoren und Funktionen in OCL als strikt angesehen, d.h. wenn die Auswertung eines Teilausdrucks `OclUndefined` ergibt, so ergibt die Auswertung des gesamten Ausdrucks ebenfalls `Ocl-`

`Undefined`. Für bestimmte Boolesche Operatoren ist es jedoch sinnvoll, Abweichungen von dieser Regel zu definieren. Dies wird in [W+01] auch getan. Die Autoren von [CT01] äußern sich jedoch nicht zur Semantik der von ihnen eingeführten Operatoren bei undefinierten Teilausdrücken.

4.3.2 Konzeptuelle Probleme

Die in diesem Abschnitt dargestellten Probleme gehen über fehlende Ausdrucksmittel in der OCL hinaus. Sie beschreiben Grenzen und Schwierigkeiten des Stils der Spezifikation mit Vor- und Nachbedingungen.

Hilfskonstrukte, die nur für die Spezifikation benötigt werden

Für die Spezifikation der Komponentenschnittstelle des Persistenz-Managers werden Informationen über den inneren Zustand der Komponente benötigt, die nicht über die Komponentenschnittstelle zugänglich sind. Insbesondere muß überprüft werden, welche Sperren von welchen Transaktionen für ein bestimmtes persistentes Objekt gehalten werden.

Um diese Informationen zu kapseln, wird die Schnittstelle `Pool` um einige Operationen ergänzt. Diese sind mit dem Stereotyp ≪OCLHelper≫ versehen, um zu markieren, daß sie nicht Teil der Anwendung sind, sondern nur für die Spezifikation eingeführt wurden. Zusätzlich wird noch ein Hilfsinterface — `TransactionHandle` — definiert, das eine einzelne der momentan laufenden Transaktionen beschreibt. Abbildung 4.2 zeigt die neuen Elemente.

Im einzelnen werden die folgenden zusätzlichen Operationen definiert:

`Pool::getLockedBy (p : Persistent) : Set(TransactionHandle)`
liefert die Menge von Transaktionen, die eine Sperre für ein persistentes Objekt halten.

`TransactionHandle::getLockMode (p : Persistent) : LockMode`
definiert die Sperre, welche von einer Transaktion für ein bestimmtes persistentes Objekt gehalten wird.

Diese Operationen werden wie Relationen behandelt, die implizit bei der Spezifikation von Vor- und Nachbedingungen für die Komponentenschnittstelle des Persistenz-Managers definiert werden.

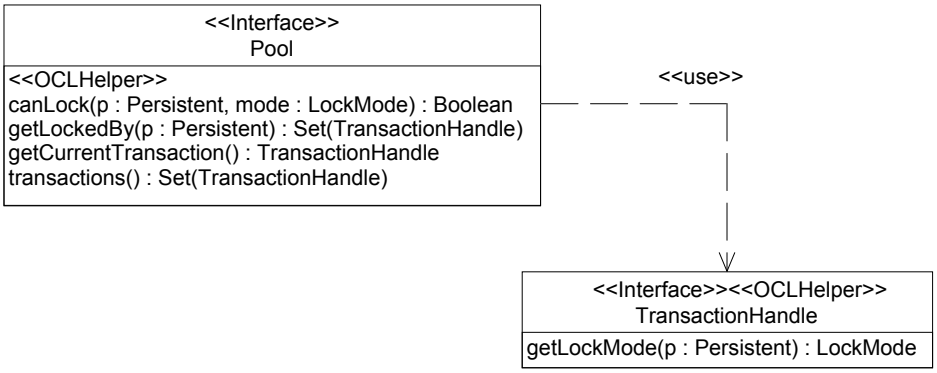

Abbildung 4.2: Zusätzliche Operationen, die nur für die Spezifikation der Komponentenschnittstelle benötigt werden.

Zur Vereinfachung der Spezifikation wird in `Pool` eine Operation `canLock (p : Persistent, mode : LockMode) : Boolean` definiert, welche überprüft, ob ein persistentes Objekt von der aktuellen Transaktion (`getCurrentTransaction()`) mit einer bestimmten Sperre belegt werden kann. Quelltext 4.2 zeigt die Definition dieser Operation.

```
context Pool::canLock (p : Persistent, mode : LockMode) : Boolean
post: -- a shortcut
      -- tests whether the given persistent object can be locked in
      -- the specified way in the current transaction
  result = not getLockedBy (p)->isEmpty() implies
    (getLockedBy (p)->includes (getCurrentTransaction()) or
     getLockedBy (p)->forAll (th |
         not th.getLockMode (p).conflicts (mode)
       )
    )
```

Quelltext 4.2: Definition der `canLock()`-Operation.

Von der Operation `getCurrentTransaction()` wird angenommen, daß sie das der aktuellen Transaktion zugeordnete `TransactionHandle` zurückliefert. Es ist wichtig zu beachten, daß es sich dabei um eine thread-abhängige Spezifikation handelt (s. Seite 21), da die aktuelle Transaktion immer dem aktuellen Thread zugeordnet ist.

Implementationsnachweis

Ein Grund für das Erstellen einer formalen Spezifikation der Komponentenschnittstelle ist die Möglichkeit, verschiedene Implementationen gegen diese Spezifikation zu evaluieren. Dazu werden die Implementationen ebenfalls in der Spezifikationssprache beschrieben und anschließend gezeigt, daß die Schnittstellenspezifikation eine Folge der Implementationsspezifikation ist. Diese Beweise werden gewöhnlich unter Verwendung spezieller, auf die Spezifikationssprache zugeschnittener Kalküle ausgeführt. In [BHTW99] schreiben die Autoren, daß nach ihrer Kenntnis keine solchen Beweiskalküle für OCL existieren.

Auch falls ein solcher Kalkül existierte, würden die eingeführten Hilfskonstrukte (s. Seite 24) ein Problem darstellen. Es wäre notwendig, detailliert zu zeigen, welche Teile der Implementation welche Teilbereiche der Hilfskonstrukte implementieren.

Hohe Komplexität

Komplexität und Umfang der erstellten Spezifikation sind sehr hoch. Die Spezifikation der Komponentenimplementation ist ähnlich schwer zu verstehen, wie die Implementation selbst. Dies schränkt den Nutzen der Spezifikation ein.

Verschiedene Lösungsansätze für dieses Problem werden in Abschnitt 4.4.3 dargestellt.

Paradigmenwechsel

Quelltext 4.3 zeigt die OCL-Definition für die Menge `cascadingDeleteEffect` der persistenten Objekte, die mit einem gegebenen persistenten Objekt mitgelöscht werden müssen, um die referentielle Integrität zu gewährleisten.

Diese Definition ist rekursiv (s. das markierte Auftreten von `cascadingDeleteEffect` in den Zeilen 21 und 36). Außerdem ist es eine deklarative Definition in dem Sinn, daß nur die betroffene Menge beschrieben wird, die Beschreibung selbst aber keine direkten Auswirkungen auf den Systemzustand hat (s. [WK99] zur Erläuterung dieser Auffassung von deklarativ).

```
context Pool
def: -- the set of persistent objects affected by cascading deletes
     -- Note that the recursion used below is indeed finite, as there
     -- will always be a fixpoint in a finite system.
     Let cascadingDeleteEffect (p : Persistent) : Set(Persistent) =
       Set{p}->union (
         self.getTypeModel (p.getClass())
             .getRelationshipModels (true)
           ->iterate (rm, acc: Set(Persistent) = Set{} |
               Let inverse : RelationshipModel =
                   rm.getInverseRelationshipModel() in
                 if (not inverse.oclIsUndefined()) then
                   if (inverse.isRequired()) then
                     -- the relationship is mandatory, i.e., the
                     -- referenced object(s) may need to be deleted
                     if (inverse.isSingleValued()) then
                       acc->union (
                           Set{rm.getValue (p)}->flatten()
                                .oclAsType (Set(Persistent))
                           ->collect (p1 |
                               self.cascadingDeleteEffect (p1)
                             )
                           ->asSet()
                         )
                     else
                       -- multivalued, but required inverse
                            relationship
                       Set{rm.getValue (p)}->flatten()
                              .oclAsType (Set(Persistent))
                         ->iterate (p1; acc1 : Set(Persistent)
                                                = Set{} |
                             if (inverse.getValue (p1)
                                        .oclAsType (Set(Persistent))
                                   ->excluding (p)
                                   ->size() = 0) then
                               -- this was the last reference from
                                    this object, so it must be deleted
                               acc1->including (self.
                                    cascadingDeleteEffect (p1))
                             else
                               acc1
                             endif
                           )
                         ->union (acc)
                     endif
                   else
                     -- only relationship link needs to be removed,
                          but not the referenced object
                     acc
                   endif
                 else
                   acc
                 endif
             )
       )
```

Quelltext 4.3: Die `cascadingDeleteEffect()`–Definition ist rekursiv und deklarativ.

Bei der Implementierung der Persistenz-Manager-Komponente in Java wird man einen imperativen und oft auch einen iterativen Stil bevorzugen. Dabei werden i.allg. das Aufsammeln der betroffenen Objekte und die darauf durchzuführenden Aktionen vermischt. Das bedeutet, daß der Software-Entwickler dasselbe Problem in zwei verschiedenen Paradigmen überdenken muß, einmal für die Spezifikation und anschließend für die Implementation.

Es ist unklar, ob dieses Problem nur in dem in dieser Arbeit betrachteten Fall auftritt. Es wäre möglich, daß ein Entwickler, der zunächst eine rekursive und deklarative Spezifikation erstellt hat, anschließend auch eine rekursive Implementierung bevorzugt sowie das Bilden der Menge betroffener Objekte von der Bearbeitung dieser Objekte trennt (also einen Teil des deklarativen Paradigmas beibehält). Spätestens in einer Optimierungsphase wird aber meist ein Paradigmenwechsel stattfinden, da rekursive und deklarative Algorithmen oft weniger effizient sind als iterative und imperative.

Zu beachten ist, daß ein Teil der Komplexität des Beispiel-Ausdrucks von der Komplexität des Beispiel-Problems herkommt. So müssen mehrwertige Relationen anders behandelt werden als einwertige. Außerdem müssen Sonderfälle wie undefinierte Relationen beachtet werden.

Systemgrenzenüberschreitende Bedingungen

Formale Spezifikationen beschreiben stets das Verhalten eines Systems. Alles, was sich außerhalb der Systemgrenzen befindet, ist daher durch die Spezifikation nicht erfaßbar. Im Fall des Persistenz-Managers betrifft das vor allem die folgenden Punkte:

- Die Spezifikation müßte eigentlich auch Aussagen über die Veränderungen in der Datenbank treffen. Die Datenbank liegt aber außerhalb des Systems Persistenz-Manager. Auf der Ebene der Schnittstellenspezifikation ist ein direkter Zugriff auf die Schnittstelle zur Datenbank ebenfalls nicht möglich. Somit befindet sich die Datenbank außerhalb der Systemgrenzen und Veränderungen in der Datenbank lassen sich nicht spezifizieren.
- Die Überprüfung der Sperrverträglichkeit wird von der Implementation der Komponente teilweise an die Datenbank übertragen. Damit

sind die Bedingungen für die Sperrverträglichkeit potentiell von anderen Anwendungen, die auf derselben Datenbank arbeiten, abhängig. Dies ist eine Überschreitung der Systemgrenzen, so daß eine vollständige Spezifikation nicht möglich ist.

4.3.3 Weitere Probleme

Konstruktoren

Die Darstellung von Konstruktoren ist im UML-Standard nicht eindeutig definiert. Es wird ein Stereotyp ≪create≫ für `BehavioralFeature` eingeführt, mit der Bedeutung „Specifies that the designated feature creates an instance of the classifier to which the feature is attached“ ([UML01]).

