Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik

Institut für Informatik
Arbeitsgruppe Softwaretechnik
Warburger Str. 100
33098 Paderborn

Generierung typsicherer Implementierungen für Assoziationen in UML-Modellen

Studienarbeit
zur Erlangung des Grades
Bachelor of Computer Science

vorgelegt von
Dietrich Travkin
Ginsterweg 1
33813 Oerlinghausen

vorgelegt bei
Prof. Dr. Wilhelm Schäfer
und
Prof. Dr. Hans Kleine Büning

21. Februar 2005

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Inhaltsverzeichnis

1. Einleitung

Mit zunehmend wachsenden Softwaresystemen und ihrer Komplexität nimmt die Bedeutung der modellbasierten Softwareentwicklung zu. Eine einheitliche Modellierungssprache wie UML [Obj03] trägt dazu bei, modellgetriebene Entwicklungsprozesse zu etablieren. Besonders wichtig ist bei dieser Art von Softwareentwicklung die Generierung von lauffähigem Code aus einem Modell.

Anstatt Änderungen während des Entwicklungsprozesses oder bei neuen Anforderungen an das Softwaresystem direkt im Quellcode durchzuführen, wird auf einer abstrakteren Ebene das zugehörige Modell geändert. Mit Hilfe von speziellen Werkzeugen wird anschließend Code aus dem Modell generiert, der die Semantik des Modells erhält. Diese Vorgehensweise ermöglicht einen schnellen Entwicklungsprozess und vereinfacht den Umgang mit Komplexität.

Ein Werkzeug, das sowohl die Modellierung von Softwaresystemen als auch die Generierung von Java-Code daraus unterstützt, ist die an der Universität Paderborn entwickelte *Fujaba Tool Suite* (kurz Fujaba) [Fuj04] [FNT98]. Diese Entwicklungsumgebung bietet neben zahlreichen UML-Diagrammarten wie Klassen-, Aktivitätsdiagrammen und Statecharts auch so genannte Story-Diagramme [FNTZ98]. Diese sind eine Art von Aktivitätsdiagrammen, die zur Beschreibung der dynamischen Änderungen von Objektstrukturen spezielle Kollaborationsdiagramme, so genannte Storypatterns, einbettet. Mit Hilfe dieser Diagrammarten, die mit einer formalen Semantik unterlegt sind, ist die Spezifikation von dynamischen und statischen Anteilen eines Softwaresystems sowie die Generierung von zugehörigem Code möglich.

In der grafischen Modellierungssprache UML können verschiedene Arten von Assoziationen zwischen zwei Klassen modelliert werden, insbesondere bidirektionale. Da diese von den meisten objektorientierten Programmiersprachen wie *Java* nicht unterstützt werden, bedürfen sie einer speziellen Abbildung von dem Modell auf lauffähigen Code, dessen Verwendung möglichst intuitiv und sicher sein soll.

1.1. Motivation

Die Entwicklungsumgebung Fujaba generiert unter Anderem Java-Quellcode für Assoziationen, an denen beliebig viele Elemente beteiligt sein können, z.B. für 1-zu-n-Assoziationen. Darin werden zur Verwaltung der durch eine Assoziation verbunden Elemente Container verwendet.

Damit diese Container beliebige Elemente enthalten können, wird der Typ der in einem Container verwaltbaren Elemente so allgemein wie möglich angegeben.

Dadurch können auch Elemente falschen Typs in die Container gelangen und es werden Typumwandlungen (*type casts*) im generierten Code notwendig. Typfehler werden erst durch Typprüfungen zur Laufzeit (dynamisch) erkannt und nur, wenn die entsprechende Stelle im Code ausgeführt wird. Deswegen bietet die von Fujaba generierte Java-Implementierung von Assoziationen bisher keine *Typsicherheit*. Eine ausschließlich statische Typisierung würde alle Typfehler bereits zur Übersetzungszeit aufdecken und Typumwandlungen unnötig machen.

Anstatt von Standard-Containern aus der Java-Bibliothek verwendet die Code-Generierung von Fujaba spezielle Container, die für die Implementierung von Assoziationen entwickelt wurden. Diese bieten die Funktionalität der Standard-Container und ergänzen diese, z.B. werden bei Änderungen des Container-Inhalts Nachrichten an Listener-Objekte verschickt. Die Implementierung der speziellen Container stellt zusätzlichen Code dar, der den Wartungsaufwand von Fujaba erhöht. Insbesondere ist die Einführung von Typsicherheit schwierig.

Fujaba generiert für jede Assoziation, an der eine Modellklasse beteiligt ist, diverse Methoden zum Verknüpfen und Trennen zweier Modellelemente (Instanzen einer Modellklasse) in die Modellklassen. Die große Anzahl der generierten Methoden macht den Quellcode unübersichtlich und erschwert seine Wartung.

An der Universität Kassel wurde eine andere Implementierung von Assoziationen vorgeschlagen [MZ04], die den generierten Code übersichtlicher und die Wartung einfacher macht. Bei dieser Implementierung werden die oben genannten Methoden in spezielle Klassen ausgelagert, die den Rollen von Assoziationen entsprechen. Anstatt der speziellen Container werden hier Standard-Container verwendet. Obwohl die Implementierung der Rollenklassen und der verwendeten Container bereits generisch ist, kann dieser Ansatz keine Typsicherheit garantieren, weil hier einige Typumwandlungen notwendig sind.

Im Rahmen dieser Studienarbeit wird untersucht, ob und wie eine typsichere Implementierung von Assoziationen erreicht und entsprechender Java-Code aus Modellen generiert werden kann. Zusätzlich sollen der Wartungsaufwand minimiert und die Lesbarkeit des generierten Codes erhöht werden.

1.2. Lösungsansatz

Der Sprachumfang der Programmiersprache Java wurde in der Version 1.5 unter Anderem um so genannte *Generics* erweitert. Dadurch sind generische Typdefinitionen und eine statische Typisierung möglich, die eine Basis für eine typsichere Assoziationsimplementierung bieten. Container können bei ihrer Implementierung mit dem Typ der von ihnen verwaltbaren Elemente parametrisiert werden. Bei der Initialisierung eines solchen Containers wird der konkrete Typ angegeben, sodass alle Typprüfungen bereits zur Übersetzungszeit erfolgen. Die Standard-Container der Java-Bibliothek wurden auf Generics umgestellt und bieten nun Typsicherheit.

Für eine Implementierung von insbesondere bidirektionalen Assoziationen werden wie in dem Ansatz in [MZ04] spezielle Rollenklassen definiert. Diese wer-

den vollständig generisch implementiert und kapseln alle für die Assoziationsimplementierung notwendigen Methoden. Für die Verwaltung der durch eine Assoziation verknüpfbaren Elemente werden intern anstatt der speziellen Container-Implementierungen die typsicheren Standard-Container der Java-Bibliothek verwendet. Die spezielle Implementierung der Rollen mit Hilfe von Generics soll den für Assoziationen generierten Code typsicher machen. Die Lesbarkeit und Benutzbarkeit sollen erhöht, sowie die Wartung erleichtert werden.

Die Rollenimplementierung wird in einer Bibliothek bereitgestellt, da diese in beliebigen Bereichen zur Implementierung von Assoziationen in Java verwendet werden kann. Zusätzlich wird ein Plug-In entwickelt, das die Code-Generierung von Fujaba anpasst und die generierte Assoziationsimplementierung typsicher macht.

1.3. Struktur der Arbeit

Zunächst werden in Kapitel 2 die verschiedenen Arten von Assoziationen in UML kurz vorgestellt. Anschließend wird das Konzept der Java Generics und ihre Verwendung an einigen kleinen Beispielen erklärt.

In Kapitel 3 werden Anforderungen an eine mögliche Implementierung von Assoziationen formuliert. Danach werden die generierte Assoziationsimplementierung von Fujaba und der Ansatz aus [MZ04] vorgestellt und bzgl. der Anforderungen evaluiert.

Wie eine typsichere Assoziationsimplementierung realisiert werden kann, wird in Kapitel 4 beschrieben. Dabei wird Bezug zu den beiden bisherigen Implementierungsansätzen genommen und darauf aufbauend eine Lösung entwickelt. Auch diese wird bzgl. der in Kapitel 3 formulierten Anforderungen evaluiert.

Die technische Realisierung dieser Lösung wird in Kapitel 5 beschrieben. Dazu gehört die Anpassung des bisherigen Code-Generierungsmechanismus von Fujaba, aber auch die Implementierung einer Rollenbibliothek, die im generierten Code verwendet wird.

Schließlich werden die Ergebnisse dieser Arbeit in Kapitel 6 zusammengefasst und in einem Ausblick mögliche Erweiterungen vorgeschlagen.

2. Grundlagen

2.1. Assoziationen

Bei der Modellierung mit UML und Fujaba [FNT98, S. 18 ff. und S. 152 ff.] können Beziehungen zwischen zwei Klassen durch zahlreiche Arten von Assoziationen in einem Klassendiagramm beschrieben werden. In Fujaba wird eine Teilmenge der in UML [Obj03, S. 3-68 ff.] spezifizierten Assoziationsarten unterstützt, die im Folgenden kurz vorgestellt wird.

Bidirektionale Assoziationen und Referenzen

In UML werden meist binäre Assoziationen verwendet. An einer binären Assoziation sind stets genau zwei Klassen beteiligt. Laut UML-Spezifikation [Obj03, S. 3-68 ff.] sind auch d-äre Assoziationen für ein beliebiges $d \in \mathbb{N}$ möglich, aber diese werden von Fujaba bisher nicht unterstützt. Binäre Assoziationen können bidirektional oder unidirektional sein. Die letzteren werden Referenzen genannt. Eine bidirektionale Assoziation kann – im Gegensatz zu einer unidirektionalen Assoziation – in beide Richtungen traversiert (durchlaufen) werden. Bei einer Referenz wird deshalb die Traversierungsrichtung (auch Navigationsrichtung genannt) explizit angegeben.

Die Verbindungsstellen einer Assoziation zu den Klassen werden Rollen genannt (zu jeder binären Assoziation gehören zwei Rollen). Sowohl die Assoziation als auch die zugehörigen Rollen können benannt werden. Es ist auch möglich, die Leserichtung der Assoziation zu kennzeichnen.

Die Rollen einer Assoziation legen die Assoziationsart fest und tragen den Hauptanteil der Information in einer Assoziation. Jede Rolle trägt eine Kardinalität (auch Multiplizität genannt), die angibt, wieviele Instanzen einer zugehörigen Klasse von der Rolle assoziiert werden dürfen. In Fujaba beschränkt man sich bei der Code-Generierung auf die Unterscheidung zwischen so genannten zu-n- und zu-1-Rollen, den Rollen mit Kardinalität 1 oder n (n steht dabei für eine beliebige Zahl). Dadurch sind nur 1-zu-1-, 1-zu-n- und n-zu-m-Assoziationen möglich.

Wenn nicht anders angegeben, wird bei zu-n-Rollen keine Ordnung der assoziierten Instanzen angenommen. Um dieses zu ändern, ist es möglich, eine solche Rolle als *geordnet* oder *sortiert* zu kennzeichnen.

Aggregation und Komposition

Aggregation und Komposition sind spezielle binäre Assoziationen. Eine Aggregation zwischen zwei Klassen stellt dar, dass eine Instanz einer der beiden Klassen Teil einer Instanz der anderen Klasse ist. Der umfassende Teil wird als Aggregat bezeichnet.

Die Semantik bei der (schwachen) Aggregation ist, dass wenn ein Aggregat gelöscht wird, es auch alle seine Teilobjekte löscht. Eine Komposition ist eine spezielle Form der Aggregation (starke Aggregation). Bei einer Komposition gilt zusätzlich, dass bei einer Erzeugung des Aggregats auch alle seine Teilobjekte erzeugt werden müssen und dass diese nicht ausgetauscht werden können (Das hat auch zur Folge, dass die Kardinalität bei der Rolle auf Aggregatseite 1 sein muss.). Die Code-Generierung in Fujaba unterstützt bisher nur die schwache Form der Aggregation.

Qualifizierte Assoziationen

Qualifizierte Assoziationen stellen eine weitere Form der binären 1-zu-n- und n-zu-m-Assoziationen dar. Hierbei wird die Menge der assoziierten Instanzen auf der n- bzw. m-Seite anhand eines Schlüssels partitioniert. Eine Instanz oder eine Menge von Instanzen auf der n- bzw. m-Seite ist dann über den zugehörigen Schlüssel erreichbar.

Die Kardinalitäten bei den qualifizierten Assoziationen haben auch eine etwas andere Bedeutung. Eine qualifizierte 1-zu-1-Assoziation zwischen den Klassen A und B, wobei die Instanzen von B über einen Schlüssel erreicht werden, ist tatsächlich eine 1-zu-n-Assoziation, bei der jede Instanz auf B-Seite über genau einen eindeutigen Schlüssel erreicht wird. Wäre bei dieser qualifizierten Assoziation auf B-Seite die Kardinalität n, so gäbe es zu je n Instanzen auf B-Seite genau einen eindeutigen Schlüssel, über den man diese Instanzmenge erreicht. Tatsächlich wäre es also eine 1-zu-$(n \cdot m)$-Assoziation, wenn m die Anzahl der möglichen Schlüssel wäre. Auch beidseitig qualifizierte Assoziationen sind möglich.

2.2. Generics in Java

Der Sprachumfang der Programmiersprache Java wurde in der Version 1.5 unter anderem um so genannte *Generics* erweitert. Diese ermöglichen eine generische Definition von Typen und Methoden, sodass darin verwendete Typen erst bei einer konkreten Anwendung festgelegt werden. Unabhängig von den tatsächlich verwendeten Typen können so Algorithmen und Datenstrukturen beschrieben werden, die trotz ihrer allgemeinen Definition typsicher sind.

In diesem Abschnitt werden die wichtigsten Eigenschaften der Java Generics anhand von einigen Anwendungsbeispielen kurz vorgestellt.

2.2.1. Einführung

Bei der Implementierung von allgemeinen Algorithmen und Datenstrukturen, die nur wenig oder gar nicht von den darin verwendeten Typen abhängen (z.B. bei Sortieralgorithmen und Containern), werden in Programmiersprachen wie Ada, Eiffel, ML und seit der Version 1.5 auch Java so genannte *formale Typparameter* verwendet, die bei einer konkreten Ausprägung des Algorithmus oder der Datenstruktur durch *Typargumente* ersetzt werden (siehe [Wat96, S. 124 ff, S. 141 ff und S. 248,249]). So können z.B. Listen definiert werden, die nur Einträge eines bestimmten Typs (beschrieben durch ein Typargument) haben können.

In älteren Versionen der Programmiersprache Java (vor Version 1.5) sind solche Konstrukte nicht möglich. Stattdessen wird z.B. bei der Implementierung von Containern der allgemeinste Java-Typ `java.lang.Object` verwendet. Dadurch können beliebige Elemente in einen solchen Container eingefügt werden, aber die Information über den Typ der eingefügten Elemente geht verloren. Durch Typprüfungen (*type casts*) wird zur Laufzeit sichergestellt, dass der Typ eines im Container enthaltenen Elements dem Typ entspricht, den ein Programmierer an einer Stelle im Code erwartet. Irrt sich der Programmierer, so tritt ein Typfehler zur Laufzeit auf.

Ein Beispiel ist in Abb. 2.1 abgebildet. Die Methode get der Klasse ArrayList (bzw. der Schnittstelle List) gibt ein Objekt vom Typ Object zurück. Deswegen wird eine Typumwandlung von Object zu Integer nötig, wenn das in der Liste enthaltene Integer-Objekt als solches verwendet werden soll. Durch die allgemeine Definition der Liste mit dem Typ Object als Typ der Einträge ist es auch möglich, Objekte von einem anderen Typ als Integer in die Liste einzufügen, was bei der späteren Typumwandlung zu Integer zu Laufzeitfehlern führt.

```
// Implementierung ohne Generics
List list = new ArrayList();
list.add(new Integer(1)); // auch list.add("a"); möglich
Integer value = (Integer) list.get(0); // Type Cast notwendig
```

Abbildung 2.1.: Benutzung eines Containers ohne die Verwendung von Generics

Das primäre Ziel bei der Entwicklung der Java Generics war *Typsicherheit* [Bra04, S. 15]. Dabei bedeutet Typsicherheit, dass alle Typprüfungen bereits zur Übersetzungszeit durchgeführt werden und alle Typfehler garantiert vom Übersetzer (*compiler*) erkannt werden [Wat96, S. 29] [Bra04, S. 2,15]. So kann eine wichtige Fehlerquelle vermieden werden, denn Typfehler machen einen wesentlichen Teil der Programmierfehler aus [Wat96, S. 29].

```
// Implementierung mit Generics
List<Integer> list = new ArrayList<Integer>();
list.add(new Integer(1)); // list.add("a"); führt zu Kompilierfehler
Integer value = list.get(0); // kein Type Cast mehr nötig
```

Abbildung 2.2.: Benutzung eines Containers unter Verwendung von Generics

Im Gegensatz zu dem in Abb. 2.1 dargestellten Code ist der in der Abb. 2.2 dargestellte generische Java-Code typsicher. Die verwendete Liste ist generisch mit dem Typparameter E implementiert (siehe Abb. 2.3). Dieser Typparameter wird bei der Instanziierung der Liste durch das Typargument Integer ersetzt. Auf diese Weise wird die Liste auf Einträge vom Typ Integer eingeschränkt. Der Typ einer so instanziierten Liste wird als *parametrisierter Typ* bezeichnet. Nun kann bereits zur Übersetzungszeit die Typ-Korrektheit der mit der Liste verwendeten Objekte überprüft werden. Das Einfügen eines String-Objekts ist hier nicht mehr möglich. Außerdem wird eine Typumwandlung unnötig, da die Methode get bereits Objekte des richtigen Typs, nämlich Integer, zurückgibt.

```
public interface List<E>
{
   boolean add(E o);
   E       get(int index);
   // weitere Methoden ...
}
```

Abbildung 2.3.: Generisch definierte Schnittstelle für Listen

2.2.2. Eigenschaften der Java Generics

Im Gegensatz zu den Templates der Programmiersprache C++ beschreibt ein generisch definierter Java-Typ nicht eine Familie von Klassen oder Schnittstellen, die sich nur durch die Typparameter-Ersetzungen durch konkrete Typen unterscheiden. Eine generische Typdeklaration wird nur einmal übersetzt. Alle parametrisierten Typen benutzen immer die gleiche Klasse oder Schnittstelle, auch zur Laufzeit. So benutzen z.B. die parametrisierten Typen `ArrayList<Integer>` und `ArrayList<String>` die gleiche Klasse ArrayList.