Dennoch bleiben noch mehrere Varianten, Konstruktoren zu modellieren:

1. Als statische Operation mit der Klasse als Ergebnistyp. Diese Operation wäre als *Factory-Operation* zu verstehen, die ein neues Objekt der Klasse erzeugt und zurückliefert.

2. Als nicht-statische Operation, die direkt nach der Instantiierung aufgerufen wird. In diesem Fall würde die eigentliche Instantiierung durch einen speziellen Operator der Programmiersprache (z.B. `new` in Java) übernommen, welcher direkt im Anschluß an die eigentliche Objekterzeugung den Konstruktor zur Initialisierung aufruft.

Der UML-Standard gibt keine Auskunft darüber, welche Variante zu nutzen ist. Dieser Unterschied ist aber für die Spezifikation von Konstruktoren in OCL von Bedeutung.

In [HK99] (Seiten 250f.) wird die zweite Variante verwendet. Daher werden auch im Rahmen dieser Arbeit Konstruktoren als nicht-statische Operationen ohne Ergebnistyp spezifiziert.

4.4 Lösungsansätze

In diesem Abschnitt werden für einige der aufgetretenen Probleme gefundene Lösungsansätze diskutiert.

4.4.1 Exceptions

Auf Seite 21 wurde das Problem geschildert, daß OCL keine Möglichkeit bietet, um Exceptions zu spezifizieren. Mögliche Lösungen sollen im folgenden diskutiert werden.

Wie in der Problembeschreibung geschildert, zerfällt das Problem in zwei Teile, die Spezifikation von Bedingungen, unter welchen Exceptions geworfen werden, und die Spezifikation der Reaktion auf aufgetretene Exceptions. Dementsprechend gliedert sich dieser Abschnitt in zwei Unterabschnitte, die sich diesen beiden Teilproblemen widmen.

Anschließend folgt ein dritter Unterabschnitt, der sich mit „Desastern“ und „Illegalen Situationen“ — einer von sd&m in die Diskussion gebrachten Anwendung dieser Beschreibungsmöglichkeiten — beschäftigt.

Die Exception-Klausel

In [SF99] werden sogenannte *e-post-condition*s eingeführt, die die Bedingungen spezifizieren, unter denen eine Operation eine bestimmte Exception auslöst. Bei diesen Klauseln handelt es sich im Prinzip um etwas ähnliches wie Nachbedingungen. Das Auslösen einer Exception eines bestimmten Typs wird als ein mögliches Ergebnis eines Operationsaufrufs gewertet. Die gewöhnliche Nachbedingung beschreibt dann den Systemzustand bei normaler Beendigung des Operationsaufrufs sowie — unter Verwendung von `@pre` — die Bedingungen, die zu diesem Ausgang führten. Analog beschreibt eine *e-post-condition* die Bedingungen bei Beendigung der Operation durch das Auslösen einer bestimmten Exception sowie — unter Verwendung von `@pre` — die Bedingungen, die dazu führten.

Dieser Vorschlag ist geeignet, um die Bedingungen zu spezifizieren, unter denen eine Operation Exceptions auslöst. Allerdings ist die gewählte Syntax sehr stark durch mathematische Symbolik motiviert und orientiert sich weniger an der Syntax objektorientierter Programmiersprachen. Dies war aber eine der Hauptmotivationen bei der Entwicklung von OCL ([WK99], Seite 8). Deshalb wird in dieser Arbeit eine leicht geänderte Syntax vorgeschlagen und in Anlehnung an den in [W+02b], Kapitel 7 verwendeten Stil dargestellt.

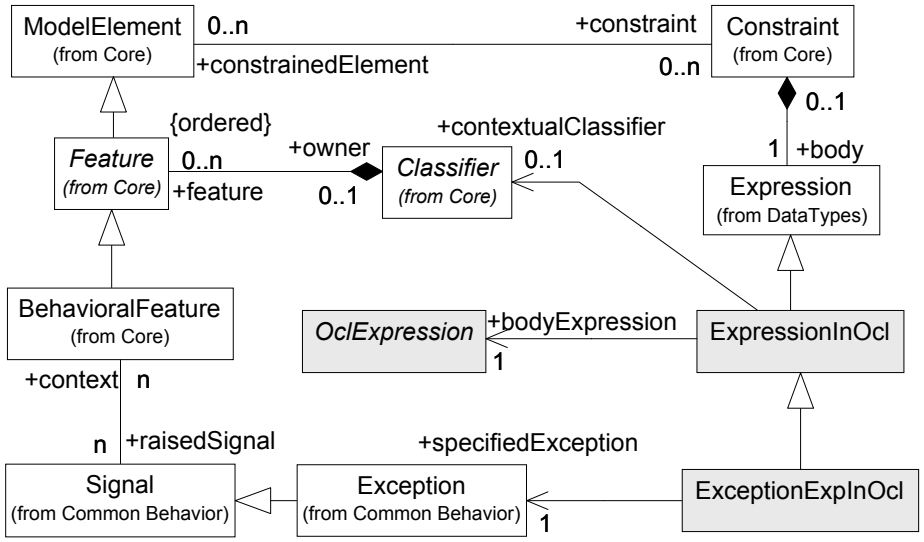

Abbildung 4.3: Verwendung von `ExceptionExpInOcl` als Exception-Klausel für ein `BehavioralFeature`.

Abbildung 4.3 zeigt einen Ausschnitt aus dem UML-Metamodell. Dieser stellt die Einbindung eines OCL-Ausdrucks als Exception-Klausel für ein `BehavioralFeature` dar. Dazu wurde eine neue Klasse `ExceptionExpInOcl` geschaffen, die eine Exception-Klausel darstellt. Zusätzlich zu den von `ExpressionInOcl` ererbten Assoziationen `contextualClassifier` und `bodyExpression` definiert diese Klasse noch eine Assoziation `specifiedException`, die die spezifizierte Exception definiert. In einer Exception-Klausel steht die Variable `exception` zur Verfügung, die eine Referenz auf die ausgelöste Exception (d.h. die `SendAction`) repräsentiert. Die einzelnen Argumente der Action können dabei wie Attribute der Variablen `exception` verwendet werden.

Die folgenden Wohlgeformtheitsregeln (Well-Formedness Rules, WFRs) gelten für `ExceptionExpInOcl`:

[1] Die Klausel ist Teil eines Constraints mit dem Stereotyp «exception».

```
context ExceptionExpInOcl
inv: self.constraint.stereotype.name = 'exception'
```

[2] OCL-Constraints des Stereotyps ≪exception≫ sind Instanzen von `ExceptionExpInOcl`.

```
context Constraint
inv: (stereotype.name = 'exception' and body.language='OCL')
    implies
       body.oclIsKindOf (ExceptionExpInOcl)
```

[3] Die Exception-Klausel ist einem `BehavioralFeature` und einer Exception zugeordnet, die von diesem `BehavioralFeature` ausgelöst werden kann.

```
context ExceptionExpInOcl
inv: self.constraint.constrainedElement->size() = 1 and
     self.constraint.constrainedElement->any (true)
       .oclIsKindOf (BehavioralFeature) and
     self.constraint.constrainedElement->any(true)
       .oclAsType (BehavioralFeature).owner->size() = 1 and
     contextualClassifier = self.constraint.constrainedElement
                              ->any (true)
                               .oclAsType (BehavioralFeature)
                               .owner and
     self.constraint.constrainedElement->any (true)
       .oclAsType (BehavioralFeature).raisedSignal
      ->includes (self.specifiedException)
```

[4] Der OCL-Ausdruck ist vom Typ Boolean.

```
context ExceptionExpInOcl
inv: self.bodyExpression.type.name = 'Boolean'
```

Ein Beispiel für die Verwendung der Exception-Klausel wird in Quelltext 4.4 dargestellt. Der zweite Teil stellt hierbei zweifelsohne eine Überspezifikation dar. Allerdings ist es mit der sehr einfachen Exception-Struktur des Persistenz-Managers, die keine Exceptions mit anderen Attributen als der Nachricht enthält, schwer, sinnvollere Beispiele zu finden. In der vorgeschlagenen konkreten Syntax wird die Exception-Klausel eingeleitet durch das Schlüsselwort `exception` gefolgt von der Angabe der zu spezifizierenden Exception in runden Klammern. Anschließend folgt der OCL-Ausdruck.

Behandlung geworfener Exceptions

[SS02] will die Behandlung von Exceptions, die während des Ablaufs einer Operation auftreten, in eine explizite, zusätzliche Klausel verlegen. Diese

```
context Pool::getInstanceType (typeName : String) : java::lang::Class
...
exception (PersistentClassNotFoundException):
  -- there's no class whose type model has the given name
  instanceTypeForNamedTypeModel (typeName).oclIsUndefined() and
  exception.getMessage() = 'Persistent type '.concat (typeName)
                                              .concat (' cannot be
                                                  determined')
```

Quelltext 4.4: Ein Beispiel für die Verwendung der Exception-Klausel.

Idee scheint problematisch, insbesondere erschwert sie die Spezifikation von Fällen, in denen die Reaktion auf eine Exception zum Auslösen weiterer Exceptions führen kann.

Die interessanten Fälle sind diejenigen, bei denen eine aufgerufene Operation eine Exception auslösen kann. Deshalb wird die folgende Spezifikationstechnik vorgeschlagen: In [W+02b] wird die Möglichkeit gegeben, Operationsaufrufe durch Action-Klauseln zu spezifizieren. Diese erzeugen eine Menge von `OclMessage`-Instanzen, über die auf verschiedene Eigenschaften der Action zugegriffen werden kann. `OclMessage` wird um die folgende Operation erweitert:

`exceptionalResult() : OclAny` liefert die ausgelöste Exception.

```
pre: hasReturned()
```

Für die bereits existierende Operation `result()` wird die Vorbedingung angepaßt:

```
pre: hasReturned() and exceptionalResult().oclIsUndefined()
```

D.h. `exceptionalResult` liefert `OclUndefined`, wenn keine Exception auftrat.

Quelltext 4.5 zeigt ein Beispiel für die Anwendung dieser Syntax (s. Zeilen 19 und 31).

Die Semantik und konkrete Syntax der OclMessage-Ausdrücke hat sich zwischen den Versionen 1.4 und 1.5 des Vorschlags für die OCL 2.0 etwas verändert. Die hier dargestellte Form entspricht der Version 1.5. In der Spezifikation, die sich auf der CD-ROM findet, wird noch die Form der Version 1.4 verwendet. Die Unterschiede sind jedoch nicht so gravierend, daß sich daraus Probleme ergäben.

```
context Relation::RelationIterator
def: -- a helper to be used by getNextObject
     Let findNextObjectLoop (persistents : Sequence(
                                             Sequence(
5                                            java::lang::Object)),
                             curIndex    : Integer,
                             newIndex    : Integer,
                             foundObject : java::lang::Object)
                           : Boolean =
10     if (curIndex < persistents->size()) then
         Let key : Sequence(java::lang::Object)
                 = persistents->at (curIndex + 1) in
           if lock then
             Let message : OclMessage
15                       = reference.getPersistenceManager()
                         ^^lookup (reference.getInstanceType(),
                                   key, lockMode, timeout) in
               message.hasReturned() and
               if (message.exceptionalResult().oclIsUndefined()) then
20               newIndex = curIndex + 1 and
                 foundObject = message.result()
               else
                 findNextObjectLoop (persistents, curIndex + 1,
                     newIndex, foundObject)
               endif
25           else
               Let message : OclMessage
                           = reference.getPersistenceManager()
                           ^^lookup (reference.getInstanceType(),
                                     key) in
30               message.hasReturned() and
                 if (message.exceptionalResult().oclIsUndefined()) then
                   newIndex = curIndex + 1 and
                   foundObject = message.result()
                 else
35                 findNextObjectLoop (persistents, curIndex + 1,
                       newIndex, foundObject)
                 endif
             endif
         else
           -- no more objects
40         newIndex = curIndex + 1 and
           foundObject.oclIsUndefined()
         endif
```

Quelltext 4.5: Ein Beispiel für die Verwendung der `exceptionalResult()`-Operation.