Als Konsequenz daraus werden alle statischen Variablen und Methoden einer generischen Klasse von allen ihren Instanzen unabhängig vom Typargument gemeinsam genutzt. Außerdem ist eine Abfrage wie `list instanceof List<Integer>`

nicht sinnvoll, da alle List-Objekte unabhängig vom Typargument Instanzen der gleichen Klasse sind. Zur Laufzeit existieren keine *Typvariablen* (in Abb. 2.3 ist das `E`), sodass eine Überprüfung auf die Verwendung eines bestimmten Typarguments hin (hier: Integer) nicht möglich ist. Ebenso kann die Korrektheit von Typumwandlungen wie

```
T something = (T) object;
```

für eine Typvariable `T` oder

```
List<Integer> list = (List<Integer>) object;
```

nicht überprüft werden. Um einen Programmierer darauf aufmerksam zu machen, gibt der Übersetzer in solchen Fällen Warnungen aus (*unchecked warnings*). Kann der Quellcode ohne diese Warnungen übersetzt werden, so ist er typsicher [Bra04, S. 15].

```
List<Integer> integerList = new ArrayList<Integer>();
List<Object> objectList = integerList; // erzeugt Kompilierfehler
objectList.add(new Object());
Integer zahl = integerList.get(0); // Zuweisung eines Object-Objekts
```

Abbildung 2.4.: Subtyping bei den Java Generics (`List<Integer>` ist nicht Untertyp von `List<Object>`)

Die Generics in Java verwenden eine nicht ganz intuitive Definition von Untertypen (*subtyping*). Obwohl der Typ `java.lang.Object` der Obertyp aller anderen nicht-primitiven Typen ist, kann z.B. `List<Integer>` nicht Untertyp von `List<Object>` sein. Das Beispiel in Abb. 2.4 soll das verdeutlichen. Hier könnte ein Objekt vom Typ Object in eine Liste von Integer-Objekten eingefügt werden, wenn der Übersetzter nicht einen Kompilierfehler erzeugen würde.

```
List<?> unknownList = new ArrayList<Integer>();
unknownList.add(new Integer()); // erzeugt Kompilierfehler
```

Abbildung 2.5.: Wildcards (Die Zuweisung erzeugt einen Kompilierfehler, da der Typ der in der Liste `unknownList` verwendbaren Objekte unbekannt ist.)

Damit ein Obertyp für alle parametrisierten Typen zu einer generischen Klasse angegeben werden kann, werden so genannte *Wildcards* verwendet, die durch das Zeichen `?` im Code gekennzeichnet werden. Der Typ `List<?>` ist dann der Obertyp von `List<Integer>` und `List<Object>`. Das Zeichen `?` steht dabei für einen unbekannten Typen. Die Konsequenz daraus ist aber, dass in eine Liste definiert wie in Abb. 2.5 keine Objekte eingefügt werden können, weil wegen des unbekannten Typen der Einträge keine Typprüfung möglich ist. Die von einer solchen Liste zurückgegebenen Objekte sind immer vom allgemeinsten Typ Object.

Ähnlich wie in der Programmiersprache Eiffel [Wat96, S. 249] können auch in Java bei der Definition einer generischen Klasse die Typparameter eingeschränkt werden. Dabei wird zwischen oberen und unteren Schranken unterschieden. Eine obere Schranke für die Einträge einer Liste kann z.B. durch den Ausdruck `List<T extends Number>` festgelegt werden. In diesem Fall kann die Liste nur Untertypen von Number enthalten. Eine untere Schranke wird definiert, indem anstatt des Schlüsselworts `extends` das Wort `super` verwendet wird. Solche Schranken können auch in Methoden oder Variablendeklarationen verwendet werden. In diesen Fällen wird die Typvariable `T` meist nicht benötigt und kann durch `?` ersetzt werden.

Damit der ohne Generics implementierte Java-Code auch in Verbindung mit generisch implementiertem Code benutzt werden kann, wurden in der Version 1.5 von Java so genannte Rohtypen *raw types* eingeführt. Ein Rohtyp ist ein generischer Typ ohne Typparameter. Ähnlich wie bei einem mit Wildcards parametrisierten Typ können dann beliebige Untertypen von Object für die Typvariablen verwendet werden. Bei der Verwendung von Rohtypen geht die Typsicherheit verloren, worauf der Übersetzer durch *unchecked warnings* aufmerksam macht, dafür kann aber der bisher existierende Java-Code unverändert weiterbenutzt werden.

```
abstract class C<A> {
   abstract A id(A x);
}

class D extends C<String> {
   String id(String x) {
      return x;
   }
}
```

```
abstract class C {
   abstract Object id(Object x);
}

class D extends C {
   String id(String x) {
      return x;
   }
   Object id(Object x) {
      return id((String) x);
   }
}
```

Abbildung 2.6.: Umwandlung von generischem Code (links) in nicht-generischen (rechts)

Der generische Java-Quellcode wird bei der Übersetzung in Bytecode so umgewandelt, dass er sich kaum von dem Bytecode älterer Java-Versionen unterscheidet. Dabei werden alle parametrisierten Typen und Typvariablen durch ein so genanntes Auslöschungsverfahren (*erasure*) entfernt. Generische Typen, Methoden und Ausdrücke werden in solche konvertiert, die keine Generics mehr verwenden [Bra04, S. 12] [BCK+01, S. 14 ff]. Wenn nötig, werden Typumwandlungen und zusätzliche Methoden eingefügt (siehe Beispiel in Abb. 2.6).

3. Ansätze zur Implementierung von Assoziationen

Es gibt verschiedene Ansätze, Assoziationen – insbesondere bidirektionale – auf Quellcode abzubilden. Bei allen Ansätzen müssen die von einer Assoziation verbundenen (referenzierten) Modellelemente abhängig von der Assoziationsart und ihren Eigenschaften verwaltet werden und es müssen diverse Methoden zur Benutzung der Assoziationen (z.B. verbinden zweier Objekte) implementiert werden.

Im Rahmen dieser Arbeit werden zwei Ansätze zur Implementierung von Assoziationen betrachtet. Bei dem ersten Ansatz wird die Verwaltung der referenzierten Elemente und die zugehörigen Methoden in die durch eine Assoziation verbunden Modellklassen verlagert. Bei dem zweiten Ansatz wird diese Verwaltung samt zugehöriger Methoden in eigenen, den Rollen einer Assoziation entsprechenden Klassen implementiert.

Nach der Formulierung von Anforderungen an eine mögliche Implementierung von Assoziationen werden in diesem Kapitel die beiden oben genannten Möglichkeiten anhand zweier vorliegender Umsetzungen genauer beleuchtet. Dazu gehören die von Fujaba [Fuj04] generierte Assoziationsimplementierung und ein neuer in [MZ04] beschriebener und in [Mai04] zum Teil realisierter Ansatz. Beide Ansätze werden bzgl. der formulierten Anforderungen evaluiert und verglichen. Zusätzlich wird diskutiert, wie die bisher generierte Assoziationsimplementierung von Fujaba typsicher gemacht werden kann.

3.1. Anforderungen an eine Implementierung

In UML können sowohl bidirektionale als auch unidirektionale Assoziationen modelliert werden (siehe Abschnitt 2.1, S. 5). Weitere Eigenschaften wie Kardinalitäten, Schlüssel (bei qualifizierten Assoziationen) und Constraints (z.B. geordnet) sowie diverse Anforderungen an den Quellcode und seine Benutzbarkeit verlangen nach einer speziellen Implementierung. Insbesondere ist es erwünscht, dass diese Implementierung Typsicherheit garantiert. Dadurch ist es möglich, bereits zur Übersetzungszeit sicherzustellen, dass eine Implementierung einer Assoziation zwischen zwei Klassen A und B nur erlaubt, Objekte vom Typ A und B (oder Untertypen davon) miteinander zu verbinden.

Im Nachfolgenden werden die entstehenden Anforderungen an eine Implementierung der Assoziationen formuliert und näher erläutert.

1. **Konsistenzerhaltung:** Bei bidirektionalen Assoziationen soll die Konsistenz stets gewahrt bleiben. D.h., wenn die Referenz von Objekt a zu Objekt b (bei einer bidirektionalen Assoziation zwischen a und b) durch den Aufruf einer Methode von a entfernt wird, so soll automatisch auch die Referenz von b zu a entfernt werden. Umgekehrt soll beim Erstellen einer Assoziation zwischen a und b sowohl eine Referenz von a zu b als auch eine Referenz von b zu a erstellt werden. Evtl. notwendige Konsistenzprüfungen dürfen nicht umgangen werden.

2. **Typsicherheit:** Alle Typprüfungen bei der Implementierung von Assoziationen sollen zur Übersetzungszeit erfolgen, um Typfehler zur Laufzeit zu vermeiden.

3. **Lesbarkeit:** Der Quellcode soll übersichtlich und für Anwendungsentwickler verständlich sein.

4. **Benutzbarkeit:** Die Implementierung von Assoziationen soll leicht zu benutzen sein. Dazu zählt auch, dass eine Iteration über die von einem Modellelement a referenzierten Elemente selbst dann sinnvoll fortgesetzt werden kann, wenn sich die Menge der von a referenzierten Elemente während der Iteration ändert.

5. **Wartbarkeit:** Der Wartungsaufwand spielt bei großen Projekten immer eine wichtige Rolle. Dieser soll für die Assoziationsimplementierung so gering wie möglich sein.

6. **Assoziationsarten:** Es sollen sowohl uni- als auch bidirektionale Assoziationen implementiert werden können. Assoziationsenden (Rollen) sollen die Kardinalitäten 1 oder n tragen. Zusätzlich sollen die Assoziationen qualifiziert sowie geordnet oder sortiert sein können.

7. **Thread-Sicherheit:** Die Assoziationsimplementierung soll für nebenläufige Anwendungen Thread-sicher sein. Dazu sollen kritische Bereiche im Code durch exklusive Zugriffe geschützt werden. Weil aber exklusive Zugriffe zwangsläufig die Laufzeit negativ beeinflussen,[1] soll diese Eigenschaft nur bei Bedarf erfüllt werden.

8. **Benachrichtigungsmechanismus:** Damit so genannte *Listener*-Objekte über Änderungen von Assoziationen informiert werden können, soll ein Benachrichtigungsmechanismus realisiert werden. Immer wenn bei einer Assoziation zwischen zwei Klassen A und B ein Objekt auf A- oder B-Seite hinzukommt

[1] Wird ein exklusiver Bereich von einem Thread erreicht, so werden andere Threads bei Zugriff auf den selben exklusiven Bereich blockiert, bis der kritische Bereich von dem ersten Thread verlassen wird. Die Überprüfung, ob ein Thread den exklusiven Bereich erreicht hat, kostet zusätzlichen Laufzeit-Overhead, was bei nicht-nebenläufigen Anwendungen zu unnötigen Laufzeiteinbußen führt.

oder entfernt wird, sollen die Listener benachrichtigt werden. Dazu sollen sie sich vorher für die Assoziation registrieren.

3.2. Rollenimplementierung innerhalb der Modellklassen

In diesem Abschnitt wird beschrieben, wie die Assoziationen in dem von Fujaba [Fuj04] generierten Code implementiert werden.

3.2.1. Beschreibung des Ansatzes

Überblick

Zur Verwaltung der durch eine Assoziation verbundenen Modellelemente werden diverse Methoden benötigt. Diese werden bei dem hier beschriebenen Ansatz innerhalb der Klassen implementiert, die in dem UML-Modell durch eine Assoziation verbunden sind. Die Methoden sollen es z.B. ermöglichen, zwei Elemente entlang einer Assoziation zu verbinden, sie zu trennen oder bei einer zu-n-Assoziation alle referenzierten Modellelemente aufzuzählen.

Wenn ein Element an mehreren Assoziationen beteiligt ist, werden diese Methoden für jede der Assoziationen implementiert. Die Art der Methoden und ihre Implementierung hängt von der Art der Assoziation und dem jeweiligen Assoziationsende (Rolle) ab: Zum Beispiel kann man bei einer zu-n-Rolle über alle referenzierten Elemente iterieren, während man bei einer zu-1-Rolle höchstens das einzige referenzierte Element zurückgegeben kann.

Zusätzlich zu den genannten Methoden erhalten die Klassen für jede der Assoziationen, an denen sie beteiligt sind, auch je ein Attribut, das entweder einen Container zur Verwaltung der Objektreferenzen (bei einer zu-n-Rolle) oder eine direkte Objektreferenz (bei einer zu-1-Rolle) speichert.

Die Methoden sind hauptsächlich für die Erhaltung der Konsistenz von bidirektionalen Assoziationen verantwortlich. Die restliche Funktionalität wird an speziell für diesen Zweck implementierte Container delegiert. Für jede Rollenart, z.B. qualifizierte und sortierte zu-n-Rolle, gibt es einen speziellen Container.

Anpassung der Modellklassen

Bei der bidirektionalen Assoziation bewohnt, wie sie in Abb. 3.1 modelliert ist, können Objekte der Klasse Haus bis zu n Mieter-Objekte referenzieren. Andersherum können Objekte der Klasse Mieter höchstens ein Haus-Objekt referenzieren. Festgelegt wird das von den beiden Rollen heim auf Haus-Seite und bewohner auf Mieter-Seite durch die Angabe der Kardinalitäten.

Damit die Objekte miteinander verbunden oder voneinander getrennt werden können, bekommt die Klasse Mieter ein Attribut mit dem Typ Haus und die Klasse

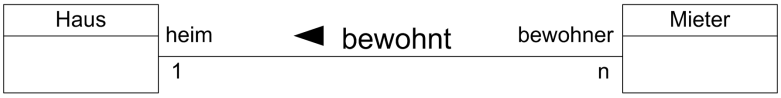

Abbildung 3.1.: UML-Klassendiagramm mit einer 1-zu-n-Assoziation

Haus einen Container (z.B. FHashSet) zum Speichern der referenzierten Elemente. Zusätzlich werden verschiedene Zugriffsmethoden implementiert. Die Methode addToBewohner in der Klasse Haus ermöglicht das Verbinden eines Haus-Objekts mit einem Mieter-Objekt und die Methode removeFromBewohner das Trennen. Die gleiche Funktionalität bieten die Methoden setHeim und getHeim in der Klasse Mieter. Andere Methoden wie iteratorOfBewohner, sizeOfBewohner und hasInBewohner werden implementiert, um alle über die Assoziation *bewohnt* referenzierten Elemente eines Haus-Objekts aufzuzählen, die Anzahl aller referenzierten Elemente anzugeben oder zu prüfen, ob ein bestimmtes Element referenziert wird.

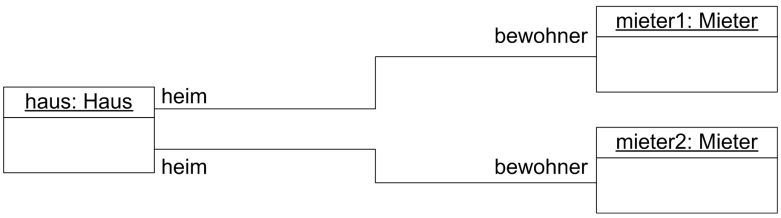

Abbildung 3.2.: Objektstrukturbeispiel zum Diagramm in Abb. 3.1

Man beachte, dass die Methodennamen den zugehörigen Rollennamen enthalten. Das dient der Unterscheidung der Assoziationsmethoden voneinander, wenn eine Klasse an mehreren Assoziationen beteiligt ist. Insbesondere bedeutet das, dass alle Rollennamen, die zu einer Klasse gehören, stets eindeutig sein müssen.

Abhängig von der Assoziationsart werden auch andere Methoden implementiert, z.B. erlaubt eine geordnete Assoziation das Einfügen einer Objektreferenz an einer bestimmten Stelle in der Liste der Referenzen.

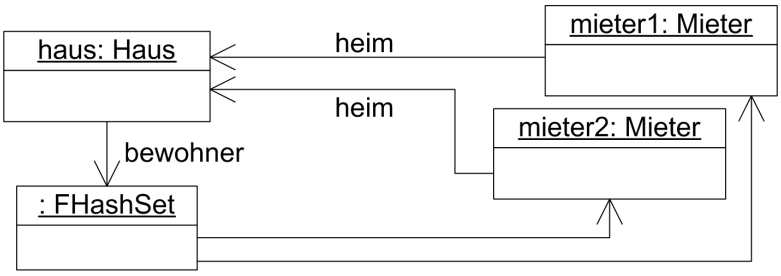

Abbildung 3.3.: Tatsächliche Objektstruktur zum Beispiel in Abb. 3.2

Realisierung der Assoziationen

Die Verwaltung der Referenzen unter Berücksichtigung der Assoziationseigenschaften übernehmen die verwendeten Container oder Attribute. Für ein beispielhaftes Objektdiagramm (Abb. 3.2) zum Modell in Abb. 3.1 wird in Abb. 3.3 die Objektstruktur so dargestellt, wie sie tatsächlich realisiert wird.