Zu beachten ist, daß dieser Vorschlag keinerlei Garantien gibt, daß tatsächlich alle möglichen Reaktionen auf alle möglichen Exceptions spezifiziert werden. Dieses Problem wird in [SS02] durch die explizite Spezifikation ausgelöster Exceptions behandelt, sowie durch die Forderung, daß für jede Exception, die im Rumpf einer Spezifikation auftreten kann, auch eine entsprechende behandelnde Klausel existieren muß.

Ein verbleibendes kleines Problem ist die Frage der Behandlung von Exceptions bei {query}-Operationen. Operationen, die den Systemzustand nicht ändern, können in der UML mit dem Tag {query} markiert werden und anschließend in der OCL direkt verwendet werden. Ein Problem tritt auf, wenn diese Operationen Exceptions auslösen können. Obwohl natürlich die Reaktion auf ausgelöste Exceptions stets explizit spezifiziert werden kann, indem die Operation im Rahmen einer Action-Klausel verwendet wird, ist der Aufwand dafür relativ groß, und es besteht die Gefahr, den eigentlichen Spezifikationsinhalt zu überdecken. Für viele Fälle ist es angemessener, die Operation direkt zu verwenden. Allerdings benötigt man für eine vollständige Spezifikation eine Möglichkeit, zwischen normalem Verhalten und Exception zu unterscheiden. In der im Rahmen dieser Arbeit erstellten Spezifikation wird unterstellt, daß Ausdrücke mit {query}-Operationen `OclUndefined` ergeben, wenn eine Exception auftrat.

Eine grundsätzlichere Frage ist es, ob Operationen, die Exceptions auslösen, dennoch als {query} markiert werden dürfen. Die Frage ist hierbei, ob das Auslösen einer Exception den Systemzustand ändert (s. [UML01], Seite 2-25). Exceptions beeinflussen den Kontrollfluß. Das einzige Argument dafür, daß Exceptions den Systemzustand ändern, ist die Tatsache, daß in objektorientierten Sprachen, beim Auslösen einer Exception normalerweise ein neues Objekt erzeugt wird. Das verändert zumindest das Ergebnis von <*exception class*>`::allInstances()`. Da aber die Definition von Constraints unter Verwendung von <*exception class*>`::allInstances()` relativ selten sein dürfte, werden Operationen, die Exceptions auslösen, in dieser Arbeit als {query}-Operationen zugelassen.

„Desaster“ und „Illegale Situationen“[2]

In Informationssystemen, wie sie von sd&m entwickelt werden, werden Exceptions zu zwei verschiedenen Zwecken eingesetzt:

1. **Illegale Situation:** Um unzulässige Systemzustände zu markieren, die nach Spezifikation nicht auftreten dürften.
2. **Desaster:** Um Systemfehler (also z.B. I/O-Probleme) an die Anwendung weiterzureichen.

Auf Entwurfsebene wünscht man sich eine Möglichkeit, um diese beiden sehr verschiedenen Fälle getrennt darzustellen. Die vorgestellte Exception-Klausel bietet eine hervorragende Möglichkeit, dies umzusetzen. Dazu werden illegale Situationen durch Vorbedingungen spezifiziert, Desaster hingegen mit Hilfe von expliziten Exception-Klauseln.

4.4.2 Hilfskonstrukte

Im folgenden Abschnitt wird zunächst dargestellt, wie sich mit Hilfe von ICL Hilfskonstrukte vermeiden lassen. Anschließend wird diskutiert, wie Hilfskonstrukte als Hinweis auf mögliche Schwächen der Schnittstellendefinition verstanden werden können.

ICL

Die Spezifikation mit Hilfe von Vor- und Nachbedingungen erfordert expliziten Zugriff auf den inneren Zustand einer Komponente. Wenn dieser Zugriff nicht durch Operationen der Komponentenschnittstelle möglich ist, müssen entsprechende Hilfskonstruktionen eingeführt werden (s. Seite 24).

In [BHTW99] wird ICL (wahrsch. für „Interface Constraint Language“), eine von OCL abgeleitete Sprache, vorgestellt, die algebraische Spezifikationen erlaubt. Dabei sind die Ergebnisse und Parameter von Operationen Systemzustände, so daß alle Zustandsinformationen implizit in der Spezifikation verwendet werden können. Dadurch ist die Spezifikation von Schnittstellen auch ohne Hilfskonstrukte möglich. [BHTW99] nutzt diese Eigenschaft, um Implementationsbeziehungen nachzuweisen.

[2] „Desaster“ und „Illegale Situationen“ sind von sd&m entwickelte Konzepte.

Wichtig ist hierbei, daß die vorgestellte Sprache nicht nur eine Erweiterung der OCL darstellt, sondern eine eigenständige Sprache, die auf einem anderen Prinzip beruht als OCL. Deshalb ist eine entsprechende Spezifikation mit der OCL selbst nicht möglich.

Vollständigkeit von Schnittstellen

Bei der Definition von Interfaces ist zwischen Schlankheit und Vollständigkeit abzuwägen. Hilfskonstrukte, die für die Spezifikation einer Schnittstelle notwendig werden, sollten immer auch Anlaß sein, über möglicherweise fehlende Operationen in der Schnittstelle nachzudenken. Beispielsweise ist zu überlegen, ob `Pool` zusätzliche Methoden `canLock (p : Persistent, mode : LockMode) : Boolean`, `setTransactionAdapter (ta : TransactionAdapter)` und `getTransactionAdapter() : TransactionAdapter` erhalten sollte.

Die Notwendigkeit von Hilfskonstrukten für die Spezifikation läßt allerdings nicht automatisch auf fehlende Vollständigkeit der Schnittstelle schließen. Es ist weiterhin nötig, im Entwurfsprozeß zwischen den beiden genannten Polen abzuwägen. Das Erstellen einer Spezifikation hat jedoch den Vorteil, daß solche potentiellen Probleme frühzeitig aufgedeckt werden. Eine Suche nach verallgemeinerbaren Kriterien für unvollständige Schnittstellen würde den Rahmen dieser Arbeit überschreiten. Leider finden sich auch in der Literatur keine entsprechenden Hinweise.

4.4.3 Komplexität

Auf Seite 26 wird das Problem der hohen Komplexität der OCL-Spezifikationen beschrieben. Zur Lösung dieses Problems wurden verschiedene Ansätze getestet.

Gezielte Unterspezifikation

Ein möglicher Ansatz zur Reduktion der Komplexität ist die Reduktion des Umfangs der Spezifikation. Dabei müssen natürlich Abstriche bei der Vollständigkeit gemacht werden. Eine sinnvolle Möglichkeit (die auch von sd&m so vorgeschlagen und als sinnvoller erster Schritt gesehen wird) stellt

dabei dar, sich auf die Spezifikation der zulässigen Wertebereiche von Parametern und Funktionsergebnissen zu beschränken. Die gesamte Semantik ist dabei aus der Spezifikation ausgeklammert. Dadurch wird eine Umfangsreduktion um ca. 60% von durchschnittlich 24,5 Constraints pro Klasse auf durchschnittlich 9,78 Constraints pro Klasse erreicht.

Eine Fassung der Spezifikation der Komponentenschnittstelle, welche nur die Wertebereiche der Parameter beschreibt, findet sich auf der beiliegenden CD-ROM.

Subject-Oriented Design

In den letzten Jahren sind verschiedene Vorschläge unter Namen wie *Aspektorientierung*, *Subjektorientierung* erschienen (s. u.a. [KLM+97], [HO93], [OT99], [CHOT99]), die eine Strukturierung von Software nach problembezogenen Kriterien (Features) im Gegensatz zur Strukturierung nach technologischen Kriterien (Module, Klassen etc.) zum Ziel haben.

Ein Grund für diese Vorschläge ist die bessere Verständlichkeit der Software durch logischere Strukturierung. Daher erscheint es angebracht, die vorgeschlagenen Prinzipien auch auf Spezifikationen in OCL zu übertragen. Dazu werden zwei zusätzliche Schlüsselworte (`subject` und `endsubject`) eingeführt, die zur Markierung einzelner Subjects im Sinne von [CHOT99] dienen. Ein weiteres Schlüsselwort `import_subject` wird eingeführt, um eine zusätzliche Strukturierungsmöglichkeit (Subjekte, die gemeinsam genutzte Funktionen mehrerer anderer Subjekte kapseln) zur Verfügung zu stellen. Die gesamte Spezifikation ergibt sich durch einfaches Mischen aller Subjekte, d.h. sämtliche Constraints müssen erfüllt werden, unabhängig davon, in welchem Subjekt sie spezifiziert wurden. Vereinfacht kann man sich das als `and`-Verknüpfung der Constraints vorstellen. Quelltext 4.6 zeigt ein Beispiel für die Anwendung dieser Syntax. Die konkreten Constraints wurden dabei ausgelassen. Im Sinne der *Subjektorientierung* erscheint es sinnvoll, die Spezifikation in eine Datei pro Subjekt (und eventuell Klasse) aufzuteilen.

Ein objektiver Nachweis geringerer Komplexität ist schwierig zu führen, insbesondere versagt die Anwendung der Halstead-Metriken (s. Abschnitt 5.3.2). Das liegt wahrscheinlich daran, daß diese Metrik auf den Anzahlen

```
package com::sdm::jeff::persistence

...

subject Mapping
-- This subject deals with mappings between java classes
-- and type models

...

endsubject -- Mapping


subject Locking
-- This subject specifies the behavior of the component
-- regarding locking of persistent objects

import_subject Mapping  -- so that statements can be made using
                        -- cascadingDeleteEffect

...

endsubject -- Locking


subject Repository
-- This subject deals with the repository functionality, i.e. with
-- creating, deleting, casting, and looking up persistent objects.

import_subject Mapping  -- so that statements can be made using
                        -- cascadingDeleteEffect and type model
                        -- information

...

endsubject -- Repository

endpackage
```

Quelltext 4.6: Ein Beispiel für die Anwendung von Prinzipien des subjektorientierten Designs.

von Operatoren und Operanden beruht und aus dieser Sicht einfach zwei Operatoren mehr verwendet werden (Erläuterungen zur Halstead-Metrik s. Seite 57).

Eine Fassung der Spezifikation der Komponentenschnittstelle unterteilt in Subjekte befindet sich auf der beiliegenden CD-ROM.

Spezifikations-Schablonen[3]

Muster können helfen, komplexe Dinge zu strukturieren. Daher wäre ein Katalog von Spezifikationsmustern/Spezifikationsschablonen nützlich. Ein solcher Katalog würde Schablonen für die Spezifikation von Standardfällen — wie z.B. der Angabe, daß Parameter nicht `null` sein dürfen — enthalten.