Um die Konsistenz bei bidirektionalen Assoziationen wie bewohnt in Abb. 3.1 zu erhalten (*Anforderung 1*), wird der Inhalt der Methoden addToBewohner und removeFromBewohner auf der Seite der zu-n-Rolle und die Methoden setHeim und getHeim auf der Seite der zu-1-Rolle an die gegenüberliegende Rolle angepasst. Wird eine Referenz auf ein Objekt mieter bei einer zu-n-Rolle zu dem Objekt haus hinzugefügt, so wird auch das haus-Objekt als Referenz zu dem Objekt mieter hinzugefügt bzw. gesetzt. Das gleiche gilt auch umgekehrt mit dem Unterschied, dass das mieter-Objekt von einem evtl. vorher referenzierten anderen Haus-Objekt (z.B. haus1) als Referenz entfernt wird, bevor das neue Haus-Objekt, nämlich haus, als Referenz gesetzt wird (siehe Abb. 3.4). Auch das Entfernen von Objektreferenzen wird ähnlich behandelt.

```
public boolean addToBewohner(Mieter value)
{
   boolean changed = false;
   if (value != null)
   {
      if (this.bewohner == null)
      {
         this.bewohner = new FHashSet ();
      }
      changed = this.bewohner.add (value);
      if (changed)
      {
         value.setHeim (this);
      }
   }
   return changed;
}
```

```
public boolean setHeim(Haus value)
{
   boolean changed = false;
   if (this.heim != value)
   {
      if (this.heim != null)
      {
         Haus oldValue = this.heim;
         this.heim = null;
         oldValue.removeFromBewohner (this);
      }
      this.heim = value;
      if (value != null)
      {
         value.addToBewohner (this);
      }
      changed = true;
   }
   return changed;
}
```

Abbildung 3.4.: Implementierung der Methoden addToBewohner und setHeim zum Diagramm in Abb. 3.1

Bei qualifizierten Assoziationen erhalten die Assoziationsmethoden einen Schlüssel als zusätzlichen Parameter. Bei dem Beispiel in Abb. 3.5 wird die Matrikelnummer eines Studenten als Schlüssel verwendet. Dieser wird in der Methode addToImmatrikulierte und removeFromImmatrikulierte der Klasse Universität verwendet. Dieser Schlüssel wird aber nicht nur auf der qualifizierten Seite der Assoziation benötigt, sondern auch auf der nicht qualifizierten (also in der Klasse Student). Er wird dazu benutzt, beim Verbinden oder Trennen zweier Modellelemente auch die Referenz in Rückrichtung – die ja qualifiziert ist – zu erstellen bzw. zu löschen, um die Konsistenz der bidirektionalen Assoziation zu erhalten. Beim Aufruf der Methode setUni der Klasse Student muss also zusätzlich zum Universität-Objekt

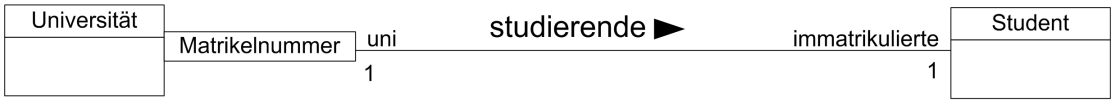

Abbildung 3.5.: Beispiel für eine einseitig qualifizierte Assoziation

auch der Schlüssel (die Matrikelnummer) übergeben werden, unter dem das Student-Objekt von dem Universität-Objekt aus zu erreichen ist. Bei einer beidseitig qualifizierten Assoziation werden sogar zwei Schlüssel benötigt, für je eine Seite der Assoziation einer. Fujaba enthält auch eine Unterstützung für beidseitig qualifizierte Assoziationen.

Bei einer unidirektionalen Assoziation (*Anforderung 6*) entfallen die oben beschriebenen Methoden auf der referenzierten Seite. Dann benötigt nur die referenzierende Seite die Assoziationsmethoden und beim Verbinden oder Trennen zweier Elemente werden keine Methoden auf der referenzierten Seite aufgerufen.

Damit auch Aggregationen auf Code abgebildet werden können, bekommen die Modellklassen so genannte removeYou-Methoden, die für jede Assoziation alle Referenzen zu anderen Modellelementen unter Einhaltung der Konsistenz bei bidirektionelen Assoziationen entfernen.[2] Bei dem Beispieldiagramm aus Abb. 3.1 (S. 14) wird dazu die Methode removeAllFromBewohner auf der Haus-Seite für die Assoziation *bewohnt* aufgerufen. Diese entfernt alle Referenzen zu Mieter-Objekten. Eine removeAllFrom...-Methode existiert für jedes Assoziationsende (Rolle), das mehr als ein Element referenzieren kann.

Verwendung von Containern

Für die Verwaltung der von einem Assoziationsende (Rolle) referenzierten Modellelemente werden Container oder Attribute verwendet.

Bei nahezu allen in Abschnitt 2.1 vorgestellten Assoziationsarten referenziert eines der beiden Assoziationsenden (Rollen) mehr als nur ein Modellelement. Die einzige Ausnahme bilden die zu-1-Rollen von nicht qualifizierten Assoziationen. Da diese Rollen maximal ein Element referenzieren können, reicht für die Verwaltung ein Attribut mit dem Typ des zu referenzierenden Objekts aus. Wird ein Objekt referenziert, so wird eine Referenz darauf in dem Attribut gespeichert, ansonsten ist der Eintrag `null`.

Qualifizierte Rollen sowie nicht qualifizierte zu-n-Rollen müssen eine beliebige Anzahl von Objekten verwalten können, wobei der Zugriff auf die von einer qualifizierten Rolle verwalteten Objekte über den zugehörigen Schlüssel erfolgen muss. Zusätzlich können diese drei Rollenarten, nämlich nicht qualifizierte zu-n- und qualifizierte zu-1- und zu-n-Rollen, geordnet oder sortiert sein. In diesen Fällen müssen die Einträge bei einer geordneten Assoziation stets ihre Reihenfolge behal-

[2] In Java ist es nicht möglich, Objekte zu löschen. Um der Semantik von Aggregationen nachzukommen, werden alle Objektreferenzen auf „zu löschende“ Objekte entfernt, damit diese von dem *Garbage Collector* von Java eingesammelt werden können.

ten, während sie bei einer sortierten Assoziation nach einem bestimmten Kriterium sortiert vorliegen müssen. Für alle diese Rollenarten außer der nicht qualifizierten zu-1-Rolle werden also spezielle Container benötigt, die eine beliebige Anzahl von Objekten in der beschriebenen Weise verwalten (*Anforderung 6*).

Die in dem *Java Collections Framework* enthaltenen Definitionen von Set-, Map- und List-Schnittstellen (Paket `java.util`) stellen eine gute Basis für die Container-Implementierung dar. Die zugehörigen Standard-Implementierungen der Schnittstellen (z.B. `java.util.HashMap`) haben aber die Eigenschaft, dass es bei der Verwendung von Iteratoren (`java.util.Iterator`) zu Ausnahmen (ConcurrentModificationExceptions) kommt, wenn sich der Container-Inhalt während der Iteration ändert. Dadurch wird die Benutzung der Container erschwert (*Anforderung 4*). Außerdem ist es möglich, Container-Einträge während der Iteration zu entfernen. Das führt dazu, das evtl. notwendige Konsistenzprüfungen innerhalb der Assoziationsmethoden umgangen werden können (*Anforderung 1*).

Es wurde festgestellt, dass in allen möglichen Fällen der Iterator eines Set-, Map- oder List-Objekts sinnvoll weiterverwendet werden kann, nachdem der Container-Inhalt sich geändert hat. Um ConcurrentModificationExceptions zu vermeiden und Modifikationen des Container-Inhalts durch Iteratoren zu verbieten, haben sich die Entwickler von Fujaba dafür entschieden, die von Java bereitgestellten Container anzupassen, obwohl die benötigte Funktionalität durch die Standard-Container der Java-Bibliothek zum größten Teil bereitgestellt wird.

Unglücklicherweise war es nicht möglich, die bereits vorhandene Container-Implementierung von Java (z.B. die Klassen HashMap, TreeSet, LinkedList) durch Vererbung (Spezialisierung) zu erweitern. Die Iteratoren der Standard-Container werden durch interne Klassen implementiert. Diese Klassen und einige für eine Spezialisierung wichtige Variablen sind für erbende Klassen nicht sichtbar. Aus diesem Grund wurde für die Fujaba-Implementierung der Container der Quellcode der Java-Standard-Container kopiert und an die Anforderungen angepasst. So entstand für jede der oben genannten Rollenarten eine spezielle Container-Implementierung in dem Paket `de.upb.tools.fca`.

Für jeden der realisierten Container gibt es auch eine Version, die einen Benachrichtigungsmechanismus implementiert (*Anforderung 8*). Mit Hilfe von weiteren Methoden in den Modellklassen können sich Listener-Objekte bei einem Modellelement registrieren, um bei Änderungen der Assoziation durch PropertyChangeEvents benachrichtigt zu werden.

Damit die Assoziationsimplementierung auch in nebenläufigen Anwendungen verwendet werden kann und Thread-sicher ist (*Anforderung 7*), werden alle Zugriffsmethoden der Container als kritische Bereiche betrachtet und durch exklusive Zugriffe geschützt. Das passiert durch das Schlüsselwort `synchronized` bei der Deklaration der Container-Methoden. Diese Funktionalität kann aber nicht abgestellt werden.

Typsicherheit

Die Standard-Container der Java-Bibliothek, die für die spezielle Container-Implementierung verwendet wurden, sind als allgemeine nicht-homogene Container[3] implementiert. Generics waren zu diesem Zeitpunkt nicht verfügbar. Deswegen sind Typumwandlungen (*type casts*) von dem allgemeinen Typ `java.lang.Object` zu dem speziellen Typ (z.B. Mieter) notwendig. Die Korrektheit dieser Typumwandlungen kann nur zur Laufzeit überprüft werden, wodurch bei der speziellen Implementierung der Container und damit auch bei der gesamten Assoziationsimplementierung keine Typsicherheit gegeben ist (*Anforderung 2*).

Außerdem können Objekte falschen Typs in die Container eingefügt werden, obwohl die Assoziationsmethoden – wie addToBewohner und setHeim in Abb. 3.4 (S. 15) – die richtigen Typen (hier: Mieter und Haus) bei ihren Parametern verwenden. Eine Methode, die in der gleichen Klasse wie eine der Assoziationsmethoden definiert ist, kann auf alle in der gleichen Klasse definierten Container direkt zugreifen. Dadurch kann diese Methode auch Objekte beliebigen Typs in die allgemein definierten Container einfügen. Eine Verwendung der Assoziationsmethoden kann so umgangen werden. Der entstehende Typfehler wird erst zur Laufzeit erkannt.

Es ist möglich, diese Implementierung typsicher zu machen, indem generische und typsichere Container verwendet und die Rümpfe der generierten Assoziationsmethoden angepasst werden (siehe Abschnitt 3.2.3, S. 20). Die Container können mit den Typen der an einer Assoziation beteiligten Modellelemente parametrisiert werden. Dadurch können Typumwandlungen (z.B. bei der Iteration über die referenzierten Elemente) innerhalb der Assoziationsmethoden verhindert und eine statische Typisierung erreicht werden.

3.2.2. Evaluation

Die Assoziationsimplementierung bei diesem Ansatz ist bisher nicht typsicher (*Anforderung 2*). Dadurch, dass sie sich komplett innerhalb der beiden an einer Assoziation beteiligten Klassen befindet (und nicht etwa in einer Bibliothek), ist es aber möglich den Code beliebig anzupassen und insbesondere auch typsicher zu machen. Die Konsistenz bei bidirektionalen Assoziationen wird stets gewahrt (*Anforderung 1*).

Leider werden sehr viele Assoziationsmethoden in den modellierten Klassen implementiert, was den Code unübersichtlich (*Anforderung 3*) macht und die Benutzung (*Anforderung 4*) so implementierter Modelle sowie deren Wartung erschwert (*Anforderung 5*). Bei einer geordneten 1-zu-n-Assoziation z.B. werden in der Klasse auf der Seite der zu-n-Rolle 18 verschiedene Methoden zur Verwen-

[3] *Homogene Container* können nur Objekte eines bestimmten Typs enthalten. Das Einfügen von Objekten eines anderen Typs ist nicht möglich. Im Gegensatz dazu gibt es allgemein definierte Container, die Objekte beliebigen Typs enthalten können (in Java sind das Objekte vom Typ `java.lang.Object`).

dung der Assoziation implementiert. Schon bei wenigen Assoziationen, an denen eine Klasse beteiligt ist, überwiegt damit die Anzahl der darin implementierten Assoziationsmethoden die Anzahl der modellierten Methoden.

Die Wartung der speziellen Container-Implementierung ist schwierig, denn die Standard-Container wurden durch eine Modifikation ihres Quellcodes neu implementiert. Änderungen der Container-Implementierung von Java (z.B. die Umstellung auf Generics in der Java-Version 1.5) haben keine Auswirkungen auf diese spezielle Implementierung. Sie muss von ihren Entwicklern extra angepasst werden.

Bei einer Änderung der Assoziationsimplementierung (ohne, die Schnittstellen zu verändern) würden sich auch die Rümpfe der Assoziationsmethoden ändern. Dadurch muss die Implementierung des Modells neu generiert bzw. angepasst und erneut kompiliert werden.

Die Methodenrümpfe der verwendeten Assoziationsmethoden sind bei Rollen (bzw. Assoziationen) mit gleichen Eigenschaften nahezu identisch. Trotzdem werden sie für jede Assoziation getrennt implementiert. Dadurch entsteht Code-Redundanz, was wiederum die Wartung eines so implementierten Modells erschwert.

Dadurch, dass es keine allgemeinen Schnittstellen für Assoziationen gibt und die Methodennamen sich immer unterscheiden, ist eine Gleichbehandlung ähnlicher oder gleicher Assoziationen (Assoziationen mit gleichen Eigenschaften, z.B. mehrere geordnete 1-zu-n Assoziationen) nicht ohne größeren Aufwand (z.B. durch *Reflection*[4]) möglich.

Die vorgestellte Assoziationsimplementierung ermöglicht eine Abbildung aller in Abschnitt 2.1 (S. 5) vorgestellten Assoziationen auf Java-Quellcode (*Anforderung 6*).

Durch die exklusiven Zugriffe auf Container wird Thread-Sicherheit geboten (*Anforderung 7*), allerdings kann diese nicht abgestellt werden, um die Laufzeit von nicht-nebenläufigen Anwendungen zu verbessern.

Ein Benachrichtigungsmechanismus wird mit Hilfe von speziellen Containern realisiert (*Anforderung 8*).

Vorteile

- Typsicherheit (fehlt in Fujaba, kann aber realisiert werden)
- Konsistenzerhaltung bei bidirektionalen Assoziationen
- Thread-Sicherheit (nicht abstellbar, aber vorhanden)
- Benachrichtigungsmechanismus

[4] *Reflection* ist eine Technik der Programmiersprache Java, zur Laufzeit Informationen über die Klasse eines Objekts zu bekommen. Das ermöglicht unter Anderem auch das Finden und Aufrufen von in einer Klasse deklarierten Methoden, wenn die Methodensignatur bekannt ist.

Nachteile

- erschwerte Anwendungsentwicklung und hoher Wartungsaufwand durch zu viele Methoden, Fehlen von allgemeinen Assoziationsschnittstellen, Code-Redundanz und eine spezielle Container-Implementierung
- Kompilierabhängigkeit zwischen Modell und der Assoziationsimplementierung: Änderungen an der Assoziationsimplementierung erfordern eine Anpassung (oder Neugenerierung) der Modellimplementierung und ihre Neukompilierung

3.2.3. Typsicherheit durch spezielle Java-Container

In diesem Abschnitt wird erläutert, wie die in dem Abschnitt 3.2.1 (S. 13) beschriebene Assoziationsimplementierung typsicher gemacht werden könnte. Dabei wird der Ansatz aus Abschnitt 3.2.1 angepasst und anschließend bzgl. der Anforderungen an eine Assoziationsimplementierung evaluiert.

Typsicherheit

Die Assoziationsmethoden werden bei dem in Abschnitt 3.2.1 (S. 13) beschriebenen Ansatz für jede Assoziation, an der eine Modellklasse beteiligt ist, speziell generiert. Die Parameter der Methoden haben den Typ der durch eine Assoziation verbindbaren Modellelemente.

Die Typsicherheit geht durch die Verwendung von nicht-homogenen Containern und nicht generischen Iteratoren verloren. Diese können nur Objekte von dem allgemeinsten Java-Typ `java.lang.Object` zurückgeben. Das wiederum erfordert Typumwandlungen in den tatsächlichen Typ der verwalteten Elemente und damit auch Typprüfungen zur Laufzeit.

Durch die Verwendung von typsicheren Containern, die seit der Version 1.5 in der Java-Bibliothek verfügbar sind, und einigen Anpassungen am generierten Code kann Typsicherheit hergestellt werden.

Dazu werden die verwendeten Container mit den Typen der Modellelemente parametrisiert. Anstatt der allgemeinen Iteratoren geben die Assoziationsmethoden generische parametrisierte Iteratoren zurück. Bei dem Beispiel für das Klassendiagramm in Abb. 3.1 (S. 14) wird der Container anstatt durch die Anweisung

```
this.bewohner = new FHashSet();
```

in der Methode addToBewohner in Abb. 3.4 (S. 15) durch die Anweisung

```
this.bewohner = new HashSet<Mieter>();
```

instanziiert. Der verwendete Container wird also mit dem Typ Mieter der verwaltbaren Modellelemente parametrisiert.

Einige weitere Methoden müssen ebenfalls angepasst werden, um die parametrisierten Container zu verwenden. Für das Beispiel Abb. 3.1 (S. 14) wird unter Anderem auch die Methode removeAllFromBewohner in die Modellklasse Haus generiert. Diese würde nun einen mit dem Typ Mieter parametrisierten Iterator verwenden, wodurch die Typumwandlung (*type cast*) unnötig wird (siehe Abb. 3.6).

```
public void removeAllFromBewohner()
{
   Mieter tmpValue;
   Iterator iter =
      this.iteratorOfBewohner();
   while (iter.hasNext())
   {
      tmpValue = (Mieter) iter.next ();
      this.removeFromBewohner (tmpValue);
   }
}
```

```
public void removeAllFromBewohner()
{
   Mieter tmpValue;
   Iterator<Mieter> iter =
      this.iteratorOfBewohner();
   while (iter.hasNext())
   {
      tmpValue = iter.next ();
      this.removeFromBewohner (tmpValue);
   }
}
```

Abbildung 3.6.: Anpassung der Methode removeAllFromBewohner zu dem Beispiel in Abb. 3.1 (S. 14)

Durch die Verwendung typsicherer parametrisierter Container und die zugehörigen Anpassungen in den Assoziationsmethoden kann der für die Assoziationsimplementierung generierte Code typsicher gemacht werden. Alle Typprüfungen erfolgen dann zur Übersetzungszeit.

Verwendung von Containern

Das Problem, das auch schon die Fujaba-Entwickler hatten und durch eine eigene Implementierung gelöst haben, ist die fehlende Funktionalität bei den Standard-Containern der Java-Bibliothek. Diese bieten z.B. keinen Benachrichtigungsmechanismus (*Anforderung 8*) und erzeugen Laufzeitfehler, wenn sich bei der Iteration über die in einem Container verwalteten Elemente der Container-Inhalt ändert (*Anforderung 4*).

Abhilfe schaffen da die seit der Java-Version 1.5 verfügbaren Container in dem Paket `java.util.concurrent` der Java-Bibliothek. Dazu zählen die Klassen ConcurrentHashMap und ConcurrentLinkedQueue. Diese Container sind nicht nur typsicher (*Anforderung 2*), sondern sie stellen auch Iteratoren zur Verfügung, die keine Laufzeitfehler erzeugen (*Anforderung 4*) und bieten Thread-Sicherheit (*Anforderung 7*) durch exklusive Zugriffe auf die Container.

Nicht-exklusive Zugriffe sind allerdings nicht möglich, weil alle Methoden mit dem Schlüsselwort `synchronized` deklariert sind und diese Eigenschaft nicht abgestellt werden kann.

Diese Container sind für die Anwendung in nebenläufigen Programmen optimiert. Sie sind so organisiert, dass sie eine bestimmte feste Anzahl nebenläufiger Zugriffe erlauben, ohne die zugreifenden Threads zu blockieren. Das wird dadurch erreicht, dass die Container-Inhalte partitioniert und die Zugriffe unabhängig voneinander auf den einzelnen Partitionen behandelt werden.

Damit auch Benachrichtigungen (PropertyChangeEvents) bei Änderungen des Container-Inhalts verschickt werden (*Anforderung 8*), könnten spezielle Wrapper-Klassen für die Container-Klassen implementiert werden. Die Wrapper-Klassen würden die gesamte Funktionalität an die Container delegieren und zusätzlich das Registrieren von Listener-Objekten ermöglichen sowie diese bei Änderungen benachrichtigen.

Die von diesen Containern bereitgestellten Iteratoren (`java.util.Iterator`) bieten die Möglichkeit, die Container-Inhalte zu verändern (`Iterator.remove()`). Es ist aber unerwünscht, dass der Container-Inhalt außerhalb der Assoziationsmethoden verändert werden kann. Dadurch wäre es möglich, Konsistenzprüfungen innnerhalb der Assoziationsmethoden zu umgehen (*Anforderung 1*).

Um die remove-Operation auf einem Iterator zu verbieten, kann eine Wrapper-Klasse für Iteratoren implementiert werden, die die Iterator-Methoden hasNext und next an den tatsächlichen Iterator delegiert, bei der Methode remove aber eine UnsupportedOperationException wirft. Anstatt des tatsächlichen Iterators wird von den Assoziationsmethoden (z.B. iteratorOfBewohner bei dem Beispiel in Abb. 3.1 auf S. 14) der Iterator-Wrapper verwendet bzw. zurückgegeben.

In der Klasse Collections (Paket `java.util`) sind auch Wrapper für unveränderbare Sets, Maps und Lists vorhanden (z.B. durch die Methode unmodifiableMap), die es ermöglichen, Änderungen von Container-Objekten zu verbieten. Auch eine Änderung durch den Iterator eines Containers ist dann nicht mehr möglich. Die Verwendung dieser Wrapper ist eine Alternative für die Implementierung eines eigenen Iterator-Wrappers.

Leider existieren zur Zeit nur die zwei genannten Container-Klassen in der Java-Bibliothek (Version 1.5). Für die Implementierung von Assoziationen werden aber unter Anderem Container benötigt, die die verwalteten Objekte sortieren. Das trifft auf die Klassen ConcurrentHashMap und ConcurrentLinkedQueue nicht zu. Erwünscht wären entsprechende Concurrent-Versionen der Klassen LinkedList, HashSet, TreeMap und TreeSet aus dem Paket `java.util`. Laut Aussagen von Doug Lea[5] vom Februar 2005 sind einige zusätzliche Klassen dieser Art und andere bereits im Rahmen des JSR-166[6] implementiert worden und sollen in die nächste Java-Version[7] intergriert werden. Eine Vorschau der überarbeiteten Schnittstellen (APIs) ist unter [Lea05] erhältlich. Zu den neuen Klassen zählen insbesondere ConcurrentSkipListMap und ConcurrentSkipListSet. Diese bieten eine Obermenge der Funktionalität von TreeMap und TreeSet aus dem Paket `java.util`. Zusammen mit diesen neuen Klassen wäre es vermutlich möglich, eine Assoziationsimplemen-

[5] *Doug Lea* ist ein Professor der Informatik an der State University of New York at Oswego. Er ist Autor des Buchs *Concurrent Programming in Java: Design principles and patterns* (ISBN 0-201-31009-0) und verschiedener weit verbreiteter Softwarekomponenten. Insbesondere ist er Autor der beiden Klassen ConcurrentHashMap und ConcurrentLinkedQueue.

[6] JSR steht für Java Specification Request. Im JSR-166 werden einige Werkzeuge für die Verwendung in nebenläufigen Programmen vorgeschlagen. Hauptveranwortlicher ist Doug Lea.

[7] Voraussichtlich wird Java 1.6 mit dem Code-Namen *Mustang* die nächste Java-Version sein, nachdem Version 1.5.1 (*Dragonfly*) doch nicht veröffentlicht wird.

tierung unter Verwendung der Concurrent-Klassen zu realisieren. Das muss aber nach Veröffentlichung der neuen Klassen genauer untersucht werden.

Evaluation

Angenommen, es wären alle genannten Concurrent-Klassen verfügbar, dann wären bei der Verwendung dieser Container-Implementierung nur einige Wrapper-Klassen notwendig. Eine eigene Implementierung der Container könnte vermieden werden.

Die Konsistenz bei bidirektionalen Assoziationen wäre weiterhin garantiert (*Anforderung 1*). Zusätzlich wäre die Assoziationsimplementierung aber typsicher (*Anforderung 2*).

Die Container in dem Paket `java.util.concurrent` erzeugen bei Iterationen selbst dann keine Laufzeitfehler, wenn sich der Container-Inhalt währenddessen ändert (*Anforderung 4*). Das erhöht die Benutzbarkeit.

Der Aufwand für die Wartung der Container-Klassen könnte drastisch reduziert werden (*Anforderung 5*), da dieser Ansatz auf Containern aus der Java-Bibliothek basiert.

Thread-Sicherheit wäre gegeben (*Anforderung 7*), aber es wären keine nichtexklusiven Zugriffe auf Container-Inhalte möglich, was die Performance von nichtnebenläufigen Anwendungen negativ beeinflussen würde. Bei nebenläufigen Anwendungen dagegen könnten die Container die Performance erhöhen, da auch mehrere nebenläufige Zugriffe auf die Container-Inhalte ohne Blockieren möglich sind.

Ein Benachrichtigungsmechanismus (*Anforderung 8*) wird von den Java-Containern nicht bereitgestellt und müsste extra implementiert werden, z.B. in Wrapper-Klassen.

Es müsste auch überprüft werden, um wie viel der Laufzeit- und Speicherbedarf wächst, der durch die Partitionierung der Container-Inhalte und die exklusiven Zugriffe darauf entsteht.

Solange sich keine weiteren Container wie ConcurrentHashMap und ConcurrentLinkedQueue in der Java-Bibliothek befinden, insbesondere mit der Unterstützung für Sortierung, ist eine Umsetzung dieses Ansatzes vorerst nicht möglich.

3.3. Rollen als eigenständige Klassen

Mitarbeiter der Software Engineering Research Group an der Universität Kassel haben eine andere als die bisher in Fujaba verwendete Lösung entwickelt und für die Implementierung von Assoziationen vorgeschlagen [MZ04]. Diese Lösung zielt hauptsächlich darauf ab, den von Fujaba generierten Code lesbarer und die Code-Generierung für Assoziationen leichter wartbar zu machen. Insbesondere wollte man die vielen für die Implementierung der Assoziationen generierten Methoden aus den Modellklassen in eigene spezielle Klassen verlagern. Zusätzlich ist für

diesen Ansatz die Verwendung von Java Generics geplant, um Typsicherheit zu garantieren, allerdings ist das bisher nicht vollständig umgesetzt worden.

Auch in [HBR00] wird eine Abbildung von UML-Modellen auf Java-Code vorgestellt. Hier werden für jede Modellklasse zwei Java-Klassen generiert (eine abstrakte für die Schnittstelle und eine für das Verhalten). Assoziationen werden mit Hilfe von so genannten *Cursor*n implementiert, die den Rollen von an Assoziationen beteiligten Objekten entsprechen. Der in [HBR00] verwendete Ansatz zur Implementierung von Assoziationen ähnelt dem Ansatz aus [MZ04], jedoch verspricht der Ansatz von Maier und Zündorf [MZ04] eine höhere Benutzbarkeit und Unabhängigkeit der Assoziationsimplementierung von der Modellimplementierung. Deswegen wird hier nur der in [MZ04] vorgeschlagene Ansatz betrachet.

3.3.1. Beschreibung des Ansatzes

Überblick

Bei dieser Vorgehensweise werden Assoziationen mit Hilfe von Rollenklassen implementiert. Jedes Rollenobjekt repräsentiert ein Ende einer Assoziation und stellt alle zur Verwaltung der Assoziation benötigten Methoden zur Verfügung, die bisher in die Modellklasse generiert wurden.

Jedes Modellelement besitzt für jede Assoziation, an der es beteiligt ist, ein Rollenobjekt. Die Rollen ermöglichen das Verbinden und Trennen von Modellelementen, deren Klassen laut UML-Modell durch eine Assoziation verbunden sind. Insbesondere wird durch die Rollen die Konsistenz der bidirektionalen Assoziationen beim Verbinden und Trennen aufrechterhalten.

Die Rollen bieten abhängig von der Art der Assoziation, zu der sie gehören, verschiedene Methoden und Funktionalität. So gibt es z.B. spezielle Klassen für qualifizierte und geordnete Rollen. Da die Rollenimplementierung allgemein verwendbar ist, werden alle Rollen in einer öffentlich zugänglichen Bibliothek zusammengefasst [Mai04].

Für die Verwaltung der durch eine Assoziation verbundenen Modellelemente werden Standard-Container der Java-Bibliothek innerhalb der Rollen verwendet.

Bevor die Details genauer erläutert werden, muss erwähnt werden, dass dieser Ansatz [MZ04] bis zur Fertigstellung dieser Studienarbeit noch in Entwicklung war. Aus diesem Grund kann hier nur die bis dahin (21.02.2005) vorliegende Version 0.4 [Mai04] vorgestellt und evaluiert werden.

Anpassung der Modellklassen

Bei diesem Ansatz werden Assoziationen mit Hilfe von eigenständigen Klassen für Rollen implementiert. Für jede Assoziation, an der ein Modellelement beteiligt sein kann, erhält es ein an die Assoziationsart angepasstes Rollenobjekt. Über je ein Attribut hält das Modellelement eine Referenz zu seiner Rolle und umgekehrt (d.h. jedes Rollenobjekt kennt auch seinen Besitzer).

Die Rollen werden als Container innerhalb der Modellklassen benutzt. Sie werden bei Bedarf in einer Getter-Methode der Modellklasse erstellt, eine Setter-Methode existiert nicht. Um zwei Modellelemente miteinander zu verbinden oder voneinander zu trennen, können die Methoden der Rollenobjekte verwendet werden.

```
class Haus implements PropertyChangeSource
{
  private ToManyRole<Haus,Mieter> bewohner = null;
  public ToManyRole<Haus,Mieter> bewohner()
  {
    if (this.bewohner == null)
    {
      this.bewohner = new ToManyRole<Haus,Mieter>(
        "heim", "propertyHeim", this);
    }
    return this.bewohner;
  }
  // PropertyChangeSource methods
  ...
}
```

```
class Mieter implements PropertyChangeSource
{
  private ToOneRole<Mieter,Haus> heim = null;
  public ToOneRole<Mieter,Haus> heim()
  {
    if (this.heim == null)
    {
      this.heim = new ToOneRole<Mieter,Haus>(
        "bewohner", "propertyMieter", this);
    }
    return this.heim;
  }
  // PropertyChangeSource methods
  ...
}
```

Abbildung 3.7.: Generische Implementierung der Modellklassen zum Diagramm in Abb. 3.1

Durch die Verlagerung der für die Verwaltung von Assoziationen benötigten Methoden in eigene dafür vorgesehene Rollenklassen soll der Quellcode für die modellierten UML-Klassen (Modellklassen) kürzer und übersichtlicher werden (*Anforderung 3*). Der Quellcode zu dem Klassendiagramm in Abb. 3.1 auf Seite 14 würde wie in Abb. 3.7 aussehen. Es wird für jede Assoziation, an der ein Modellelement beteiligt sein kann, nur noch eine Methode und ein Attribut innerhalb der Modellklasse implementiert.

Die bisher vorliegende Rollenimplementierung [Mai04] sieht eine Unterstützung für einen Benachrichtigungsmechanismus vor (*Anforderung 8*), die aber bisher nicht realisiert wurde. Dafür ist es allerdings notwendig, dass die Modellklassen die Schnittstelle PropertyChangeSource implementieren (siehe Abb. 3.7). Diese definiert Methoden zum Feuern von PropertyChangeEvents und Registrieren von Listenern.

Realisierung der Assoziationen

Die Rollenklassen kapseln die Implementierung von Assoziationen. Je zwei Rollen repräsentieren dabei eine Assoziation. Diese Klassen sind generisch implementiert und brauchen für ihre Verwendung in der Modellimplementierung nicht neu implementiert oder erweitert zu werden.

Zum Vergleich zum vorhergehenden Implementierungsansatz zeigt Abb. 3.8 die Objektstruktur wie sie bei der beispielhaften Instanziierung (Abb. 3.2, S. 14) von dem Klassendiagramm in Abb. 3.1 (S. 14) aussehen würde.

Eine Rolle verwaltet alle über eine Assoziation referenzierten Modellelemente und ist dafür verantwortlich, beim Verbinden oder Trennen zweier Elemene über

eine bidirektionale Assoziation die Konsistenz zu erhalten (*Anforderung 1*).

Bei dem Beispiel in Abb. 3.1 auf Seite 14 (siehe auch Abb. 3.8) wird die Konsistenz der bidirektionalen Assoziation bewohnt erhalten, indem die Rolle *bewohner* (hier: ToManyRole) des Objekts haus beim Verbinden dieses Objekts mit einem anderen Objekt mieter1 entlang der Assoziation *bewohnt* nicht nur eine Referenz auf das Objekt mieter1 zu den in der Rolle *bewohner* verwalteten Referenzen hinzufügt, sondern auch eine Referenz auf das Objekt haus zu den in der Rolle von Objekt mieter1 (hier: ToOneRole) verwalteten Referenzen. Das Trennen zweier Modellelemente erfolgt analog. Damit das möglich ist, bekommt eine Rolle bei ihrer Instanziierung ihren eigenen Namen übergeben. Die Zugriffsmethoden für die Rollen zweier an einer Assoziation beteiligten Klassen haben den gleichen Namen, wie die Rolle, die sie zurückgeben. In Abb. 3.7 (S. 25) sind das gerade heim und bewohner. Mit Hilfe des Reflection-Mechanismus von Java und dieser Namenskonvention ist es der Rolle *bewohner* möglich, über ein Objekt, das referenziert werden kann (hier mieter1 oder mieter2), auf dessen Rollenobjekt (die gegenüberliegende Rolle *heim*) zuzugreifen, um darauf eine Methode aufzurufen, die den Besitzer der Rolle *bewohner* (Objekt haus) zu den von der Rolle *heim* verwalteten Objekten hinzufügt bzw. entfernt.