Bei der Suche nach entsprechend verallgemeinerbaren Strukturen in der erstellten Spezifikation wurde jedoch schnell klar, daß solche Strukturen kaum existieren. Die einzigen Muster, die gefunden wurden, sind:

- **Definierter Parameterwert:** Soll spezifiziert werden, daß für einen Parameter einer Operation nicht `null` übergeben werden darf, so instanziiere man folgende Schablone als Vorbedingung der Operation:

  pre: not <*parameter name*>**.oclIsUndefined()**

  Wobei <*parameter name*> durch den Namen des Parameters zu ersetzen ist. In früheren OCL-Spezifikationen (u.a. in der Spezifikation der OCL 1.4 als Teil der UML 1.4 [UML01]) findet sich gelegentlich noch die Form:

  pre: not <*parameter name*>**->isEmpty()**

  Diese Syntax verwendet die Tatsache, daß OCL in diesem Fall eine implizite Menge erzeugt, die nur den aktuellen Parameter enthält. Ist dieser undefiniert, so ist die erzeugte Menge leer. Allerdings funktioniert dieses Vorgehen nur, wenn <*parameter name*> einen Parameter beschreibt, dessen Typ nicht wieder eine Kollektion ist. Für Kollektionen ist es durchaus ein Unterschied, ob sie leer oder undefiniert sind. Auch ist die Semantik der impliziten Mengenbildung nicht genau

[3] Dies ist ein Vorschlag von sd&m.

geklärt. Es ist insbesondere nicht explizit definiert, ob der Mengenkonstruktor strikt ist. In diesem Fall wäre eine Menge, die ein undefiniertes Element enthält, nicht leer, sondern undefiniert.

- **Parameter, die `null` sein dürfen:** Kann ein Parameter auch den Wert `null` annehmen, so ist darauf zu achten, daß alle Verwendungen dieses Parameters im Regelfall in das folgende Konstrukt eingebunden werden:

```
if (<parameter name>.oclIsUndefined()) then
  -- Reaktion auf null Parameter
  ...
else
  -- Reaktion auf definierten Parameterwert
  ...
endif
```

- **Faktorisieren von Spezifikationen:** Wann immer Teile der Spezifikation mehrfach verwendet werden, ist zu überlegen, ob diese nicht in einen ≪definition≫- oder ≪invariant≫-Constraint faktorisiert werden können. Ein Beispiel für einen ≪invariant≫-Constraint ist die Verwendung von `LockMode`. Statt in jeder Operation mit einem Parameter `mode : LockMode` die folgende Vorbedingung zu wiederholen:

```
pre: Set{LockMode::NOLOCK,
         LockMode::OPTIMISTIC,
         LockMode::REFERENCE,
         LockMode::ACCESS,
         LockMode::MODIFY,
         LockMode::DELETE}->includes (mode)
```

kann diese in eine Invariante wie folgt ausgelagert werden:

```
context LockMode
inv: Set{LockMode::NOLOCK,
         LockMode::OPTIMISTIC,
         LockMode::REFERENCE,
         LockMode::ACCESS,
         LockMode::MODIFY,
         LockMode::DELETE}->includes (self)
```

Bei der Auslagerung in ≪definition≫-Constraints ist zu beachten, daß diese im Prinzip der Definition von Hilfsoperationen entsprechen. Daher ist die Verwendung mancher Konstrukte (insbesondere der Action-Klausel) innerhalb eines ≪definition≫-Constraints zumindest diskussi-

onswürdig. [SS02] geht einen anderen Weg und schlägt unter den Namen *Parameterisiertes Prädikat* und *Parameterisierte Funktion* eine Faktorisierungsmethode vor, die ähnlich wie C++-Makros arbeitet. Teile eines Constraints erhalten einen eigenen Namen und können dann unter diesem Namen in beliebigen Constraints verwendet („instanziiert“) werden. Dabei wird quasi der Text in den Ziel-Constraint kopiert, und die Auswertung erfolgt auf dem Resultat dieser Verbindung. Daher können alle Konstrukte, die im Ziel-Constraint zulässig sind, auch im Prädikat bzw. der Funktion verwendet werden. Dieser Vorschlag basiert auf Ideen aus [DW98].

Der Grund für die geringe Zahl verallgemeinerbarer Hinweise scheint die falsche Abstraktionsebene zu sein. Es wäre durchaus möglich, noch weitere Schablonen zu finden, jedoch nicht, ohne bereits Aussagen über Architektur und Entwurfsentscheidungen zu treffen. Damit wäre man dann allerdings auf der Ebene, die durch Publikationen wie [GHJV94] und [Fow97] abgedeckt wird.

Spezifikation von Schnittstellen

Bei genauerer Betrachtung der Spezifikation läßt sich feststellen, daß es einen großen Unterschied in der Komplexität zwischen Schnittstellenspezifikation und Implementationsspezifikation gibt. Die erstere ist relativ leicht verständlich, während die zweite sehr komplex ist und auch gegenüber dem Quelltext kaum einen Informationsgewinn bringt. Tatsächlich ergibt sich der (nicht quantifizierte) Eindruck, daß Komplexität und Informationsgehalt der Implementationsspezifikation auf dem gleichen Niveau liegen wie für den Quelltext selbst.

Unter diesen Bedingungen ist die vollständige Spezifikation der Implementation zu Dokumentationszwecken nicht sinnvoll. Da mit der OCL auch ein Implementationsbeweis nicht zu führen ist (s. Seite 26), ist es somit sinnvoll, die OCL-Spezifikation auf die Schnittstelle zu beschränken.

„Literate Programming“

Der Titel dieses Abschnitts ist etwas irreführend. Der hier beschriebene Vorschlag beruht nicht explizit auf Konzepten des *literate programming* (s. u.a. [Knu01]), die Idee leitet sich aber davon her. Ursprünglich entstand der Vorschlag beim Versuch, im Rahmen der Evaluation der Spezifikation eine Bewertung der Abdeckung der natürlichsprachlichen Dokumentation durch die OCL-Spezifikation zu ermöglichen (s. Abschnitt 5.2). Dazu wurden die OCL-Constraints in die Javadoc eingebettet, so daß einer natürlichsprachlichen Aussage jeweils direkt die entsprechenden Constraints zugeordnet werden können. Abbildung 4.4 zeigt einen Ausschnitt aus der entsprechenden Javadoc für `Pool`.

Diese Form der Einbettung in die Javadoc fördert das Verständnis der Spezifikation. Die — relativ leicht verständliche — natürlichsprachliche Dokumentation erhält ein formales Fundament durch die direkte Einbindung der OCL-Ausdrücke.

Um die Einbettung technisch zu realisieren, müssen neue Javadoc-*Tags* definiert werden, die ein zu entwickelndes *Taglet* ([Jav]) auswertet und in HTML umsetzt. Die neuen *Tags* sind *Inline-Tags*, die an der Stelle eingesetzt werden, an der das OCL-Constraint in der generierten Dokumentation eingebunden werden soll. Tabelle 4.2 stellt die neuen *Tags* vor. Quelltext 4.7 zeigt, wie der Javadoc-Kommentar aussähe, der die in Abbildung 4.4 gezeigte Dokumentation erzeugt.

Die Notwendigkeit der meisten neuen *Tags* ist direkt einsichtig. Das einzige *Tag*, welches einer Erklärung bedarf, ist `'{@oclRef'`<*name*>`'}'`. Dieses wird benötigt, da die natürlichsprachliche Javadoc-Spezifikation Redundanzen enthält. So taucht im Beispiel `Pool::lookup` die Aussage, daß `instanceType Persistent` implementieren muß, zweimal auf, nämlich im normalen Kommentartext (Zeile 2) und in der Spezifikation des Parameters (Zeile 14). Hier verwendet man das `@oclPre`-*Tag* im normalen Kommentartext, um dann in der Spezifikation des Parameters dieses Constraint einfach nur zu referenzieren. Das *Taglet* kann dann den Text des Constraints an die entsprechende Stelle in der generierten Dokumentation kopieren oder auch nur einen Verweis aufnehmen. Gleichzeitig findet ein Tool, das die OCL-Spezifikation

lookup

```
public Persistent lookup(java.lang.Class instanceType,
                         java.lang.Object[] primaryKey,
                         LockMode lockMode)
                  throws PersistenceException
```

An existing instance of the supplied persistent class

```
pre: -- the demanded type is a persistent type
     clsPersistent.isAssignableFrom (instanceType)
```

is searched for the supplied primary key and returned. Before the object is returned, it will be locked in the supplied `LockMode`.

```
post: -- the result is locked as specified
     getCurrentTransaction().getLockMode (result) = lockMode and
     getLockedBy (result) = getLockedBy@pre (result)
       ->including (getCurrentTransaction())
```

Parameters:

`instanceType` - the persistent class of the object to be found

```
pre: -- the demanded type is a persistent type
     clsPersistent.isAssignableFrom (instanceType)
```

`primaryKey` - the primary key of the object to be found

`lockMode` - the required `LockMode` for the object

Returns:

the persistent object with the given primary key

```
post: result = self.lookup (instanceType, primaryKey)
```

Throws:

`PersistenceException` - if the object cannot be found or if an error occurred

```
pre: -- the object exists
     not self.lookup (instanceType, primaryKey).oclIsUndefined()
pre: -- the potential result can be locked as specified
     canLock (
         lookup (instanceType, primaryKey),
         lockMode
       )
```

See Also:

`PersistentNotFoundException`

Abbildung 4.4: Einbettung der OCL-Constraints in die Javadoc.

Syntax	**Bedeutung**
'{@oclAttDef'[<*name*>]':'<*ocl ausdruck*>'}'	<*ocl ausdruck*> wird als ≪definition≫-Constraint für ein zusätzliches Attribut interpretiert.
'{@oclOpDef'[<*name*>]':'<*ocl ausdruck*>'}'	<*ocl ausdruck*> wird als ≪definition≫-Constraint für eine zusätzliche Operation interpretiert.
'{@oclPre'[<*name*>]':'<*ocl ausdruck*>'}'	<*ocl ausdruck*> wird als ≪precondition≫-Constraint für die dokumentierte Operation interpretiert. Dieses *Tag* ist nur in der Dokumentation einer Operation zulässig.
'{@oclPost'[<*name*>]':'<*ocl ausdruck*>'}'	<*ocl ausdruck*> wird als ≪postcondition≫-Constraint für die dokumentierte Operation interpretiert. Dieses *Tag* ist nur in der Dokumentation einer Operation zulässig.
'{@oclInv'[<*name*>]':'<*ocl ausdruck*>'}'	<*ocl ausdruck*> wird als ≪invariant≫-Constraint für die dokumentierte Klasse interpretiert.
'{@oclRef'<*name*>'}'	Dieses *Tag* fügt eine Kopie des benannten Constraints an dieser Stelle in die Javadoc ein. Da der Constraint bereits definiert wurde, wird er von anderen Werkzeugen als dem Doclet nicht berücksichtigt.