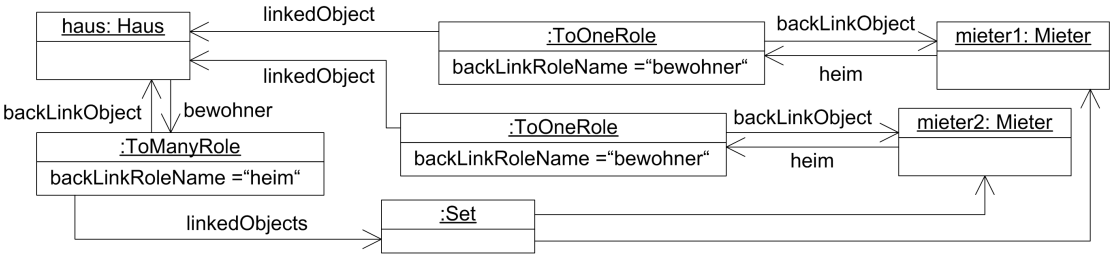

Abbildung 3.8.: Tatsächliche Objektstruktur zu dem Beispiel in Abb. 3.2

Die Assoziationsimplementierung – wie sie momentan vorliegt – ist ausschließlich für bidirektionale Assoziationen implementiert, kann aber durch eine einfache Anpassung auch auf Referenzen (*Anforderung 6*) erweitert werden. Dazu reicht es aus, bei einer unidirektionalen Assoziation zwischen den Klassen A und B, wobei B referenziert wird, die Methode der Klasse B, die bei einer bidirektionalen Assoziation die zugehörige Rolle zurückgeben würde, `null` zurückgeben zu lassen. B hätte also keine Rolle. Die Rollenimplementierung müsste zusätzlich so angepasst werden, dass in dem Fall, dass keine gegenüberliegende Rolle existiert (wie bei B), auch keine Rückverknüpfung beim Verbinden oder Trennen zweier Modellelemente erstellt wird (es wird keine Referenz von der B-Seite zur A-Seite erstellt).

Um die Semantik von Aggregationen bei der Abbildung auf Java-Code zu ermöglichen, implementieren die Rollenklassen eine parameterlose unlink-Methode, die – wie die in den Modellklassen implementierten removeAllFrom...-Methoden des in Abschnitt 3.2 beschriebenen Ansatzes (siehe S. 16) – alle Referenzen zu Modellelementen unter Einhaltung der Konsistenz bei bidirektionalen Assoziationen

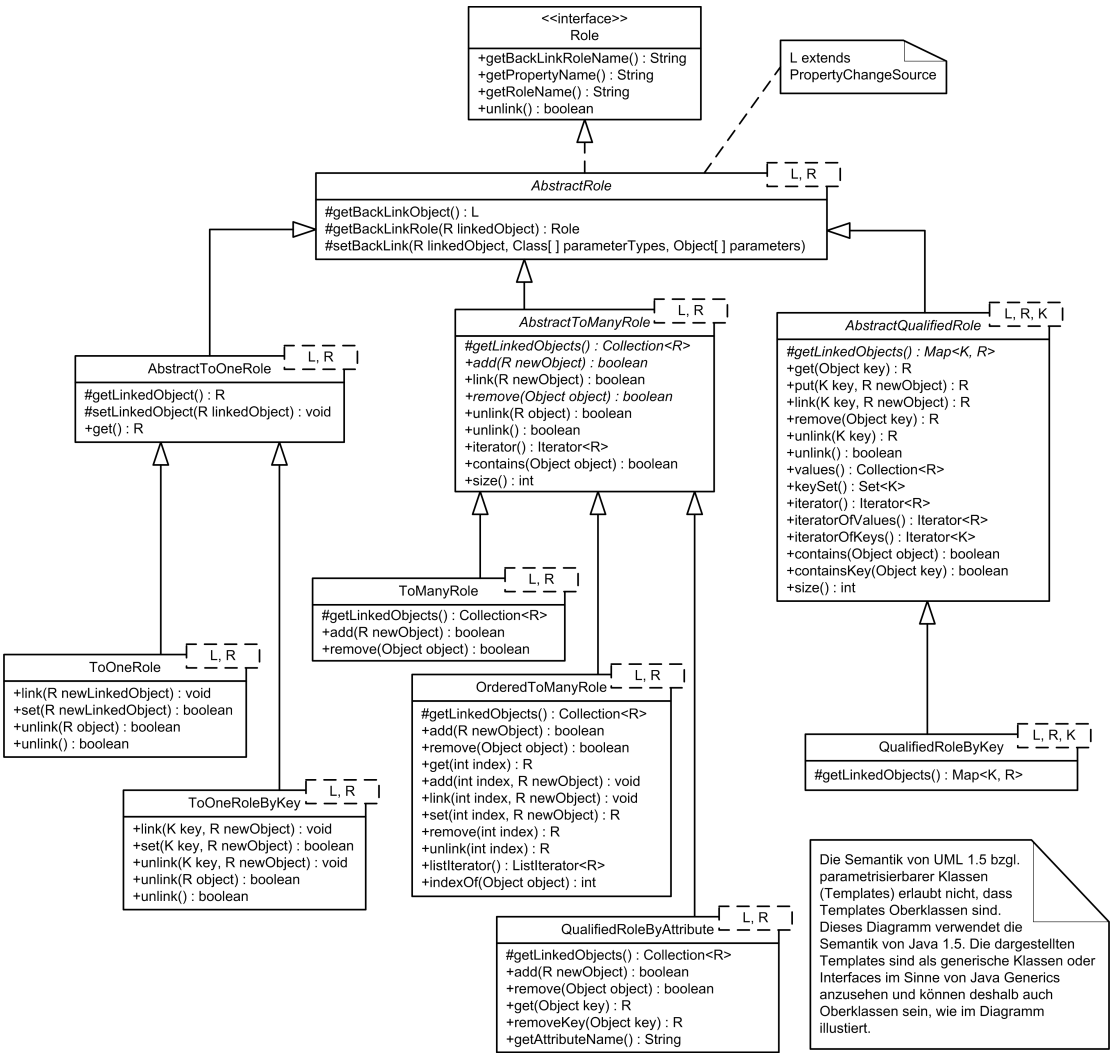

Abbildung 3.9.: Klassenhierarchie der Rollenbibliothek mit den wichtigsten Methoden und Attributen (Stand: November 2004, Vers. 0.4)

entfernt.

Rollenbibliothek

Da die Anzahl der referenzierten Elemente und die Anforderungen an ihre Verwaltung von der Art der Assoziation bzw. der Rolle abhängen, gibt es verschiedene Rollenimplementierungen, die entsprechende Eigenschaften erfüllen. Dabei wird zwischen zu-1- und zu-n- sowie qualifizierten Rollen unterschieden (*Anforderung 6*), es gibt z.B. die Klassen ToOneRole und ToManyRole und einige weitere für qualifizierte und geordnete oder sortierte Rollen.

Die Hierarchie der bisher verfügbaren Rollenklassen ist in vereinfachter Form (nur die wichtigsten Methoden sind dargestellt) in der Abb. 3.9 dargestellt. Wie die Hierarchie (Abb. 3.9) und die Abbildung 3.7 (S. 25) zeigen, werden die Rollen

mit den Typen der Modellelemente parametrisiert, die durch eine Assoziation verbunden werden können. In dem Beispiel aus Abb. 3.7 sind das gerade Haus und Mieter für die Typvariablen `L` und `R` in der Abb. 3.9.

Beim Hinzufügen oder Entfernen eines Elements bei qualifizierten Rollen reicht es nicht aus, nur das zu referenzierende oder zu entfernende Modellelement an die zuständige Methode zu übergeben. Die Qualifizierung verlangt einen Schlüssel, über den das Element erreicht werden kann. Dieser muss zusätzlich zum Modellelement an die entsprechende Methode als Parameter übergeben werden. Das gleiche gilt auch für die gegenüberliegende Seite, selbst wenn diese nicht qualifiziert ist, denn beim Verbinden oder Trennen zweier Modellelemente muss auch die Referenz in Rückrichtung erstellt bzw. gelöscht werden, um die Konsistenz zu erhalten. Dazu wird aber der Schlüssel als zusätzlicher Parameter benötigt.

Dieses Problem wird hier dadurch gelöst, dass bei einer einseitig qualifizierten Assoziation spezielle zu-1- und zu-n-Rollenklassen verwendet werden. Für das Beispiel in Abb. 3.5 (S. 16) würden die in der Abb. 3.9 dargestellten Rollenklassen QualifiedRoleByKey für Universität und ToOneRoleByKey für Student verwendet werden. Bei beidseitig qualifizierten Assoziationen werden sogar zwei Schlüssel benötigt, für je eine Seite der Assoziation einer. Wie mit beidseitig qualifizierten Assoziationen bei diesem Ansatz umgegangen wird, ist aus [Mai04] und [MZ04] nicht ersichtlich.

Verwendung von Containern

Bei der Implementierung in [Mai04] benutzen die Rollen, die mehrere Referenzen verwalten müssen, Container aus der Java-Bibliothek, z.B. HashSet oder LinkedList. Bei den zu-1-Rollen wird die einzig mögliche Referenz in einem Attribut gespeichert.

Funktionalität wie das Sortieren der referenzierten Elemente oder der Zugriff darauf über einen Schlüssel wird an die Container delegiert. Speziell für geordnete und sortierte Assoziationen gibt es nur eine Rollenklasse, nämlich OrderedToManyRole. Durch einen Parameter wird entschieden, ob der verwendete Container die Reihenfolge der verwalteten Elemente behalten (geordnete Rolle) oder sie nach einem bestimmten Kriterium sortieren soll (sortierte Rolle).

Typsicherheit

Der hier beschriebene Ansatz zur Assoziationsimplementierung ist zwar bereits mit Java Generics implementiert, kann aber keine Typsicherheit (*Anforderung 2*) garantieren. Das liegt zum einen an der Verwendung des Reflection-Mechanismus[8] von Java, der zur Laufzeit Informationen über eine Klasse beschaffen kann, nicht

[8] Man beachte, dass Java Generics so realisiert sind, dass alle Objekte einer generisch implementierten parametrisierten Klasse, auch wenn sie mit verschiedenen Typparametern initialisiert worden sind, immer die gleiche Klasse verwenden. Der Reflection-Mechanismus kann nur statische Informationen zur Klasse, ihren Methoden und Attributen beschaffen.

aber Informationen über die von einem Objekt verwendeten Typparameter, da diese zur Laufzeit nicht existieren. Zum anderen verwendet die bisher vorliegende Implementierung Typumwandlungen (*type casts*), die nur zur Laufzeit durchgeführt werden können und daher nicht typsicher sind. Die Typumwandlungen sind aber hauptsächlich aufgrund der Verwendung des Reflection-Mechanismus nötig, da dieser bei einem Methodenaufruf durch `java.lang.reflect.Method.invoke` als Rückgabetyp immer den allgemeinsten Java-Typ `java.lang.Object` verwendet.

Obwohl diese Implementierung keine vollständig statische Typprüfung erlaubt, ist es dennoch nicht möglich, Objekte falschen Typs in die Container einzufügen. Das liegt daran, dass die Rollenklassen nur Methoden zum Einfügen von Objekten des richtigen Typs haben. Bei einer Spezialisierung einer der Rollenklassen können die verwendeten Container-Objekte nur durch die definierten Zugriffsmethoden erreicht werden, denn die Container-Variablen sind durch `private`-Deklarationen in erbenden Klassen nicht sichtbar. Weil die Rollenklassen in einer Bibliothek liegen und deswegen nicht direkt verändert werden können, ist es einem Entwickler nicht möglich durch Hinzufügen weiterer Methoden in eine der Rollenklassen die Zugriffsmethoden zu umgehen. Bei dem in Abschnitt 3.2 beschriebenem Ansatz ist das möglich (siehe Seite 18).

3.3.2. Evaluation

Die Vorteile dieses Ansatzes sind vor Allem die bessere Benutzbarkeit (*Anforderung 4*), höhere Lesbarkeit (*Anforderung 3*) und der geringere Wartungsaufwand (*Anforderung 5*) durch deutlich kürzeren, übersichtlicheren und verständlicheren Code für die Modellklassen und die Auslagerung der Assoziationsmethoden in eigene dafür vorgesehene Rollenklassen. Die Klassenhierarchie der Rollen und die darin verwendeten Abstraktionen bzw. Spezialisierungen ermöglichen eine einheitliche Behandlung von Rollen mit gleicher Schnittstelle, was die Anwendungsentwicklung bequemer macht.

Durch die Auslagerung der Rollen in eine Bibliothek, ist ihre Wartung – solange sich die Schnittstellen nicht ändern – möglich, ohne dass die Implementierung der UML-Modelle angepasst oder neu generiert und kompiliert werden muss.

Der Hauptnachteil dieser Assoziationsimplementierung ist, dass keine Typsicherheit (*Anforderung 2*) garantiert werden kann. Man müsste die Verwendung von Reflection durch einen anderen flexiblen Mechanismus oder ein anderes Design ersetzen, um Typsicherheit garantieren zu können und gleichzeitig die Konsistenz der bidirektionalen Assoziationen zu erhalten (*Anforderung 1*).

Die Implementierung der Rollen (und damit auch der Assoziationen) ist durch die Verwendung von Reflection nur noch schwer lesbar und erschwert die Wartung der Rollenbibliothek.

Die Benutzung von Reflection hat auch noch einen anderen nicht zu vernachlässigenden Nachteil: Methodenaufrufe mit Hilfe von Reflection benötigen eine deutlich höhere Laufzeit, als gewöhnliche Methodenaufrufe. Der größte Aufwand entsteht dabei beim Heraussuchen der aufzurufenden Methode (Methoden-Lookup).

Bei mehr als einem Aufruf der gleichen Methode kann der Gesamtaufwand durch Caching minimiert werden. Dann muss die Methode nur ein Mal vor dem ersten Aufruf herausgesucht werden. Doch auch dann ist der Laufzeitbedarf höher als bei gewöhnlichen Methodenaufrufen (siehe Anhang A.2).

Die vorgestellte Assoziationsimplementierung ermöglicht eine Abbildung der meisten in Abschnitt 2.1 (S. 5) vorgestellten Assoziationen auf Java-Quellcode (*Anforderung 6*). Für beidseitig qualifizierte Assoziationen scheint es noch keine Unterstützung in [Mai04] zu geben.

Bei Iterationen können Laufzeitfehler auftreten, wenn sich währenddessen der zugehörige Container-Inhalt ändert. Dadurch ist die Benutzbarkeit dieser Implementierung eingeschränkt (*Anforderung 4*).

Thread-Sicherheit (*Anforderung 7*) ist bei der zur Zeit vorliegenden Implementierung [Mai04] nicht gegeben. Ein Benachrichtigungsmechanismus (*Anforderung 8*) ist zwar vorgesehen, ist aber momentan ebenfalls nicht verfügbar.

Die Rollenobjekte, die zur Laufzeit paarweise je eine Verbindung zwischen zwei Modellelementen entlang einer Assoziation repräsentieren, erhöhen den Speicherbedarf der modellierten und auf diese Weise implementierten Anwendung. Bei der Implementierung der Rollen innerhalb der Modellklassen (Abschnitt 3.2, ab S.13) sind keine weiteren Objekte (außer den Containern zur Verwaltung von referenzierten Objekten, die auch bei dieser Lösung verwendet werden) nötig.

Vorteile

- die Methoden zur Verwaltung der Assoziationen werden in eigene Klassen ausgelagert, der Code für Modellklassen wird übersichtlicher, erhöhte Lesbarkeit
- unabhängige Wartung der Assoziationsimplementierung möglich, solange die Schnittstellen sich nicht ändern; Modellimplementierung bedarf dann keinerlei Anpassung oder Neukompilierung
- Benutzung der Assoziationsimplementierung unter Anderem durch Abstraktion und gemeinsame Schnittstellen vereinfacht
- Trennung der Assoziationsimplementierung von der Implementierung der Modell-Klassen erziehlt ein besser strukturiertes Design
- Konsistenzerhaltung bei bidirektionalen Assoziationen

Nachteile

- es gibt keine Typsicherheit, diese kann auch nicht realisiert werden
- bei Iterationen können Laufzeitfehler auftreten
- Thread-Sicherheit und ein Benachrichtigungsmechanismus fehlen

- Reflection senkt die Lesbarkeit der Rollenimplementierung und der Wartungsaufwand für die Rollenbibliothek wächst
- geringe Performance der Modellimplementierung: höherer Speicherverbrauch durch Rollenobjekte, höhere Laufzeit durch Reflection

3.4. Fazit

In diesem Kapitel wurden Anforderungen an eine Assoziationsimplementierung formuliert. Es wurden der bei der Code-Generierung von Fujaba verwendete Ansatz und der in [MZ04] vorgeschlagene Ansatz bzgl. der Anforderungen evaluiert. Ebenso wurde diskutiert, wie die von Fujaba generierte Assoziationsimplementierung typsicher gemacht werden kann.

Es stellte sich heraus, das bei der vollständigen Implementierung der Assoziationen innerhalb der Modellklassen, wie es bei Fujaba der Fall ist, eine typsichere Assoziationsimplementierung realisierbar ist. Wegen der vielen generierten Methoden in den Modellklassen ist diese Implementierung aber sehr unübersichtlich und erschwert seine Benutzung und Wartung. Bei dem von Fujaba generierten Code ist bisher keine Typsicherheit gegeben, dafür ist aber z.B. ein Benachrichtigungsmechanismus verfügbar. Werden statt der speziellen Container die Standard-Container von Java verwendet, so wird die Implementierung zwar typsicher, aber dafür fehlt einige Funktionalität wie der Benachrichtigungsmechanismus.

Der in [MZ04] beschriebene Ansatz löst die Probleme der Lesbarkeit, vereinfacht die Benutzung und erlaubt eine Wartung der Assoziationsimplementierung unabhängig von einer Modellimplementierung (solange sich die Schnittstellen der Assoziationsimplementierung nicht ändern). Um die Konsistenz von bidirektionalen Assoziationen zu erhalten, wird hier der Reflection-Mechanismus von Java verwendet, wodurch aber Typsicherheit verloren geht. Außerdem wird die Performance einer Modellimplementierung durch Reflection beeinträchtigt. Es besteht keine Thread-Sicherheit und es fehlt Funktionalität wie ein Benachrichtigungsmechanismus.

4. Typsichere Implementierung von Assoziationen

Aufbauend auf den in Kapitel 3 vorgestellten Ansätzen zur Assoziationsimplementierung wird im Rahmen dieser Studienarbeit überprüft, ob und wie die Typsicherheit in der Implementierung von Assoziationen garantiert werden kann.