Tabelle 4.2: Javadoc-*Tags* zur Einbettung von OCL-Constraints.

```
/**
 * An existing instance of the supplied persistent class
 * {@oclPre isPersistentClass: -- the demanded type is a persistent
 *                             -- type
 *    clsPersistent.isAssignableFrom (instanceType)}
 * is searched for the supplied primary key and returned. Before
 * the object is returned, it will be locked in the supplied
 * <code>LockMode</code>.
 * {@oclPost resultIsLocked: -- the result is locked as specified
 *    getCurrentTransaction().getLockMode (result) = mode and
 *    getLockedBy (result) = getLockedBy@pre (result)
 *      ->including (getCurrentTransaction())}
 *
 * @param instanceType the persistent class of the object to be
 *                     found {@oclRef isPersistentClass}
 * @param primaryKey the primary key of the object to be found
 * @param lockMode the required <code>LockMode</code> for the object
 *
 * @return the persistent object with the given primary key
 * {@oclPost: result = self.lookup (instanceType, pk)}
 *
 * @exception PersistenceException if the object cannot be found or
 *                                 if an error occurred
 * {@oclPre:  -- the object exists
 *    not self.lookup (instanceType, pk).oclIsUndefined()}
 * {@oclPre:  -- the potential result can be locked as specified
 *    canLock (
 *        lookup (instanceType, pk),
 *        mode
 *      )}
 *
 * @see com.sdm.jeff.persistence.PersistentNotFoundException
 */
public abstract Persistent lookup(Class instanceType,
                                  Object[] primaryKey,
                                  LockMode lockMode)
  throws PersistenceException;
```

Quelltext 4.7: Javadoc-Kommentar mit den neuen *Tags*.

auswertet, das entsprechende Constraint nur einmal, da es `@oclRef-`*Tags* ignoriert.

Als ein nächster Schritt bietet es sich an, den *OCL-Injector* von Ralf Wiebicke ([Wie00]) so anzupassen, daß er diese *Tags* verarbeiten kann. Dabei gilt es zu entscheiden, ob die bisher vom *OCL-Injector* unterstützten *Standalone-Tags* beibehalten werden sollen oder nicht. Das einzige Argument dafür ist die weitere Unterstützung existierender Anwendungen. Ein entscheidendes Argument dagegen ist, daß der Programmierer ohne diese *Standalone-Tags* gezwungen wäre, auch zu jedem OCL-Constraint eine Klartext-Erläuterung zu schreiben, in die er das Constraint einbettet. Deshalb erscheint es sinnvoll, ausschließlich die hier vorgeschlagenen *Tags* zuzulassen.

4.5 Zusammenfassung

Das Verhalten des JEFF-Persistenz-Managers ist zu einem großen Teil mit OCL spezifizierbar. Einige Probleme werden durch diese Arbeit aufgezeigt. Die wichtigsten Lösungsvorschläge sind:

- Möglichkeiten zur Spezifikation von Exceptions (Abschnitt 4.4.1). Dies ermöglicht die Unterscheidung von „Desastern“ und „Illegalen Situationen“ auf der Entwurfsebene.
- Konzentration auf die Spezifikation der Schnittstelle. Die Spezifikation der Implementation ist weniger hilfreich, da ihre Komplexität der des Quelltexts entspricht.
- Einbettung der OCL-Constraints in die Javadoc, um die Verständlichkeit der Spezifikation zu erhöhen (Seite 43).

Kapitel 5

Kriterien für die Spezifikation und deren Evaluation

5.1 Einleitung

Für eine Aufwand-Nutzen-Abwägung für die Spezifikation mit OCL ist eine Bewertung der Spezifikation anhand möglichst objektiver Kriterien notwendig. Dazu müssen zunächst entsprechende Kriterien definiert werden, die beschreiben, was eine Spezifikation erbringen muß, um nützlich zu sein. Es ist dabei wichtig, sich stets bewußt zu machen, daß der Begriff des „Nutzens“ sehr stark subjektiver Natur ist. Dadurch haben sowohl die Kriterien als auch die Bewertung der Spezifikation stets einen subjektiven Charakter.

Dennoch wurde versucht, Kriterien möglichst scharf abzugrenzen. Dabei sollte die Bewertung anhand der entsprechenden Kriterien möglichst objektiv sein, d.h. nach Möglichkeit durch numerische Metriken oder zumindest durch Angabe konkreter Beispiele erfolgen. Die gefundenen Kriterien ergeben sich zum einen direkt aus den im Rahmen der Fallstudie aufgetretenen Problemen, zum anderen aus Gesprächen mit sd&m.

Im wesentlichen wird die Spezifikation anhand der folgenden drei Kriterien evaluiert:

1. Abdeckung

2. Komplexität

3. Nutzen

Die Kriterien werden im folgenden näher erläutert sowie jeweils auf die Spezifikation angewandt.

5.2 Abdeckung

Das Kriterium der Abdeckung beschreibt, welcher Anteil der natürlichsprachlichen Spezifikation durch die OCL-Spezifikation ausgedrückt werden kann. Es ist somit auch ein Maß für die Ausdrucksmächtigkeit der OCL.

Neben den Detailaussagen, die zur Ausdrucksmächtigkeit bereits in Abschnitt 4.3.1 getroffen wurden, ist der Versuch interessant, die Abdeckung durch die Spezifikation zahlenmäßig zu erfassen.

Dazu wurde die JEFF-Javadoc der Komponentenschnittstelle mit der OCL-Spezifikation gemischt, wie in Abschnitt „Literate Programming“ auf Seite 43 beschrieben. Anschließend wurde versucht, die Anzahl abgedeckter Einzelaussagen zu vergleichen. Dabei ergaben sich die in Tabelle 5.1 dargestellten Ergebnisse. Es bedeuten „Javadoc“ die Anzahl Elementaraussagen in der Javadoc; „OCL“ die Anzahl elementarer OCL-Aussagen, die Javadoc-Elementaraussagen zugeordnet werden konnten; „Anteil“ den Anteil in Prozent der Javadoc-Aussagen, der durch entsprechende OCL-Aussagen ausgedrückt wurde, sowie „Extra OCL“ die Anzahl der zusätzlichen OCL-Aussagen, die keiner Javadoc-Elementaraussage zugeordnet werden konnten. Diese stellen Unvollständigkeiten der Javadoc dar. Die untersuchte Fassung der Spezifikation beschränkt sich auf die Spezifikation der Schnittstelle und verwendet bereits die in Abschnitt 4.4.1 beschriebene Exception-Klausel.

Mit rund 62% Abdeckung ist die OCL-Spezifikation also anscheinend gar nicht so schlecht. Allerdings gibt es zwei Punkte, die bei der Interpretation der Daten zu beachten sind:

1. Im Einzelfall ergeben sich sehr geringe Abdeckungswerte. Diese begründen sich meist dadurch, daß die Spezifikation der Klasse Aussagen enthält, die nicht mit OCL darstellbar sind. Die genauen Gründe für

Klasse	**Javadoc**	**OCL**	**Anteil**	**Extra OCL**
`AccessMode`	7	2	28,6	0
`AttributeModel`	5	3	60,0	4
`AttributeTypeModel`	7	1	14,3	0
`LockMode`	12	9	75,0	0
`Multiplicity`	7	1	14,3	0
`PackageModel`	10	8	80,0	2
`PersistenceModelManager`	14	13	92,9	1
`Persistent`	nicht betrachtet, da zum größten Teil aus `Pool`			
`Pool`	49	32	65,3	7
`PropertyModel`	8	2	25,0	2
`RelationshipModel`	11	5	45,5	3
`TypeIdentifierModel`	1	1	100,0	0
`TypeModel`	17	14	82,4	4
Gesamt	148	91	61,49	23

Tabelle 5.1: Abdeckung der JEFF-Javadoc durch die OCL-Spezifikation.

jede Klasse sind in Tabelle 5.2 aufgeführt. Die Tabelle bezieht sich auf Tabelle 5.3, welche die verschiedenen Gründe kategorisiert.

Würden sämtliche `toString`-Methoden mitspezifiziert, so erhöht sich die Abdeckung auf 63, 51%. Da die `toString`-Methoden in JEFF keine wesentliche Semantik haben (also z.B. nicht als Serialisierungshilfe verwendet werden), wurden sie von der Spezifikation ausgeschlossen.

Klasse	**Bemerkungen**
`AccessMode`	*1*
`AttributeModel`	`getInitValue`, `belongsToPrimaryKey`: *2*
`AttributeTypeModel`	alle Operationen: *2*
`LockMode`	*1*
`Multiplicity`	*3*
`PackageModel`	`getTypeModels`, `getAttributeTypeModels`: *2*
`PersistenceModelManager`	`getPackageModels`: *2*
`Pool`	`getTypeModel (Class)`: *2*
`PropertyModel`	`getValue`, `setValue`: *4*
`RelationshipModel`	`getToTypeModel`, `getPathName`, `getMultiplicity`: *2*
`TypeModel`	`isAbstract`, `isAccessPermitted`, `getTypeIdentifierModels`: *2*

Tabelle 5.2: Spezifikationsbesonderheiten. Die *hervorgehobenen* Zahlen beziehen sich auf Tabelle 5.3

2. Die Abgrenzung von Elementaraussagen ist schwierig. Es gelang nicht, dafür einen formalen Ansatz zu finden. Deshalb sind die ermittelten Werte insbesondere von Spezifikationen verschiedener Klassen nicht zuverlässig miteinander vergleichbar. Es wurde aber versucht, die Unterteilung in Teilaussagen auf möglichst gleiche Art für jede Klasse durchzuführen.

Dennoch scheint die Aussage gerechtfertigt, daß die Abdeckung relativ hoch ist (mehr als 50%).

1	`toString`: Spezifikation möglich, jedoch nicht explizit ausgeführt
2	Informationsquelle: Operation, die überschrieben werden muß, um Informationen zur Verfügung zu stellen (z.B. Prädikatoperation). Diese sind schwer formalisierbar, außer in Bezug auf Konsistenzbedingungen für die gelieferten Informationen.
3	Implementationsaussagen nicht spezifizierbar.
4	An anderer Stelle spezifiziert.

Tabelle 5.3: Kategorien von Spezifikationsbesonderheiten.

Der vollständige gemischte Text aus Javadoc und OCL findet sich auf der beiliegenden CD-ROM (s. Anhang B).

5.3 Komplexität

Das Kriterium der Komplexität versucht zu erfassen, wie schwierig es für einen menschlichen Schreiber bzw. Leser ist, die Spezifikation zu erstellen bzw. zu verstehen.

Wie (beispielsweise) in [HS96] sehr anschaulich beschrieben, handelt es sich dabei selbst um ein sehr komplexes Gebiet. Zunächst muß der Begriff der Komplexität genauer definiert werden. *Komplexität* ist stets ein *externes Attribut* von Software, d.h. eines, das menschliche Eigenschaften mit einbezieht und sich deshalb nur schwer (oder gar nicht) direkt und objektiv messen läßt. Der Begriff Komplexität läßt sich unterteilen in die drei Kategorien

- Rechenaufwand (*Computational Complexity*), relevant vor allem für Hardware-Lösungen,
- Darstellungskomplexität (*Representational Complexity*), durch das gewählte Darstellungsmittel (textuell, grafisch, konkrete Sprache ...)

erzeugte Komplexität und

- Psychologische Komplexität (*Psychological Complexity*), diese zerfällt in
  - Funktionelle Komplexität (*Functional Complexity*), problemimmanente Komplexität,
  - Eigenschaften des Programmierers (*Programmer Characteristics*), diese sind schwer objektiv zu messen, haben aber Einfluß auf die Empfindung von Komplexität, und
  - Strukturelle Komplexität (*Structural Complexity*).

Software-Komplexitäts-Metriken ([Zus94], [HS96], [Hal77] ...) messen strukturelle Eigenschaften von Artefakten. Anschließend wird versucht, daraus Aussagen über die strukturelle Komplexität der Artefakte zu gewinnen oder Prognosen über Wartbarkeit, Fehleranfälligkeit etc. zu erstellen. Wie in [HS96] dargestellt, muß man dabei sehr sorgfältig vorgehen. Der Nachweis, daß die aus den gemessenen Werten gezogenen Schlüsse zulässig sind, ist sehr schwierig. Unter Umständen ist dieser Nachweis auch sehr stark von den genauen Bedingungen abhängig, so daß die Validität von Schlüssen nicht damit begründet werden kann, daß analoge Schlüsse aus ähnlichen Werten schon in anderen Projekten (also unter potentiell anderen Bedingungen) als zulässig nachgewiesen wurden.

Trotz der geschilderten Schwierigkeiten wird im folgenden versucht, die Komplexität der erstellten Spezifikation zu bewerten. Dazu werden zunächst einige zu berücksichtigende Punkte beschrieben. Anschließend wird über die Anwendung einer bereits existierenden Metrik für strukturelle Komplexität — die Halstead-Schwierigkeits-Metrik — berichtet.

5.3.1 Eine Komplexitäts-Metrik für OCL

Zu den Faktoren, die die strukturelle Komplexität einer OCL-Spezifikation steigern, gehören intuitiv:

- die Anzahl der Constraints.

- die Anzahl elementarer Teilformeln pro Constraint.

  Elementare Teilformeln sind Teilausdrücke, die keine logischen Junktoren (`and`, `or`, `if-then-else-endif` ...) enthalten.

- die Verschachtelungstiefe von `if-then-else-endif`-Verzweigungen pro Constraint.

- die Anzahl verschachtelter Iteratoren.

- das Verhältnis von pre-/post-condition-Constraints zu Invarianten.

Für die ersten drei Kriterien wurden mit dem im Anhang erläuterten `MetricsMeasurer` (Anhang A) entsprechende Werte ermittelt. Dabei handelt es sich um ein kleines Java-Tool aus OCL-Parser und generischem Baum-Traversierer, dem konkrete Traversierer übergeben werden, die dann entsprechende Metriken ermitteln. Die ermittelten Werte sind in Tabelle 5.4 und Tabelle 5.5 zusammengestellt.

Klasse	**I**	**II**	**III**
`AccessMode`	1	1,0	0
`AttributeModel`	8	1,38	0
`LockMode`	4	7,0	6
`PackageModel`	14	1,57	0
`PersistenceModelManager`	21	1,10	0
`Persistent`	17	1,53	0
`Pool`	73	1,70	4
`PropertyModel`	6	1,67	0
`RelationshipModel`	12	1,83	0
`TransactionHandle`	1	2,0	0
`TypeIdentifierModel`	2	2,5	1
`TypeModel`	22	2,05	2

Tabelle 5.4: Ermittelte Komplexitätsmaße für die Komponentenschnittstelle. Legende: **(I)** Anzahl Constraints, **(II)** Mittlere Anzahl Teilformeln pro Constraint und **(III)** Maximale Verschachtelungstiefe von `if-then-else-endif`-Konstrukten.

Klasse	**I**	**II**	**III**
`AbstractPersistent`	23	1,09	0
`AttributeController`	22	1,36	0
`LocalLockManager`	50	1,28	1
`NoCache`	7	1,0	0
`ObjectReference`	36	1,94	0
`PersistenceConnectionManager`	67	2,99	2
`PersistenceManager`	113	1,38	1
`PersistenceMethodCache`	18	2,5	0
`PersistenceResourceManager`	34	1,68	0
`PersistentCache`	11	1,64	0
`PersistentState`	3	2,33	1
`Reference`	8	1,0	0
`Relation`	35	3,17	3
`RelationReference`	27	2,07	0
`RelationshipController`	38	1,95	2
`TransactionCache`	12	1,33	0
`TransactionState`	118	2,69	2
`TypeRegistry`	30	2,07	1

Tabelle 5.5: Ermittelte Komplexitätsmaße für die Komponentenimplementation.
Legende: **(I)** Anzahl Constraints, **(II)** Mittlere Anzahl Teilformeln pro Constraint und **(III)** Maximale Verschachtelungstiefe von `if-then-else-endif`-Konstrukten.

Sieht man sich diese Werte genauer an, fallen einige Besonderheiten auf:

- Die Spezifikation der Klasse `LockMode`, die sehr einfach ist, hat im Mittel 7 elementare Teilformeln pro Constraint und bis zu 6 Ebenen tief verschachtelte `if-then-else-endif`-Konstrukte. Es werden alle möglichen Fälle mit Hilfe von `if-then-else-endif`-Konstrukten beschrieben. Die einzelnen Fälle selbst sind sehr einfach, dennoch erhöhen sich natürlich die für die Metriken gemessenen Werte.
- Die als sehr komplex empfundene Spezifikation für `Pool` hat im Mittel nur 1,7 elementare Teilformeln pro Constraint. Die Komplexität liegt hierbei wahrscheinlich im großen Umfang von 73 Constraints.

Diese Beispiele zeigen, daß die vorgeschlagenen Metriken allein noch keinen Schluß auf die strukturelle Komplexität einer OCL-Spezifikation zulassen.

5.3.2 Die Halstead-Metrik

Eine bekannte Metrik für die Komplexität von Programmen ist die Schwierigkeitsmetrik von Halstead ([Hal77]).

Für ein Programm in einer beliebigen Programmiersprache definiert Halstead die folgenden Parameter:

η_1 Anzahl verschiedener Operatoren (*Unique Operator Count*)

η_2 Anzahl verschiedener Operanden (*Unique Operand Count*)

N_1 Gesamtzahl Operatoren (*Total Operator Count*)

N_2 Gesamtzahl Operanden (*Total Operand Count*)

Operanden sind Konstanten und Variablen. Die Definition von Operatoren entspricht der intuitiven Auffassung von Operatoren, wobei zusätzlich alle Kontrollstrukturen (`if-then-else-endif`, `let-:-=-in-` ...) sowie Operationsaufrufe als Operatoren gezählt werden. Alle Dinge, die für den Inhalt des Programms nicht relevant sind (Kommentare, ungenutzte Labels etc.), werden nicht gezählt.

Dann ist mit

$$\begin{aligned} \eta &= \eta_1 + \eta_2 \\ N &= N_1 + N_2 \end{aligned}$$

der Umfang des Vokabulars (η) und die Länge der Implementation (N) gegeben.

Basierend auf diesen Werten definiert Halstead verschiedene Metriken. Unter anderem:

$$\begin{aligned} \hat{L} &= \frac{2}{\eta_1}\frac{\eta_2}{N_2} \\ D &= \frac{1}{L} \\ &\approx \frac{1}{\hat{L}} \\ &= \frac{\eta_1 N_2}{2\eta_2} \end{aligned}$$

mit

L Programmlevel (*Progam Level*) — Ein Wert, der das Abstraktionslevel der verwendeten Programmiersprache beschreibt. $L = 1$, für die abstraktest mögliche Fassung eines Algorithmus, d.h. in einer Sprache, die den Algorithmus bereits als Operation zur Verfügung stellt, die nur noch aufgerufen werden muß. Es gilt $0 < L \leq 1$.

$\hat{L}$ angenähertes Programmlevel — eine Annäherung an L, basierend auf leichter meßbaren Werten: $\hat{L} \approx L$.

D Schwierigkeit (*Difficulty*) — Ein Wert, der mißt, wie schwer es einem Experten fällt, ein Programm zu verstehen. Die Berechnungsvorschrift geht davon aus, daß der Experte die Sprache fließend beherrscht, d.h., daß er alle vorhandenen Operatoren und deren Semantik kennt. D stellt somit eine untere Schranke für die Schwierigkeit dar, einem Laien fällt das Verständnis eines Programms entsprechend noch schwerer.

Die Schwierigkeitswerte (D) wurden mit Hilfe des in Anhang A beschriebenen `MetricsMeasurer` für die erstellte Spezifikation ermittelt. Dabei ergaben sich die in Tabelle 5.6 und Tabelle 5.7 dargestellten Werte. Im Mittel

Klasse	*D*
`AccessMode`	3,5
`AttributeModel`	13,54
`LockMode`	10,18
`PackageModel`	21,27
`PersistenceModelManager`	39,75
`Persistent`	31,82
`Pool`	96,44
`PropertyModel`	14,68
`RelationshipModel`	28,15
`TransactionHandle`	6,5
`TypeIdentifierModel`	12,14
`TypeModel`	54,0

Tabelle 5.6: Halstead-Lesbarkeits-Metrik für die spezifizierten Klassen der Komponentenschnittstelle.

ergibt sich für die Komponentenschnittstelle (12 Klassen) ein Wert von ca. 27,66. Für die Komponentenimplementation (22 Dateien) beträgt der Mittelwert rund 41,8. Die gesamte Komponente hat eine mittlere Schwierigkeit D von 36,81. Dies unterstützt auch die Hypothese, daß die Implementationsspezifikation komplexer ist als die der Schnittstelle.

5.4 Nutzen

Das zentrale Kriterium für die Praxistauglichkeit einer Spezifikation(-ssprache) ist der Nutzen für die Entwickler. Dabei sind vor allem die folgenden zwei Fragen zu beantworten:

1. Entsteht durch die Spezifikation eine höhere Qualität der fertigen Software?

2. Wird der Software-Entwicklungsprozeß durch die Spezifikation effizienter?

Klasse	D
`AbstractPersistent`	22,22
`AttributeController`	30,18
`LocalLockManager`	47,35
`NoCache`	8,5
`ObjectReference`	36,81
`PersistenceConnectionManager`	119,71
`PersistenceManager`	119,75
`PersistenceMethodCache`	23,33
`PersistenceResourceManager`	47,14
`PersistentCache`	18,0
`PersistentState`	13,33
`Reference`	10,0
`Relation`	81,22
`RelationReference`	47,67
`RelationshipController`	59,38
`TransactionCache`	19,64
`TransactionState`	142,77
`TypeRegistry`	70,62

Tabelle 5.7: Halstead-Lesbarkeits-Metrik für die spezifizierten Klassen der Komponentenimplementation.

Kann man auch nur eine dieser Fragen mit „Ja“ beantworten, so kann man die Spezifikation als nützlich bezeichnen. Anhand einiger Beispiele wird im folgenden dargestellt, wie diese Fragen für die erstellte Spezifikation beantwortet werden könnten.

5.4.1 Präzision durch Formalisierung

Durch die Anwendung der formalen Sprache kann ein Gewinn an Präzision entstehen. Implizite Annahmen müssen in der formalen Spezifikation explizit gemacht werden. In der natürlichsprachlichen Spezifikation können diese leicht „versteckt“ werden. Höhere Präzision dient sowohl dem Entwickler als auch dem Nutzer einer Komponente. Für den Entwickler bringt sie eine Erhöhung an Qualität, da zu jeder Zeit alle Randbedingungen für die Implementation präzise bekannt sind und potentielle Probleme zeitig aufgedeckt werden können. Der Nutzer gewinnt durch die vollständigere Spezifikation an Effizienz bei der Verwendung der Komponente durch besseres Verständnis der erwarteten Bedingungen.

Als Beispiel sei die Vorbedingung für `RelationshipModel::setValue` in Quelltext 5.1 dargestellt. Diese beschreibt eine Bedingung, die in der natürlichsprachlichen Spezifikation nur implizit auftaucht.

```
context RelationshipModel::setValue (p     : Persistent,
                                     value : java::lang::Object)
pre: -- the specified value must be of a class conforming to the type
     -- of the relationship
    isSingleValued() implies
      p.getPool()
       .getInstanceType (getToTypeModel())
       .isAssignableFrom (value.getClass())
  and
    isMultiValued() implies
      (
        value.oclIsKindOf (Set(Persistent)) and
        value.oclAsType (Set(Persistent))
             ->forAll (p1 |
                  p.getPool()
                   .getInstanceType (getToTypeModel())
                   .isAssignableFrom (p1.getClass())
                )
      )
```

Quelltext 5.1: Vorbedingung für `RelationshipModel::setValue()`: Diese Vorbedingung ist nur implizit in der natürlichsprachlichen Spezifikation enthalten.