Dabei entstand ein neuer Implementierungsansatz, der auf der in [MZ04] entwickelten und in [Mai04] umgesetzten Idee (siehe auch Abschnitt 3.3, ab S. 23) basiert und Typsicherheit garantiert. Dieser Ansatz wird in diesem Kapitel vorgestellt und wie die anderen Ansätze bzgl. der in Kapitel 3 formulierten Anforderungen evaluiert.

4.1. Beschreibung des Ansatzes

In Kapitel 3 wurden zwei Möglichkeiten zur Implementierung von Assoziationen vorgestellt. Beide Ansätze weisen aber Mängel auf. Während bei dem ersten zu viele Methoden innerhalb der Modellklassen implementiert werden, kann der zweite keine Typsicherheit garantieren. Das Ziel des hier vorgestellten Ansatzes zur Implementierung von Assoziationen ist es, möglichst viele Vorteile der anderen Ansätze mit der Typsicherheit zu verbinden.

4.1.1. Überblick

Damit der Quellcode übersichtlich bleibt, der bei der Implementierung eines UML-Modells entsteht, wird hier die Idee aus [MZ04] aufgegriffen und die Assoziationsmethoden werden in Rollenklassen gekapselt.

Bei der Implementierung von bidirektionalen Assoziationen muss die Konsistenz immer aufrechterhalten werden. Bei den in Kapitel 3 vorgestellten Ansätzen ruft deswegen eine Methode zum Verbinden oder Trennen zweier Modellelemente, die auf einer Assoziationsseite aufgerufen wird, immer auch eine entsprechende Methode der gegenüberliegende Seite auf.

Damit das möglich ist, muss in irgendeiner Form auf die Methode der anderen Assoziationsseite zugegriffen werden. In einer generischen Rollenimplementierung werden anstatt der Typen der verbindbaren Elemente Typvariablen benutzt, wodurch keine Methoden der Modellklassen bekannt sind. Deswegen wird der Zugriff auf die Methoden bei dem Ansatz in [MZ04] (Abschnitt 3.3, S. 23) mit Hilfe von Reflection durchgeführt. Um Typsicherheit bei der Assoziationsimplementierung

zu ermöglichen, muss aber auf Reflection und Typumwandlungen verzichtet werden.

Bei dem hier vorgestellten Ansatz wird deshalb zusätzlich zu den allgemeinen generischen Rollenklassen für jede Assoziation ein Paar von speziellen Rollenklassen implementiert. Diese erben von den allgemeinen und legen die Typparameter fest. Durch die Belegung der Typparameter mit den Typen von an einer Assoziation beteiligten Modellelementen kann innerhalb der speziellen Rollenklassen auf die Methoden der Modellklassen direkt zugegriffen werden. Auf diese Weise kann Typsicherheit garantiert und gleichzeitig die Konsistenz von bidirektionalen Assoziationen durch gegenseitigen Methodenaufruf erhalten werden.

Die generischen allgemeinen Rollenklassen enthalten den größten Teil der Implementierung und werden als abstrakte Klassen in einer Bibliothek bereitgestellt. Die speziellen Rollenklassen werden für jedes Modell neu implementiert. Sie dienen nur dem typsicheren Zugriff von einer Rolle aus auf die ihr gegenüberliegende Rolle, um Methoden zum Verbinden oder Trennen zweier Modellelemente aufzurufen. Zur Verwaltung der verbundenen Modellelemente werden die typsicheren Standard-Container der Java-Bibliothek innerhalb der allgemeinen Rollenklassen verwendet.

4.1.2. Anpassung der Modellklassen

Wie bei dem in in Abschnitt 3.3 (S. 23) beschriebenen Ansatz erhält hier ein Modellelement für jede Assoziation, an der es beteiligt sein kann, ein an die Assoziation angepasstes Rollenobjekt. Dieses hält eine Referenz zum Modellelement und umgekehrt. Die Rollen werden in einer Getter-Methode der Modellklasse erstellt, während eine Setter-Methode nicht existiert.

Wie Abbildung 4.1 zeigt, ähnelt die Modellklassenimplementierung bei diesem Ansatz stark der generischen in Abschnitt 3.3 (Abb. 3.7, S. 25). Der Hauptunterschied ist die Verwendung von speziell für die Assoziation implementierten und bereits mit den an der Assoziation beteiligten Modellklassen parametrisierten Rollenklassen. Die Verwendung von Generics ist dadurch im Quellcode der Modellklassen nicht erkennbar.

Bei diesem Ansatz werden keinerlei Schnittstellen für die Modellklassen vorausgesetzt. Insbesondere müssen keine Methoden für die Realisierung eines Benachrichtigungsmechanismus implementiert werden, wie das bei dem Ansatz in Abschnitt 3.3 der Fall ist (vgl. Abb. 4.1 und Abb. 3.7, S. 25).

4.1.3. Implementierung spezieller Rollenklassen

Der Name der durch eine Spezialisierung von Rollenklassen aus der Rollenbibliothek entstehenden speziellen Rollenklassen ist irrelevant, aber um die Lesbarkeit zu erhöhen, bekommen sie den Namen der abstrakten Rollenklasse, die spezialisiert wird, und den eigenen Rollennamen sowie den der gegenüberliegenden Rolle als Präfix. Das erleichtert die Zuordnung zur zugehörigen Assoziation. Zum Beispiel

```
class Haus
{
   private Heim_Bewohner_NonQualifiedToManyRole bewohner = null;
   public final Heim_Bewohner_NonQualifiedToManyRole bewohner()
   {
      if (this.bewohner == null)
      {
         this.bewohner = new Heim_Bewohner_NonQualifiedToManyRole (this);
      }
      return this.bewohner;
   }
}

class Mieter
{
   private Bewohner_Heim_NonQualifiedToOneRole heim = null;
   public final Bewohner_Heim_NonQualifiedToOneRole heim()
   {
      if (this.heim == null)
      {
         this.heim = new Bewohner_Heim_NonQualifiedToOneRole (this);
      }

      return this.heim;
   }
}
```

Abbildung 4.1.: Typsichere Implementierung der Modellklassen zum Diagramm in Abb. 3.1 auf Seite 14

hat die Rolle Heim_Bewohner_NonQualifiedToManyRole (Abb. 4.1 und Abb. 4.2) der Klasse Haus in dem verwendeten Beispiel den eigenen Rollennamen *heim* gefolgt von dem der gegenüberliegenden Rolle *bewohner* und dem Rollenklassennamen *NonQualifiedToManyRole* als Namen.

Die Implementierung der speziellen Rollenklassen erfordert nur einen Konstruktor und das Implementieren der abstrakten Methode getOppositeRole. Beides bedarf zusammen nur weniger Zeilen Code, wie Abb. 4.2 zeigt. Der Rollenname wird bei diesem Ansatz nicht benötigt. Seine Übergabe an den Konstruktor der Oberklasse dient hier nur dazu, den Namen einer Rolle zu speichern und wieder auslesen zu können.

Die Methode getOppositeRole kann nicht in einer der allgemeinen generischen Klassen implementiert werden. Diese verwenden bei ihrer Imlementierung eine Typvariable, die den Typ der verwalteten Elemente repräsentiert (in den Abbildungen 4.3 auf Seite 36, 4.5 auf Seite 38 und 4.6 auf Seite 40 ist das `E`). Weil für diesen Typ keine Einschränkungen in Form einer Schnittstelle gemacht werden, sind die Methoden zur Rückgabe des Rollenobjekts zu einem Modellelement in diesen Klassen nicht bekannt. In dem Beispiel in Abb. 4.1 und Abb. 4.2 sind das die Methoden heim und bewohner, die auf einem Modellelement aufgerufen werden.

Alternativ könnte man eine Schnittstelle für die entlang einer Assoziation verbindbaren Typen festlegen. Diese würde z.B. eine Methode

```
public Role<E,O> getRole(String roleName)
```

```
public class Heim_Bewohner_NonQualifiedToManyRole extends NonQualifiedToManyRole<Mieter, Haus>
{
    public Heim_Bewohner_NonQualifiedToManyRole(Haus owner)
    {
        super("heim", owner);
    }
    protected NonQualifiedRole<Haus, Mieter> getOppositeRole(Mieter oppositeElement)
    {
        return oppositeElement.heim();
    }
}

public class Bewohner_Heim_NonQualifiedToOneRole extends NonQualifiedToOneRole<Haus, Mieter>
{
    public Persons_House_NonQualifiedToOneRole(Mieter owner)
    {
        super("bewohner", owner);
    }
    protected NonQualifiedRole<Mieter, Haus> getOppositeRole(Haus oppositeElement)
    {
        return oppositeElement.bewohner();
    }
}
```

Abbildung 4.2.: Typsichere Implementierung der Rollenklassen zum Diagramm in Abb. 3.1 auf Seite 14

definieren, die anhand des Rollennamens, der eindeutig ist, die entsprechende Rolle zurückgibt (`E` ist hier der Typ der referenzierten Elemente und `O` der Typ des Modellelements, zu dem die Rolle gehört). Legt man nun fest, dass die in den allgemeinen Rollenklassen verwendeten Typvariablen Typen repräsentieren, die diese Schnittstelle implementieren, so wird die Methode getRole auch in diesen Klassen bekannt.

Abgesehen von der Schnittstelle, die nun alle entlang einer Assoziation verbindbaren Klassen implementieren müssten, entsteht so ein anderes Problem: Die Methoden zum Verbinden und Trennen zweier Modellelemente benötigen abhängig davon, ob eine nicht qualifizierte, einseitig qualifizierte oder beidseitig qualifizierte Assoziation vorliegt, verschiedene Parameter. Bei einseitig qualifizierten Assoziationen wird zusätzlich zum Modellelement auch ein Schlüssel, unter dem dieses Element erreichbar ist, verlangt. Bei beidseitig qualifizierten Assoziationen wird

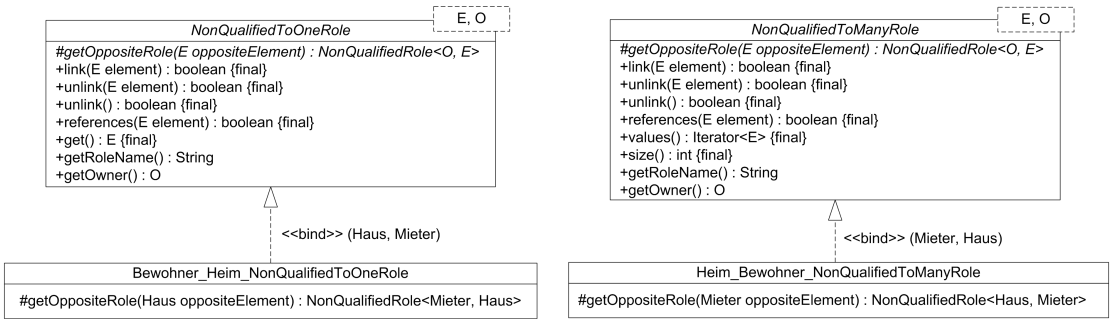

Abbildung 4.3.: UML-Klassendiagramm: Spezialisierung der abstrakten Rollenklassen zu Abb. 4.2

so ein Schlüssel wegen des gegenseitigen Methodenaufrufs zur Erhaltung der Konsistenz für beide Seiten benötigt. Definiert man für jeden der drei Fälle eine eigene Rollenklasse mit passenden Methoden und Parametern, so werden Typumwandlungen und `instanceof`-Abfragen im Quellcode der Rollenimplementierung notwendig, die wiederum nicht typsicher sind. Eine Alternative ist, für alle drei Fälle Methoden mit gleicher Signatur zu verwenden. Dadurch müsste man aber z.B. bei nicht qualifizierten Rollen immer zwei zusätzliche Parameter übergeben, die eigentlich nicht benötigt werden.

Als Ausweg werden spezielle Rollenklassen implementiert, bei denen durch die Festlegung der Typparameter der allgemeinen generischen Rollenklassen die Zugriffsmethoden für die Rollenobjekte zu einem Modellelement sichtbar werden. Anstatt für die Modellklassen die Implementierung einer Schnittstelle vorauszusetzen und Assoziationsmethoden mit gleicher Signatur für alle Assoziationsarten zu verwenden, werden für jede Assoziationsart spezielle Rollenklassen mit passenden Parametern in den Assoziationsmethoden definiert. Durch die Verwendung des richtigen Rollentyps, nämlich nicht qualifiziert, einseitig qualifiziert oder beidseitig qualifiziert, als Rückgabetyp der Zugriffsmethoden für eine Rolle, ist eine Typumwandlung nicht notwendig.

Bei dem Beispiel in Abb. 4.2 und Abb. 4.3 werden durch die Festlegung der Typparameter (hier Haus und Mieter) die allgemeiner definierten generischen Rollenklassen an die Assoziation, in der sie verwendet werden sollen, angepasst. Die Methode getOppositeRole verwendet hier die Klasse NonQualifiedRole für nicht qualifizierte Rollen als Rückgabetyp.

4.1.4. Realisierung der Assoziationen

Wie bei dem Ansatz in [MZ04] (Abschnitt 3.3, S. 23) kapseln die Rollenklassen die Implementierung von Assoziationen. Je zwei Rollen repräsentieren dabei eine Assoziation.

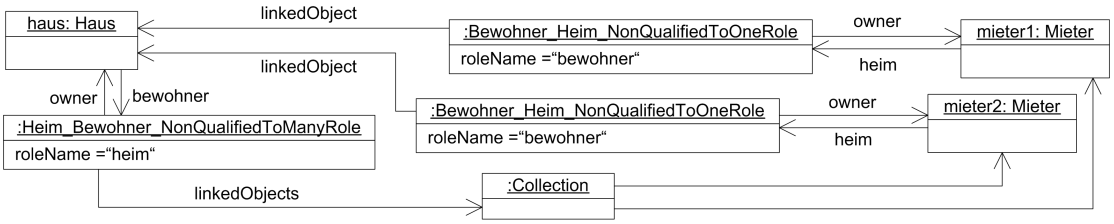

Abbildung 4.4.: Tatsächliche Objektstruktur zu dem Beispiel in Abb. 3.2 (S. 14)

Die Objektstrukturen, die zur Laufzeit bei Verwendung dieser Assoziationsimplementierung entstehen können, sehen – bis auf die Rollenklassen- und Attributnamen – identisch zu denen aus, die bei dem in Abschnitt 3.3 beschriebenen Ansatz entstehen. Zum Vergleich illustriert die Abb. 4.4 die Objektstruktur zu dem Beispiel in Abb. 3.2 auf Seite 14 (vgl. Abb. 3.8, S. 26).

Die Konsistenz bei bidirektionalen Assoziationen wird durch einen gegenseitigen Aufruf der Methoden zum Verbinden (link) oder Trennen (unlink) zweier Modellelemente sichergestellt. Zusätzlich zu den Methoden link und unlink wird auch die Methode references von allen Rollen mit den für die jeweilige Assoziationsart (nicht qualifiziert, einseitig qualifiziert oder beidseitig qualifiziert) passenden Parametern implementiert.

```
public final boolean link(E element)
{
   if (element == null)
   {
      throw new IllegalArgumentException("The parameter \'element\'must not be null.");
   }
   if (this.references(element))
   {
      return false;
   }
   this.referencedModelElements.add(element);
   NonQualifiedRole<O, E> oppositeRole = this.getOppositeRole(element);
   if (oppositeRole != null && !oppositeRole.references(this.getOwner()))
   {
      oppositeRole.link(this.getOwner());
   }
   return true;
}
```

Abbildung 4.5.: Implementierung der Methode link in der Klasse NonQualifiedToManyRole

Die Methode references dient der Überprüfung auf eine evtl. schon vorhandene Referenz vor der Verbindung mit (bzw. Trennung von) einem Objekt. Diese Prüfung ist wichtig für die Erhaltung der Konsistenz einer bidirektionalen Assoziation, denn ein Aufruf der link-Methode auf der einen Seite der Assoziation hat einen Aufruf der gleichen Methode auf der anderen Seite der Assoziation zur Folge (ebenso für unlink). Ohne die Abfrage auf eine evtl. schon vorhandene Referenz, würde dabei eine Endlosschleife entstehen. Zur Verdeutlichung wird die Methode link der Klasse NonQualifiedToManyRole in der Abb. 4.5 dargestellt (vgl. Methode addToBewohner in Abb. 3.4 auf S. 15).

Das Einfügen von `null`-Einträgen wird von den Rollenklassen durch eine Abfrage verhindert, denn eine Referenz zu `null` macht bei einer Assoziation keinen Sinn. Ist ein Parameter der link- oder unlink-Methoden `null`, so wird eine IllegalArgumentException geworfen.

Dieser Ansatz unterstützt sowohl uni- als auch bidirektionale Assoziationen (*Anforderung 6*). Bei unidirektionalen Assoziationen (Referenzen) gibt es nur Rollen auf der referenzierenden Seite (eine bei einer 1-zu-x- und n bei einer n-zu-x-Assoziation). Die Methode getOppositeRole der Rolle auf der referenzierenden Seite gibt einfach `null` zurück. Eine Konsistenzprüfung wie bei bidirektionalen Assoziationen ist hier nicht notwendig.

Die Abbildung von Aggregationen auf Java-Quellcode wird, wie bei dem in Abschnitt 3.3 beschriebenen Ansatz (siehe S. 26), durch eine parameterlose unlink-Methode in den Rollenklassen ermöglicht, die unter Einhaltung der Konsistenz bei

bidirektionalen Assoziationen alle Referenzen zu Modellelementen entfernt.

4.1.5. Rollenbibliothek

Die allgemeinen generischen Rollenimplementierungen liegen in einer Bibliothek vor. Alle Rollenklassen haben eine gemeinsame Oberklasse Role, die den Rollennamen und den Besitzer einer Rolle (das Modellelement, zu dem die Rolle gehört) speichert. Die Klasse Role erhält zwei Typvariablen: `E` für den Typ der referenzierten Elemente (*elements*) und `O` für den Typ des Modellelements, zu dem die Rolle gehört (*owner*).