5.4.2 Einbettung in die Javadoc

Durch die Einbettung der OCL-Constraints in die Javadoc können verschiedene Vorteile erreicht werden. Zum einen kann die Konsistenz zwischen Dokumentation und Code und damit die Qualität des fertigen Produkts verbessert werden, indem Werkzeuge verwendet werden, um aus den OCL-Constraints Java-Quelltext zu generieren (z.B. [Wie00]). Ein Beispiel ist die in Quelltext 5.2 gezeigte Vorbedingung für `Pool::make`. Diese wird in der Javadoc implizit erwähnt, erscheint jedoch nicht als Assertion im Quelltext. Die Generierung der Assertions hilft auch, Fehler bei der Übertragung zu vermeiden (zumal, wenn es zu einem Paradigmenwechsel kommt, s. Seite 26).

```
context Pool::make (type : java::lang::Class,
                    pk   : Sequence(java::lang::Object)) : Persistent
pre: -- the specified type is a class that implements Persistent
  clsPersistent.isAssignableFrom (type) and
  not type.isInterface()
```

Quelltext 5.2: Diese Vorbedingung für `Pool::make()` erscheint in der Javadoc, jedoch nicht als Assertion im Quelltext.

Auf der anderen Seite entsteht durch die Möglichkeit der Generierung von Assertions auch ein Gewinn an Prozeßeffizienz, da diese Constraints nun nur noch einmal (zur Entwurfszeit) niedergeschrieben werden müssen. Anschließend werden sie automatisch gepflegt.

5.4.3 Effizienzverlust durch Überspezifikation

Die Spezifikation mit OCL bietet natürlich nicht nur Vorteile. Bei der Spezifikation der Implementation kommt es beispielsweise fast zu einer Verdopplung des Arbeitsaufwands, da die entsprechenden Algorithmen zweimal aufgeschrieben werden müssen. Erschwerend kommt noch der auf Seite 26 beschriebene Paradigmenwechsel hinzu.

Deshalb ist von einer vollständigen Spezifikation der Implementation abzuraten. Die einzig sinnvolle Spezifikationsaufgabe im Zusammenhang mit der Implementation wäre die Spezifikation neuer öffentlicher Operationen. Allerdings sollte auch hier weitestgehend von der konkreten Implementation abstrahiert werden und nur die allgemeine Funktion — am besten auf der Ebene der Schnittstelle — spezifiziert werden.

5.4.4 Generierung von Testfällen

Potentiell wäre es möglich, aus den OCL-Constraints zusammen mit der restlichen UML-Spezifikation Testfälle und Testdatensätze zu generieren. Diese könnten für Funktionstests im Black-Box-Verfahren eingesetzt werden. Beispielsweise könnten für eine Vorbedingung wie

```
context Number::div (a : Integer, b: Integer) : Real
pre: b <> 0
```

die folgenden Äquivalenzklassen automatisch gefunden werden:

- `b` ist negativ. Zufällig gewählter Repräsentant: $\texttt{b} = -367$.
- `b` ist positiv. Zufällig gewählter Repräsentant: $\texttt{b} = +67$.

Zusätzlich würden die beiden Grenzfälle $\texttt{b} = \pm 1$ sowie der unzulässige Wert $\texttt{b} = 0$ als Testfälle vorgeschlagen.

Eine solche automatische Generierung von Testfällen würde einen erheblichen Effizienzgewinn bedeuten. Außerdem könnte es auch einen Qualitätszuwachs bringen, da eine vollständige Testabdeckung gewährleistet werden könnte.

Auf dem Gebiet Testfallgenerierung wird im Moment von verschiedenen Seiten geforscht. [OXL99] gibt einen Überblick über den Stand der Forschung. Außerdem werden Kriterien beschrieben, nach denen Testfälle aus einer Spezifikation (in diesem Fall in Form eines Automaten) abgeleitet werden können. Speziell für OCL-Spezifikationen konnte keine Literatur gefunden werden.

5.4.5 Werkzeugunterstützung

Die dargestellten Verbesserungen der Qualität und der Prozeßeffizienz lassen sich nur in die Realität umsetzen, wenn geeignete Werkzeuge zur Verfügung stehen. Im Moment gibt es die im folgenden aufgezeigten Werkzeuge (die Liste erhebt keinen Anspruch auf Vollständigkeit):

- Dresden OCL Toolkit ([Dre]): Das Toolkit enthält einen OCL-Parser, Syntax- und Typ-Checker, Codegeneratoren für Java und SQL sowie einen Injector. Dieser ermöglicht es, Code, der OCL-Constraints zur

Laufzeit überprüft, direkt in die zu überprüfenden Klassen einzubinden. Außerdem existiert eine Integration in ArgoUML ([Arg]) und Poseidon. Das Toolkit unterstützt im wesentlichen OCL 1.1.

- USE-Tool der Universität Bremen ([USE]): Dieses Tool erlaubt das interaktive Erzeugen von Instanzenmengen für ein gegebenes Klassendiagramm und die Simulation des Ablaufs im Modell definierter Operationen. Dabei werden im Modell definierte OCL-Constraints stets auf ihre Gültigkeit überprüft.

- BoldSofts ([Bol]) Bold!-Architektur verwendet OCL, um darzustellende Mengen zu beschreiben. Mit *ModelRun* existiert ein ähnliches Werkzeug wie USE, das die Animation von Modellen erlaubt.

- Der ObjeXion *Model Prototyper* ([Obj]) ist ein ähnliches Werkzeug wie das USE-Tool der Uni Bremen. Er erlaubt, interaktiv Instanzenmengen von UML-Modellen zu erzeugen. Dabei beachtet er in UML formulierte Einschränkungen (Assoziationen, Multiplizitäten, Generalisierungen ...), jedoch keine OCL-Constraints. Die OCL wird von ObjeXion als Abfrage- und Evaluationssprache während der Simulation sowie zur Definition von {query}-Operationen verwendet. ObjeXion basiert auf der OCL Version 1.3.

- Cybernetic Intelligence OCL-Compiler ([Cyb]): Die offizielle Zielstellung ist ein OCL-Compiler, der die Konsistenz von OCL-Spezifikationen überprüfen kann. Zur Zeit (Version 1.5) existiert nur ein Syntax- und Typ-Checker.

- Klasse Objecten bietet einen kostenlosen Syntax-Checker für OCL an: [Kla].

Diese Werkzeuge sind oft nur prototypisch angelegt, unterstützen nur ältere Versionen der OCL oder bieten nur begrenzte Funktionen. Hier besteht noch Entwicklungsbedarf.

5.5 Zusammenfassung

Die OCL bietet ausreichende Ausdrucksmittel, um einen wesentlichen Teil von Software-Systemen zu spezifizieren. In bestimmten Einzelbereichen müssen jedoch noch Ausdrucksmöglichkeiten geschaffen werden (s. Kapitel 4).

Die Komplexität einer OCL-Spezifikation kann leicht sehr hoch werden. Dabei ist ein großer Unterschied zwischen der Spezifikation von Schnittstelle und Implementation zu beobachten. Die Schnittstellenspezifikation ist deutlich weniger komplex als die Spezifikation der Implementation. Es scheint insgesamt, daß das Spezifizieren von Schnittstellen sinnvoller ist als eine Implementationsspezifikation.

Richtig angewendet kann die OCL die Qualität des Codes erhöhen und die Effizienz des Entwicklungsprozesses verbessern. Durch Formalisierung wird höhere Präzision und damit Qualität, durch die Möglichkeit der Generierung verschiedener Artefakte (Dokumentation, Assertion-Code, Testfälle) bessere Effizienz und Konsistenz erreicht. Dies gilt bereits bei teilweiser Spezifikation der Komponente. Bis zu einer gewissen Grenze (s. Abschnitt 5.4.3) gilt natürlich, daß eine umfassendere Spezifikation auch einen höheren Qualitäts- und Effizienzgewinn ergibt.

Ein Problem ist die bisher mangelnde Werkzeugunterstützung. Hier werden noch Werkzeuge benötigt, die die Generierung von Code, Dokumentation und Testfällen sowie eine Konsistenzprüfung einer OCL-Spezifikation ermöglichen.

Kapitel 6

Zusammenfassung und Ausblick

OCL ist eine formale Sprache mit einer Syntax, die sich an die objektorientierter Programmiersprachen anlehnt. Trotz einiger kleinerer Probleme in der Ausdrucksmächtigkeit scheint die Sprache gut geeignet, um auch komplexe Systeme zu spezifizieren.

Bereits eine teilweise Spezifikation (beispielsweise nur der zulässigen Wertebereiche von Parametern) kann Qualitäts- und Prozeßeffizienzgewinne erbringen. Durch die Formalisierung werden implizite Annahmen aufgedeckt. Bei geeigneter Werkzeugunterstützung kann die Prozeßeffizienz durch Generierung von Artefakten gesteigert werden. Besonders interessant sind hier:

- Assertions, d.h. generierte Code-Bestandteile, die die Einhaltung von Invarianten und Vor-/Nachbedingungen zur Laufzeit überprüfen (s. [Wie00]).
- Testfälle, d.h. Nutzung der OCL-Spezifikation, um automatisch Testklassen zu identifizieren und entsprechende Testdatensätze zu generieren.
- Mischung von formaler und natürlichsprachlicher Spezifikation, beispielsweise für die Zielsprache Java durch Einbettung der OCL-Constraints in die Javadoc (s. Abschnitt „Literate Programming“).

Ist die erstellte Spezifikation zu detailliert, so kann sich der positive Effekt leicht ins Gegenteil umkehren. Eine detaillierte Spezifikation von Implementationen ist daher nicht sinnvoll. Dagegen ist der Nutzen einer vollständigen Spezifikation von Schnittstellen deutlich erkennbar.

Eine wichtige Voraussetzung für den Nutzen einer OCL-Spezifikation ist eine gute Werkzeugunterstützung. Diese ist im Moment eher schlecht ausgeprägt. Daher wäre es ein interessanter nächster Schritt, Werkzeuge zur automatischen Generierung von Testfällen und zur Integration der OCL-Constraints in die Javadoc-Dokumentation zu entwickeln.

Anhang A

`MetricsMeasurer`

Zur Ermittlung der verwendeten Metriken wurde ein kleines Java-Werkzeug entwickelt. Dieses besteht aus:

1. einem OCL-Parser, der die OCL-Spezifikation in eine rechnerinterne Darstellung als abstrakter Syntaxbaum übersetzt,
2. einem abstrakten `ASTWalker`, der die Baumtraversierung implementiert und von konkreten Klassen zur Metrikermittlung spezialisiert wird sowie
3. einem Hauptprogramm, das als Parameter die zu untersuchenden Dateien sowie die Namen der zu verwendenden konkreten Klassen zur Metrikermittlung übernimmt. Dieses parst jede der übergebenen Klassen und übergibt anschließend den erzeugten abstrakten Syntaxbaum an die metrikermittelnden Klassen. Abschließend gibt es einen Bericht aus.

Für den Parser wurde eine Entscheidung gegen die Verwendung des OCL-Parsers aus dem *Dresden OCL Toolkit* getroffen. Stattdessen wurde eine *public domain antlr*-Grammatik von Frederic Fondement verwendet und um die in dieser Arbeit eingeführten Konstrukte erweitert. Dies war nötig, da der Parser aus dem *Dresden OCL Toolkit* auf die OCL 1.1 beschränkt und sehr schwer anpaßbar ist.