Während bei dem Ansatz in Abschnitt 3.3 (S. 23) aus dem Papier [MZ04] und der bisherigen Implementierung [Mai04] nicht ersichtlich wird, wie mit beidseitig qualifizierten Assoziationen umgegangen wird, werden bei diesem Ansatz die beidseitig qualifizierten Assoziationen explizit berücksichtigt.

Bei nicht qualifizierten, einseitig qualifizierten und beidseitig qualifizierten Assoziationen werden verschiedene Anzahlen von Parametern benötigt. Deshalb werden für jeden der drei Fälle jeweils passende Rollenklassen mit nur einem, zwei oder drei Parametern in den betroffenen Methoden definiert. Dadurch ist die Klassenhierarchie der Rollen in drei Teilhierarchien unterteilt, mit den Klassen NonQualifiedRole (für nicht qualifizierte), SingleQualifiedRole (für einseitig qualifizierte) und DoubleQualifiedRole (für beidseitig qualifizierte Assoziationen) als Oberklassen (siehe Abb. 4.6). Diese bekommen weitere Typvariablen: `K` und `L` für die Typen der verwendbaren Schlüssel. So können sich bei beidseitig qualifizierten Assoziationen die Schlüsseltypen der beiden Seiten unterscheiden.

Die drei Klassen in dieser Ebene definieren die drei wichtigsten Methoden zum Verbinden und Trennen zweier Modellelemente mit den für die jeweilige Hierarchie passenden Parametern. Die Methoden sind link, unlink und references. Zusätzlich zu diesen drei Methoden wird die abstrakte Methode getOppositeRole definiert, die zu einer Rolle ihre gegenüberliegende Rolle liefern soll und bei der Spezialisierung der Rollenklassen (wie in Abb. 4.2 und Abb. 4.3 auf S. 36) implementiert werden muss. Durch die Verwendung des speziellen Rollentyps (NonQualifiedRole, SingleQualifiedRole oder DoubleQualifiedRole) als Rückgabetyp dieser Methode können Typumwandlungen in der generischen Implementierung der Methoden link und unlink vermieden werden.

In jeder der drei Teilhierarchien (nicht qualifiziert, einseitig qualifiziert, beidseitig qualifiziert) wird in der nächsten Hierarchieebene zwischen zu-1- und zu-n-Rollen unterschieden. Darin werden die zuvor abstrakt definierten Methoden link, unlink und references sowie einige weitere für zu-1- oder zu-n-Rollen typische Methoden, die das Aufzählen oder Durchlaufen aller referenzierten Elemente erlauben, implementiert.

Während bei nicht qualifizierten und beidseitig qualifizierten Assoziationen die Rollen auf beiden Seiten gleicher Art sind, nämlich beide nicht qualifiziert oder beide qualifiziert, entsteht bei den einseitig qualifizierten Assoziationen ein besonderer Fall: Hier ist nur eine der beiden Rollen qualifiziert, d.h. die eine Rolle greift

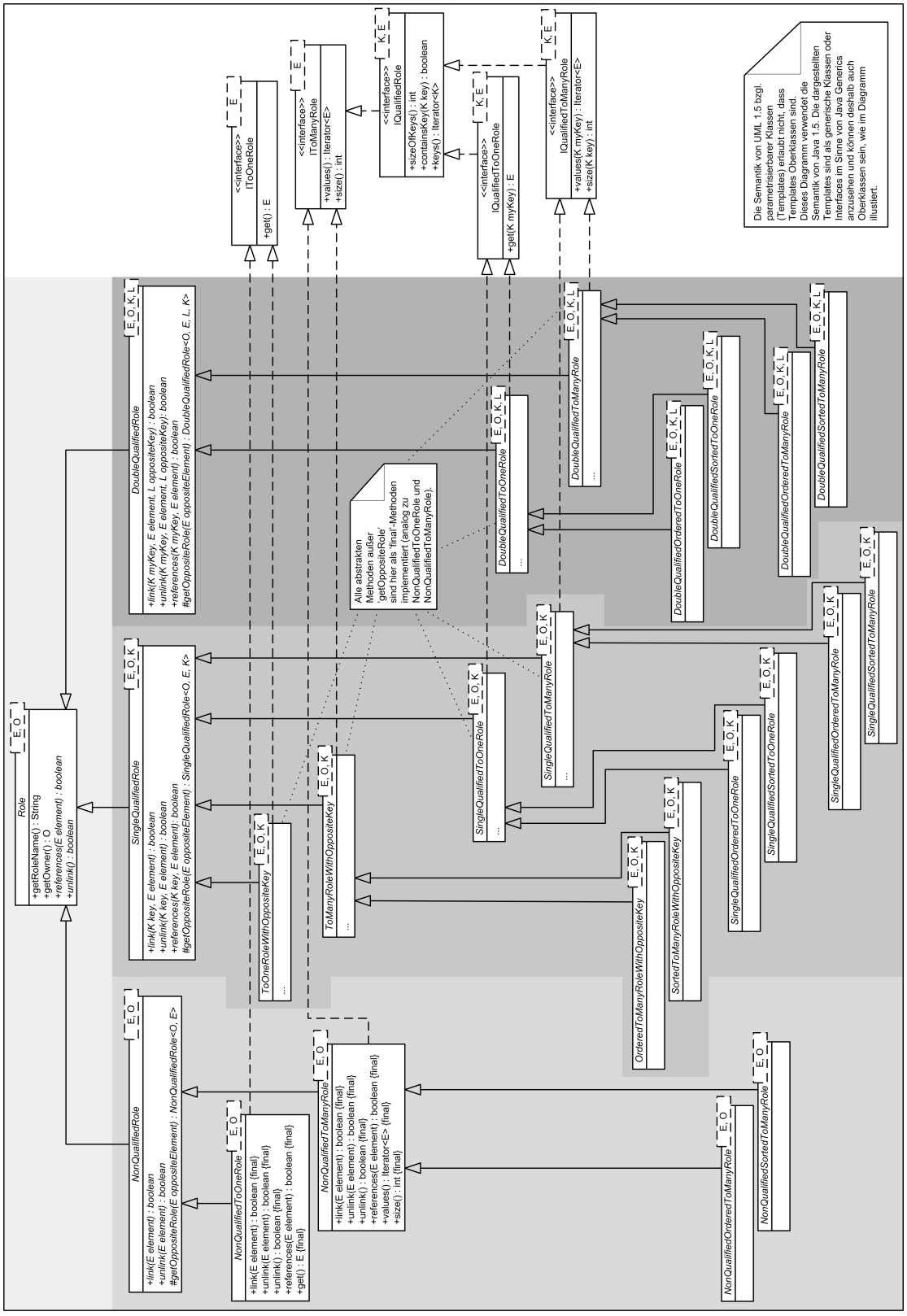

Abbildung 4.6.: Hierarchie der abstrakten Rollenklassen

auf die referenzierten Objekte über einen Schlüssel zu, die andere nicht. Dennoch haben die in der Klasse SingleQualifiedRole definierten Methoden für beide Rollenarten die gleiche Signatur, denn beide benutzen einen Schlüssel. Es werden also zwei zu-1- und zwei zu-n-Rollenklassen gebraucht, je eine für den qualifizierten und je eine für den nicht qualifizierten Fall. Diese Fälle werden durch die Klassen ToOneRoleWithOppositeKey und ToManyRoleWithOppositeKey für die nicht qualifizierte und die Klassen SingleQualifiedToOneRole und SingleQualifiedToManyRole für die qualifizierte Seite einer Assoziation abgedeckt.

Damit auch geordnete und sortierte Assoziationen (*Anforderung 6*) realisiert werden können, gibt es zu jeder der Rollenklassen, die mehr als ein Element referenzieren können, auch je eine Unterklasse für geordnete und sortierte Rollen. Das erlaubt sogar eine genauere Spezifikation von Assoziationen, denn die Constraints *geordnet* oder *sortiert* können für jede Rolle einzeln anstatt für die ganze Assoziation festgelegt werden. Die Klassen für sortierte oder geordnete Rollen unterscheiden sich von ihren Oberklassen hauptsächlich durch die Verwendung eines anderen Containers zur Verwaltung der referenzierten Modellelemente. Außerdem sind einige zusätzliche Methoden möglich, z.B. für das Festlegen der Sortierreihenfolge bei einer sortierten Rolle durch einen `java.util.Comparator` (das könnte in einem zusätzlichen Konstruktor passieren). Bei geordneten Rollen könnte das Einfügen oder das Abfragen eines Eintrags an einer bestimmten Stelle ermöglicht werden.

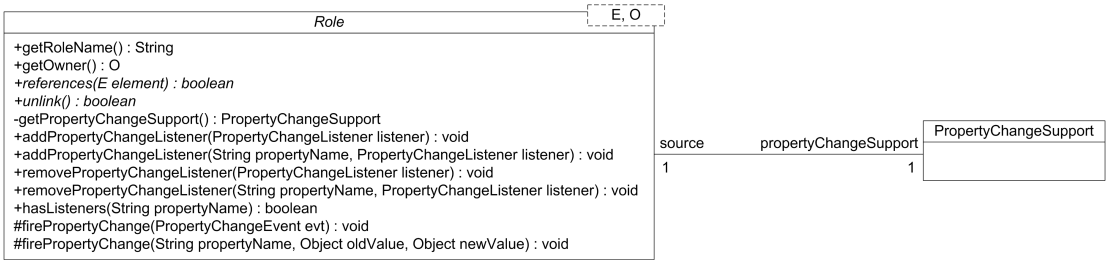

Abbildung 4.7.: Implementierung des Benachrichtigungsmechanismus

Ein Benachrichtigungsmechanismus (*Anforderung 8*) kann in der abstrakten Klasse Role implementiert werden. Diese erhält dann zusätzlich zu den in Abb. 4.6 (S. 40) dargestellten Methoden einige weitere, die den Mechanismus implementieren (siehe Abb. 4.7). Die Funktionalität wird durch die Delegation der Aufrufe an ein Objekt der Klasse `java.beans.PropertyChangeSupport` realisiert. Dieses Objekt wird nur bei Bedarf erzeugt.[1] Die Methoden addPropertyChangeListener, removePropertyChangeListener und hasListeners ermöglichen das Registrieren und Abmelden von Listener-Objekten. Eine der beiden Methoden firePropertyChange der Klasse Role wird bei Änderung des Container-Inhalts von den erbenden Klassen (z.B. NonQualifiedToManyRole) innerhalb der link- und unlink-Methoden auf-

[1] Bei einem Aufruf der Methode getPropertyChangeSupport wird ein PropertyChangeSupport-Objekt erstellt, wenn es noch nicht existiert.

gerufen, um die Listener zu benachrichtigen.

Die vorgestellte Klassenhierarchie für Rollen ist trotz der 24 Klassen übersichtlich und erlaubt eine einfache Auswahl und Spezialisierung der passenden Rolle für eine zu realisierende Assoziation (*Anforderung 4*). Dabei gibt es für jede der in Abschnitt 2.1 (S. 5) und der *Anforderung 6* (S. 12) genannten Assoziationen eine passende Rollenklasse. Ergänzend zu dieser Hierarchie sind Schnittstellen orthogonal zu den Hierarchieebenen definiert, die eine Benutzung dieser Assoziationsimplementierung weiter vereinfachen sollen.

4.1.6. Verwendung von Containern

Die verschiedenen Rollen müssen abhängig von der Art der Assoziation, die sie paarweise repräsentieren, die verbundenen Modellelemente in einer bestimmten Art und Weise verwalten. Zum Beispiel müssen die Elemente bei einer qualifizierten Assoziation über einen Schlüssel erreichbar sein. Bei einer zusätzlich sortierten Assoziation müssen die Einträge in sortierter Reihenfolge vorliegen.

Tabelle 4.1.: Verwendete Container

	nicht qualifiziert		qualifiziert	
	zu-1	zu-n	zu-1	zu-n
geordnet		LinkedList	LinkedHashMap	LinkedLists in LinkedHashMap
sortiert		TreeSet	TreeMap	TreeSets in TreeMap
weder noch	Attribut	HashSet	HashMap	HashSets in HashMap

Diese Funktionalität wird, um Wartungs- und Implementierungsaufwand zu sparen (*Anforderung 5*), an die Standard-Container der Java-Bibliothek (Paket `java.util`) delegiert, welche seit der Java-Version 1.5 typsicher sind und eine gute Basis für die Verwaltung der Modellelemente bieten. Bei nicht qualifizierten zu-1-Rollen reicht ein Attribut aus, um die einzig mögliche Objektrefrenz auf ein Modellelement zu speichern. In der Tabelle 4.1 werden alle verwendeten Container geordnet nach der Rollenart, in der sie verwendet werden, aufgelistet.

Bei der Iteration über die in einem der Standard-Container von Java enthaltenen Objekte können Laufzeitfehler (ConcurrentModificationExceptions) auftreten, wenn sich der Container-Inhalt während der Iteration (z.B. durch Hinzufügen oder Entfernen von Objekten) ändert. Um die Benutzbarkeit der Assoziationsimplementierung nicht einzuschränken, soll die Iteration in diesen Fällen sinnvoll fortgesetzt und Laufzeitfehler verhindert werden (*Anforderung 4*).

Eine Möglichkeit, dieses ohne eine Neuimplementierung der Standard-Container von Java – wie bei dem Ansatz in Abschnitt 3.2 (S. 13) – umzusetzen, ist, die Container oder ihren Inhalt vor jeder Iteration zu kopieren und die Kopie für die Iteration zu verwenden. Somit ist sichergestellt, dass jeder Iterator über ein eigenes exklusives Container-Objekt verfügt, über dessen Elemente dieser iteriert.

ConcurrentModificationExceptions können nicht mehr auftreten und mehrere nebenläufige Iterationen auf dem gleichen (vorher kopierten) Container werden dadurch möglich.

In Java können Objekte und damit auch die Container auf einfache Weise geklont werden, was auch den Container-Inhalt kopieren würde (genauer: nur die Objektreferenzen auf den Inhalt). Allerdings würde dabei Typsicherheit verloren gehen, weil die zum Klonen verwendete Methode `java.lang.Object.clone` nur ein Objekt vom Typ `java.lang.Object` zurückgibt und damit Typumwandlungen erfordert. Außerdem zieht das Klonen einen relativ hohen Speicher- und Laufzeitaufwand nach sich, wie in Anhang A.1 (ab S. 55) gezeigt wird.

Deswegen wird anstatt des gesamten Containers nur sein Inhalt kopiert (nur die Objektreferenzen darauf). Das passiert mit Hilfe des in Anhang A.1 (ab S. 55) vorgestellten und speziell für diesen Zweck implementierten typsicheren CollectionIterators. Dieser bekommt einen Container und erstellt sich eine Kopie des Container-Inhalts, die in einer ArrayList gehalten wird. Die ArrayList wird anschließend für die Iteration verwendet.

Dadurch, dass alle Rollenklassen in ihren Zugriffsmethoden anstatt des Iterators eines Containers einen CollectionIterator mit dem Container-Inhalt verwenden, kann eine Iteration auch dann fortgesetzt werden, wenn sich der Container-Inhalt geändert hat.

Die Schnittstelle für Iteratoren (`java.util.Iterator`) definiert die Methode `remove`, die das Entfernen der Elemente, über die iteriert wird, ermöglichen soll. Damit Konsistenzprüfungen (*Anforderung 1*) innerhalb der Zugriffsmethoden der Rollenklassen nicht umgangen werden können, ist der CollectionIterator so implementiert, dass ein Aufruf der Methode remove einen Laufzeitfehler erzeugt (UnsupportedOperationException) und damit ein Entfernen der Container-Einträge verbietet. Alternativ könnte der CollectionIterator (zumindest bei nicht qualifizierten Rollen) die Zugriffsmethode (unlink) der entsprechenden Rolle benutzen, um ein Element zu entfernen.

Um auf flexible Weise Thread-Sicherheit zu gewährleisten (*Anforderung 7*), werden die in den Rollen verwendeten Container bei Bedarf in speziellen typsicheren Wrappern gekapselt. Diese stellen sicher, dass nur exklusive Zugriffe auf die Container möglich sind.

Die Rollen werden so implementiert, dass wenn die verwendeten Container Thread-Sicherheit garantieren, dann auch die Rollen Thread-sicher sind. Ein Parameter im Konstruktor einer Rollenklasse bestimmt, ob die darin verwendeten Container in einem Thread-sicheren Wrapper gekapselt werden sollen, bevor sie verwendet werden.

Die Wrapper sind in der Klasse `java.util.Collections` des Java Collections Frameworks implementiert. Ein Methodenaufruf wie

```
Collections.synchronizedMap(myMap)
```

liefert den übergeben Container `myMap` in einem Wrapper zurück, dessen Zugriffsmethoden `synchronized` sind. Dadurch, dass der Container-Inhalt vor einer Itera-

tion darüber kopiert und nur über die in der Kopie enthaltenen Elemente iteriert wird, ist eine zusätzliche Synchronisation bei Iterationen nicht notwendig.

Bei nebenläufigen Anwendungen wird durch die Verwendung Thread-sicherer Container-Wrapper unter Anderem auch das Kopieren der Container-Inhalte durch exklusiven Zugriff geschützt. Dadurch werden Modifikationen des Containers während des Kopierens verhindert und Laufzeitfehler wie ConcurrentModificationExceptions können nicht auftreten. Bei rein sequentiellen Anwendungen ist eine Veränderung des Container-Inhalts während des Kopierens nicht möglich.