Basierend auf dem `ASTWalker` wurden verschiedene Klassen zur Metrikermittlung implementiert:

- `ConstraintCounter`: zählt die individuellen Constraints in einer Datei.
- `ElementaryExpressionsCounter`: Bestimmt die mittlere Anzahl elementarer Teilformeln pro Constraint.
- `HalsteadMetricsCalculator`: Ermittelt die Halstead-Schwierigkeitsmetrik (s. Abschnitt 5.3.2).

  Typnamen, Variablen und direkte Werte (Zahlen, Zeichenketten, `true`, `false` ...) werden dabei als Operanden klassifiziert. Zusätzlich zu den Standard-OCL-Operatoren (`+`, `-`, `and`, `or` ...) werden auch Operationsaufrufe (sowohl Standardoperationen als auch Operationen aus dem UML-Modell) als Operatoren gezählt. Klammerpaare („(...)“, „[...]“ ...) sowie mehrteilige Operatoren (`if-then-else-endif`, `package-endpackage` ...) gelten als ein Operator.
- `MaxIfThenElseNestingCalculator`: Bestimmt die maximale Tiefe verschachtelter `if-then-else-endif`-Konstrukte.

Die Quelltexte des `MetricsMeasurer` befinden sich auf der beiliegenden CD-ROM.

Anhang B

Inhaltsübersicht der beigefügten CD-ROM

Zur Diplomarbeit gehört eine CD-ROM. Diese enthält die folgenden Verzeichnisse und Dateien:

\

MetricsMeasurer\ Quellen des `MetricsMeasurer`s (s. Anhang A).

src\ Die Quellen sowie die kompilierte Fassung des `MetricsMeasurer`.

build.xml ant-Buildscript.

compute_metrics Beispielskript, daß die Metriken für eine Spezifikation im Verzeichnis Constraints\ ermittelt.

halstead_averages.pl Perl-Skript, das aus der ermittelten Metrik den Mittelwert der Halstead-Schwierigkeitsmetrik herausfiltert.

Papers\ Artikel, die als Quellen für diese Arbeit dienten, soweit sie in elektronischer Form vorlagen.

Presentations\ PowerPoint-Folien der Vorträge zur Diplomarbeit.

Rose Models\ Rational Rose Dateien der in dieser Arbeit dargestellten UML-Modelle.

Specification\ Verschiedene Versionen der erstellten Spezifikation. Jede Version enthält eine „.OCL“-Datei pro spezifizierter Klasse sowie zusätzlich jeweils eine „.Metrics“-Datei, die die Ergebnisse des `MetricsMeasurer`s enthält.

Complete\ Komplette Spezifikation der Komponentenschnittstelle sowie der Implementation des JEFF-Persistenz-Managers. Die Spezifikation verwendet die Spezifikationsmöglichkeiten für Exceptions (s. Abschnitt 4.4.1).

Only Parameter Ranges\ Spezifikationsversion, die nur die Wertebereiche der Parameter spezifiziert (s. Seite 37).

Subject Oriented\ In Subjekte eingeteilte Version der Spezifikation (s. Seite 38).

Synopsis\ Mischung aus Javadoc und OCL-Spezifikation (s. Abschnitt „Literate Programming“). Da ein entsprechendes *Taglet* noch nicht entwickelt wurde, liegen die Dokumente nur als Word-Dateien vor.

Thesis\ Die LaTeX 2_ε-Quellen sowie die PDF-Fassung dieser Arbeit.

index.html Eine HTML-Datei zur einfachen Navigation in der CD-ROM.

Literaturverzeichnis

[Arg] *ArgoUML project homepage.* World Wide Web. – http://argo.tigris.org/

[BHTW99] Bidoit, Michel ; Hennicker, Rolf ; Tort, Francoise ; Wirsing, Martin: Correct Realizations of Interface Constraints with OCL. In: [FR99], S. 399–415

[Bol] BoldSoft: World Wide Web. – http://www.boldsoft.com

[CD94] Cook, S. ; Daniels, J.: *Designing Object Systems: Object-Oriented Modeling with Syntropy.* Prentice Hall, 1994

[CD01] Cheesman, John ; Daniels, John: *UML Components: A Simple Process for Specifying Component-Based Software.* Addison Wesley Longman, Inc., 2001. – ISBN 0–20–170851–5

[CHOT99] Clarke, Siobhán ; Harrison, William ; Ossher, Harold ; Tarr, Peri: Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code. In: *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'99).* Denver, CO, USA, November 1999, S. 325–339

[CKM+02] Cook, Steve ; Kleppe, Anneke ; Mitchell, Richard ; Rumpe, Bernhard ; Warmer, Jos ; Wills, Alan: The Amsterdam Manifesto on OCL. In: [CW02], S. 115–149

[CT01] Conrad, Stefan ; Turowski, Klaus: Temporal OCL: Meeting Specification Demands for Business Components. In: Siau, K.

(Hrsg.) ; Halpin, T. (Hrsg.): *Unified Modeling Language: Systems Analysis, Design and Development Issues.* IDEA Group Publishing, 2001, S. 151–165

[CW02] Clark, Tony (Hrsg.) ; Warmer, Jos (Hrsg.): *Object modeling with the OCL: the rationale behind the object constraint language.* Springer Verlag, 2002 (Lecture notes in computer science 2263)

[Cyb] World Wide Web. – http://www.cybernetic.org

[DHL01] Demuth, Birgit ; Hussmann, Heinrich ; Löcher, Sten: OCL as a Specification Language for Business Rules in Data Base Applications. In: [GK01],

[Dre] *Dresden OCL Toolkit Project Homepage.* World Wide Web. – http://dresden-ocl.sourceforge.net/index.html

[DW98] D'Souza, Desmond ; Wills, A.: *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison Wesley Verlag, 1998

[Fow97] Fowler, Martin: *Analysis Patterns: Reusable Object Models.* Addison Wesley Verlag, 1997. – ISBN 0–201–89542–0

[FR99] France, Robert B. (Hrsg.) ; Rumpe, Bernhard (Hrsg.): *The Unified Modeling Language – Beyond the Standard, Proc. 2nd. Int. Conf., (UML'99).* Fort Collins, CO, USA : Springer Verlag, Oktober 1999 (Lecture notes in computer science 1723)

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Verlag, 1994 (Addison Wesley Professional Computing). – ISBN 0–201–63361–2

[GK01] Gogolla, Martin (Hrsg.) ; Kobryn, Cris (Hrsg.): *UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts and Tools, Fourth International Conference.* Toronto, Canada

: Springer Verlag, Oktober 2001 (Lecture notes in computer science 2185)

[Hal77] Halstead, Maurice H.: *Elements of Software Science*. New York, NY, USA : Elsevier North-Holland, Inc., 1977

[HK99] Hitz, Martin ; Kappel, Gerti ; Hitz, Martin (Hrsg.) ; Kappel, Gerti (Hrsg.): *UML@Work*. dpunkt-Verlag, 1999

[HO93] Harrison, William ; Ossher, Harold: Subject-Oriented Programming (A Critique of Pure Objects). In: *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA'93)*. Washington, D.C., United States, 1993, S. 411–428

[HS96] Henderson-Sellers, Brian: *Object-Oriented Metrics: Measures of Complexity*. NJ, USA : Prentice Hall PTR, 1996 (The Object-Oriented Series). – ISBN 0–13–239872–9

[Jav] *SUN^{tm} Javadoc^{tm} Documentation*. World Wide Web. – http://java.sun.com/j2se/1.4/docs/tooldocs/javadoc/index.html

[JSGB00] Joy, Bill ; Steele, Guy ; Gosling, James ; Bracha, Gilad: *The Java^{tm} Language Specification, Second Edition*. Addison Wesley Verlag, Juni 2000. – ISBN 0–201–31008–2

[Kla] World Wide Web. – http://www.klasse.nl/ocl/ocl-checker-text.html

[KLM+97] Kiczales, Gregor ; Lamping, John ; Mendhekar, Anurag ; Maeda, Chris ; Lopes, Cristina V. ; Loingtier, Jean-Marc ; Irwin, John: Aspect-Oriented Programming. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97), Finland*, Springer Verlag, Juni 1997 (Lecture notes in computer science 1241)

[Knu01] Knuth, Donald E.: *Literate Programming*. CSLI Publications, Januar 2001. – ISBN 0–937–07381–4

[KW02] Kleppe, Anneke ; Warmer, Jos: The Semantics of the OCL Action Clause. In: [CW02], S. 213–227

[Löc01] Löcher, Sten: *UML/OCL für die Integritätssicherung in Datenbankanwendungen*, Technische Universität Dresden, Diplomarbeit, Juli 2001

[MS01] Marco Schmickler, sd&m AG: *JEFF-Framework: White Paper*. November 2001. – Unpublished Version 1.0

[MU01] Markus Uhlendahl, sd&m AG: *Design Rationale: [JEFF] Framework Architecture*. Dezember 2001. – Unpublished Version 3.0

[Obj] *Objexion Model Prototyper Homepage*. World Wide Web. – http://www.objexion.com/documentation

[OJ01] Oliver Juwig, sd&m AG: *Design Rationale: [JEFF] Server-Core – Persistence-Manager*. Dezember 2001. – Unpublished Version 3.0

[OT99] Ossher, Harold ; Tarr, Peri: Using Subject-Oriented Programming to Overcome Common Problems in Object-Oriented Software Development/Evolution. In: *Proceedings of the 1999 international conference on Software engineering*. Los Angeles, CA, USA, 1999, S. 687–688

[OXL99] Offutt, Jeff ; Xiong, Yiwei ; Liu, Shaoying: Criteria for Generating Specification-based Tests. In: *Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99), Las Vegas, NV*, 1999

[SF99] Soundarajan, Neelam ; Fridella, Stephen: Modeling Exceptional Behavior. In: [FR99], S. 691–705

[SS01] Sendall, Shane ; Strohmeier, Alfred: Specifying Concurrent System Behavior and Timing Constraints using OCL and UML. In: [GK01], S. 391–405

[SS02] Sendall, Shane ; Strohmeier, Alfred: Using OCL and UML to Specify System Behavior. In: [CW02], S. 250–280

[UML01] Object Management Group (OMG): *OMG Unified Modeling Language Specification Version 1.4*. September 2001. – OMG document formal/01-09-67

[USE] *USE Tool Project Page*. World Wide Web. – http://www.db.informatik.uni-bremen.de/projects/USE/

[W+01] Warmer, Jos [u. a.]. *Response to the UML 2.0 OCL RfP (ad/2000-09-03), Initial Submission, OMG Document ad/2001-08-01*. August 2001

[W+02a] Warmer, Jos [u. a.]. *Response to the UML 2.0 OCL RfP (ad/2000-09-03), Revised Submission, Version 1.4*. April 2002

[W+02b] Warmer, Jos [u. a.]. *Response to the UML 2.0 OCL RfP (ad/2000-09-03), Revised Submission, Version 1.5, OMG Document ad/2002-05-09*. Juni 2002

[Wie00] Wiebicke, Ralf: *Utility Support for Checking OCL Business Rules in Java Programs*, Technische Universität Dresden, Diplomarbeit, Dezember 2000

[WK99] Warmer, Jos ; Kleppe, Anneke: *The object constraint language: precise modeling with UML*. Reading, Massachusetts : Addison Wesley Longman, Inc., 1999

[Zus94] Zuse, Horst: Complexity Metrics/Analysis. In: Marciniak, John J. (Hrsg.): *Encyclopedia of Software Engineering*. New York, NY, USA : John Wiley et Sons, 1994, S. 131–165

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Quelltexte

Index

Erklärung

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

Dresden, den 01. August 2002