Die Standard-Container aus der Java-Bibliothek können auf einfache Weise innerhalb der Rollenklassen an die gestellten Assoziationsanforderungen angepasst werden. Der Wartungsaufwand beschränkt sich dadurch auf den der Rollenbibliothek (*Anforderung 5*). Die Typsicherheit bleibt mit Hilfe der generisch implementierten Container und Rollen erhalten.

4.1.7. Typsicherheit

Um Typsicherheit (*Anforderung 2*) bei dieser Implementierung garantieren zu können, wird auf die Verwendung des Reflection-Mechanismus von Java verzichtet und Typumwandlungen werden verhindert.

Dazu werden minimale aber an die Assoziation, in der sie verwendet werden sollen, angepasste Rollenklassen implementiert. Diese ermöglichen den typsicheren Zugriff auf die Rolle eines Modellelements, um durch Aufrufe von Zugriffsmethoden beider Assoziationsseiten die Konsistenz von bidirektionalen Assoziationen zu erhalten. Außerdem werden typsichere Container der Java-Bibliothek innerhalb der abstrakten Rollenklassen zur Verwaltung der durch eine Assoziation verbundenen Modellelemente verwendet.

4.2. Evaluation

Bei diesem Implementierungsansatz wird versucht, möglichst viele Vorteile der zuvor in Kapitel 3 vorgestellten Ansätze zusammen mit der Typsicherheit in einer Assoziationsimplementierung zu vereinen.

Als wichtigstes Ziel ist die Typsicherheit (*Anforderung 2*) erreicht worden. Auch die Konsistenz der bidirektionalen Assoziationen (*Anforderung 1*) bleibt immer gewahrt.

Durch eine Verlagerung der Assoziationsmethoden in eigene dafür vorgesehene Klassen wird die Modellklassenimplementierung übersichtlicher und erleichtert ihre Wartung (*Anforderung 5*). Klare Schnittstellen und eine einfache Anwendung (z.B. `a.rolename().link(b)` zum Verbinden zweier Modellelemente `a` und `b`) erhöhen die Lesbarkeit (*Anforderung 3*), vereinfachen die Benutzung (*Anforderung 4*) und erleichtern die Softwareentwicklung.

Der Wartungsaufwand für die Rollenbibliothek wird durch die Verwendung von bereits existierenden Containern der Java-Bibliothek reduziert. Mit Hilfe von eben-

falls existierenden Wrapper-Klassen kann ihre Funktionalität flexibel angepasst werden.

Durch die (nicht ganz vollständige) Auslagerung der Rollenklassen in eine Bibliothek ist eine unabhängige Wartung dieser Implementierung möglich, solange die Schnittstellen der Rollenklassen nicht verändert werden. Dann ist es nicht nötig, Modellimplementierungen an eine neue Bibliothek anzupassen.

Code-Redundanz – wie sie bei der Implementierung der Rollen innerhalb von Modellklassen (Abschnitt 3.2, S. 13) auftritt – wird durch die Implementierung gleicher Methoden in gemeinsamen Oberklassen verhindert.

Um Typsicherheit zu erhalten, ist es notwendig, die Rollenklassen der bereitgestellten Bibliothek zu spezialisieren. Auch wenn der Inhalt dieser spezialisierten Klassen klein ist, entsteht dennoch für jede Assoziation im Modell ein Paar neuer Rollenklassen. Das wiederum kann die Übersicht über alle Klassen erschweren. Um dem entgegen zu wirken, wird empfohlen, die spezialisierten Rollenklassen in ein anderes als das von den Modellklassen verwendete Paket zu verschieben.

Die Implementierung ist sowohl für unidirektionale als auch für bidirektionale Assoziationen verwendbar (*Anforderung 6*). Außerdem werden alle in Abschnitt 2.1 (S. 5) und *Anforderung 6* (S. 12) genannten Assoziationen berücksichtigt.

Mit Hilfe von speziellen Wrappern für Container kann bei diesem Ansatz auf flexible Weise Thread-Sicherheit garantiert werden (*Anforderung 7*), ohne unnötigen Laufzeit-Overhead für sequentielle Anwendungen zu erzeugen. Ein Benachrichtigungsmechanismus (*Anforderung 8*) ist ebenfalls verfügbar.

Bei diesem Ansatz besteht – wie bei dem Ansatz in [MZ04] (Abschnitt 3.3, S. 23) – ein erhöhter Speicherbedarf durch die Rollenobjekte, die paarweise je eine Verbindung zweier Modellelemente entlang einer Assoziation repräsentieren. Wrapper-Objekte und das Kopieren von Container-Inhalten erhöhen den Speicherbedarf und die Laufzeit zusätzlich. Dazu befindet sich eine Untersuchung in Anhang A.1.

Vorteile

- Typsicherheit
- die Methoden zur Verwaltung der Assoziationen werden in eigene Klassen ausgelagert, der Code für Modellklassen wird übersichtlicher, erhöhte Lesbarkeit
- unabhängige Wartung der Assoziationsimplementierung möglich, solange die Schnittstellen der Rollenklassen sich nicht ändern; Modellimplementierung bedarf dann keinerlei Anpassung oder Neukompilierung
- Benutzung der Assoziationsimplementierung unter Anderem durch Abstraktion und gemeinsame Schnittstellen vereinfacht

- Trennung der Assoziationsimplementierung von der Implementierung der Modell-Klassen erziehlt ein besser strukturiertes Design
- Konsistenzerhaltung bei bidirektionalen Assoziationen
- Thread-Sicherheit
- Benachrichtigungsmechanismus

Nachteile

- erhöhter Speicherverbrauch durch Rollenobjekte für jedes an einer Assoziation beteiligte Modellelement sowie durch Container-Kopien und Container-Wrapper
- für jede Assoziation müssen je zwei spezielle Rollenklassen implementiert werden

5. Technische Realisierung

Das UML Case Tool Fujaba [Fuj04] stellt außer der Unterstüzung für die Modellierung von statischen und dynamischen Teilen eines Softwaresystems mit Hilfe von UML-Diagrammen und speziellen so genannten Story-Diagrammen [FNTZ98] auch eine automatische Code-Generierung bereit. Fujaba kann Java-Code erzeugen. Die Code-Generierung von Fujaba ist aber so aufgebaut, dass auch die Generierung von Code in einer anderen objektorientierten Programmiersprache wie C++ prinzipiell möglich ist.

Im Rahmen dieser Studienarbeit wird unter Anderem untersucht, wie die bisher vorhandene Code-Generierung von Fujaba so erweitert werden kann, dass die Generierung von typsicherem Java-Code für Assoziationen in einem UML-Modell möglich wird. Als Grundlage für die Implementierung von Assoziationen dient dabei der neue, in dieser Studienarbeit erarbeitete und im Abschnitt 4.1 vorgestellte Ansatz.

Zu der technischen Realisierung dieses Ansatzes gehört zum einen die Implementierung der allgemein verwendbaren Rollenbibliothek und zum anderen die Entwicklung eines Plug-Ins zur Anpassung der Fujaba-Code-Generierung. Beides wird in diesem Kapitel beschrieben.

5.1. Implementierung einer Rollenbibliothek

Zu dem in Abschnitt 4.1 (ab S. 33) beschriebenen Ansatz wird eine Rollenbibliothek in Java implementiert. Diese enthält abstrakte Rollenklassen, die für eine konkrete Implementierung von Assoziationen zwischen zwei Modellklassen spezialisiert werden. Die Rollenimplementierung in der Bibliothek kann allgemein verwendet werden, unabhängig von Fujaba oder anderen Werkzeugen zur Code-Generierung.

Der nach dem in Abschnitt 4.1 vorgestellten Ansatz implementierte Code für Assoziationen in einem UML-Modell kann nur in Verbindung mit der Rollenbibliothek kompiliert und verwendet werden. Deswegen wird diese Bibliothek bei der neuen Code-Generierung in Fujaba verwendet.

Zur Zeit ist nur eine der drei Teilhierarchien in Abb. 4.6 (S. 40) implementiert worden, nämlich die für nicht-qualifizierte Rollen. In Zukunft soll die Rollenbibliothek vervollständigt werden.

5.2. Anpassung der Code-Generierung in Fujaba

In diesem Abschnitt wird erklärt, wie der bisherige Code-Generierungsmechanismus von Fujaba funktioniert. Anschließend wird die Entwicklung eines Plug-Ins für Fujaba beschrieben, das die bisher verwendete Java-Code-Generierung für Assoziationen in UML-Modellen anpasst.

Um die Generierung der statischen Anteile eines Modells anzupassen, ersetzt das Plug-In die bisherigen für die Implementierung von Assoziationen in die Modellklassen generierten Zugriffsmethoden und Container-Attribute durch neue an die Verwendung der Rollenobjekte angepasste Versionen. Es generiert spezielle an die modellierten Assoziationen angepasste Rollenklassen und speichert diese in einem neuen Paket namens *roles*. Damit auch die dynamischen Anteile die neue Assoziationsimplementierung verwenden, wird die Generierung von Methodenrümpfen aus Story-Diagrammen [FNTZ98] angepasst.

5.2.1. Der Code-Generierungsmechanismus

Der Mechanismus zur Code-Generierung in Fujaba ist flexibel aufgebaut. Alle in einem Diagramm vorkommenden Elemente liegen in einem so genannten *abstrakten Syntaxgraphen* (ASG) vor. Abhängig von der gewählten Zielprogrammiersprache werden diese Elemente mit Hilfe von speziellen an die Elemente und die Zielsprache angepassten *Handler*- und *Visitor*-Objekten behandelt. Dabei werden Stück für Stück Code-Fragmente zusammengesetzt und in entsprechende Dateien geschrieben.

Bei der Initialisierung des Code-Generierungsmechanismus werden die für eine gewählte Zielprogrammiersprache passenden Objekte zur Behandlung der ASG-Elemente instanziiert. Darunter befindet sich auch die in der Klasse CodeGenStrategy und ihren Unterklassen implementierte Strategie (*Strategy*-Entwurfsmuster nach [GHJV95]) zur Code-Generierung.[1]

Anschließend wird ein UMLProject-Objekt (ein Element des ASG), das ein gesamtes UML-Modell kapselt, zur Bearbeitung an die verwendete Strategie gereicht. Diese besitzt für die Code-Generierung aus ASG-Elementen eine Liste von ASG-Element-Handlern, eine Liste von CodeGenFunctions sowie für jede Zielsprache einen CodeGenVisitor (siehe Übersicht in Abb. 5.1). Während die Handler für die statischen Teile des UML-Modells zuständig sind, behandeln die CodeGenFunctions die dynamischen Teile, insbesondere Teile der Story-Diagramme (spezielle UML Aktivitäts- und Kollaborationsdiagramme) [FNTZ98].

Für jede Art von ASG-Elementen gibt es einen Handler, insbesondere gibt es auch einen für UMLProject-Objekte. Damit Code für ein ASG-Element generiert wird, reicht die Strategie das Element an den ersten Handler in der Liste. Die Liste ist implementiert nach dem Entwurfsmuster *Chain of Responsibility* nach

[1] Momentan wird ausschließlich die Klasse OOGenStrategyClient (Unterklasse von CodeGenStrategy) als Strategie verwendet, aber auch andere Strategien – vielleicht sogar für nicht objektorientierte Programmiersprachen – wären denkbar.

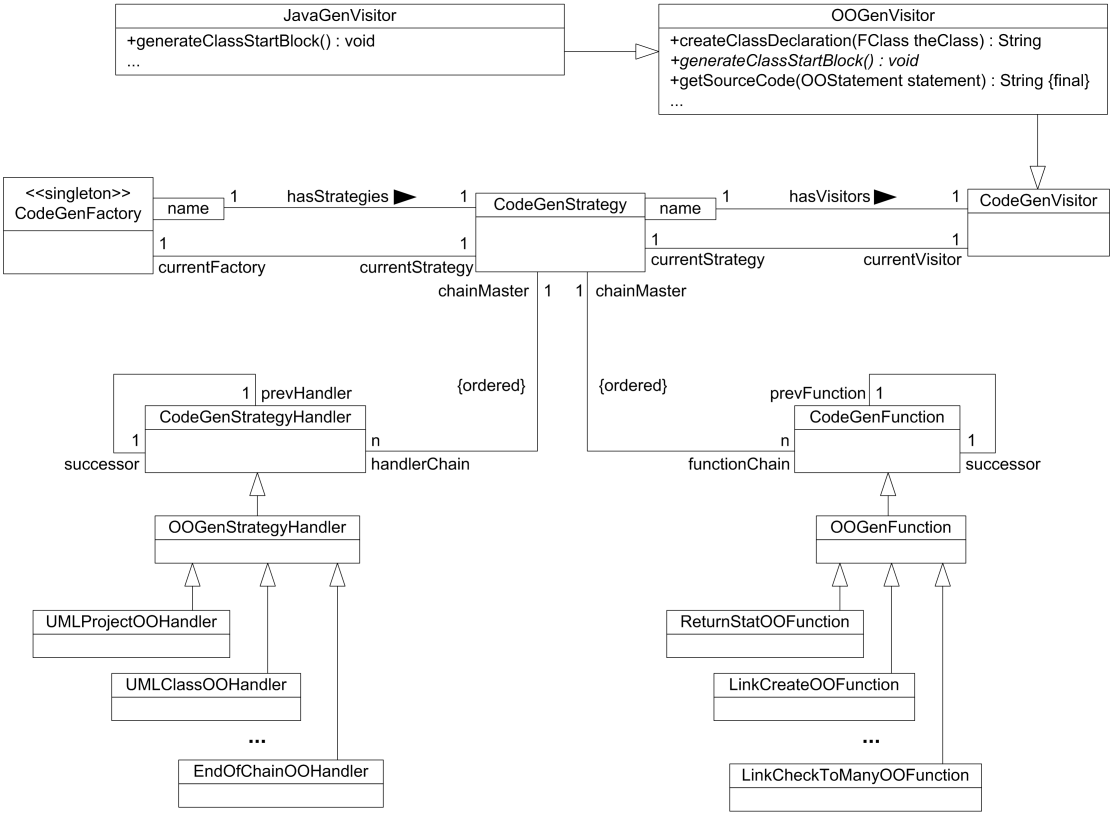

Abbildung 5.1.: Übersicht der an der Code-Generierung beteiligten Klassen (alle aus dem Paket `de.uni_paderborn.fujaba.codegen`)

Gamma et al. ([GHJV95]). Der Handler prüft, ob er für diese Art von Elementen zuständig ist und reicht das Element an den nächsten Handler in der Liste weiter bis ein zuständiger Handler gefunden wurde. Bei der Verarbeitung eines ASG-Elements werden, wenn nötig, Konsistenzprüfungen durchgeführt und darin enthaltene Elemente extrahiert. Diese werden ebenfalls an die Strategie zur Bearbeitung weitergegeben. Auf diese Weise werden alle Elemente des ASG beginnend mit einem UMLProject-Objekt nach und nach von einem zuständigen Handler bearbeitet. Ähnlich wie von den Handlern, werden die ASG-Elemente auch von den CodeGenFunctions behandelt, die ebenfalls als *Chain of Responsibility* organisiert sind. Die CodeGenFunctions sind hauptsächlich für die Generierung der Methodenrümpfe zuständig.

Die Handler und CodeGenFunctions verwenden die Strategie und diese wiederum den CodeGenVisitor, um für ein ASG-Element Code zu generieren. Für jede der unterstützten Zielprogrammiersprachen gibt es je eine Unterklasse von CodeGenVisitor. Jede davon bietet verschiedene Methoden zur Generierung von bestimmten Ausdrücken in der Zielsprache. Die CodeGenVisitor-Implementierungen ermöglichen die Generierung dieser Ausdrücke, ohne die Syntax der Programmiersprache zu kennen. Unter Verwendung dieser Methoden wird für jedes der

ASG-Elemente von den Handlern und den CodeGenFunctions Code generiert. Die CodeGenVisitor-Objekte puffern die generierten Code-Fragmente und schreiben ihn schließlich in die entsprechenden Dateien.

Eine genauere Beschreibung des Code-Generierungsmechanismus befindet sich in [Moa02].

5.2.2. Das Plug-In für die neue Code-Generierung

Für die Implementierung von Assoziationen nach dem neuen Ansatz aus Abschnitt 4.1 (ab S. 33) wird die bisherige Code-Generierung von Fujaba durch ein Plug-In angepasst. Die Anpassung betrifft dabei sowohl die Generierung von Code für statische als auch für dynamische Anteile eines modellierten Softwaresystems. Das implementierte Plug-In unterstützt momentan nur nicht-qualifizierte zu-1- und zu-n-Rollen (darunter auch sortierte und geordnete).

Die bisherige Code-Generierung bietet bereits eine Unterstützung für die Generierung von Java-Code für Assoziationen. Nach dem Ansatz aus Abschnitt 3.2 (ab S. 13) werden dabei verschiedene Methoden zum Verbinden und Trennen zweier Modellelemente in die an einer Assoziation beteiligten Modellklassen generiert.

Zuständig für die Generierung dieser Methoden ist hauptsächlich der UMLRoleOOHandler. Dieser bekommt das UMLRole-Objekt einer an einer Assoziation beteiligten Klasse (UMLClass) und erzeugt abhängig von der Art der Assoziation neue UMLAttr- und UMLMethod-Objekte für die Assoziationsimplementierung. Diese neuen ASG-Elemente werden zur weiteren Behandlung an die Code-Generierungsstrategie weitergereicht, werden aber nach der Code-Generierung wieder entfernt, um das Modell nicht zu ändern.

Um die Generierung an die neue Assoziationsimplementierung anzupassen, werden einige Handler (insbesondere UMLRoleOOHandler) für die statischen und CodeGenFunctions für die dynamischen Anteile eines modellierten Softwaresystems durch angepasste Versionen davon ausgetauscht. Damit auch parametrisierte Typen wie Rollenklassen durch ein ASG-Element beschrieben werden können, werden neue ASG-Elemente implementiert. Schließlich wird der CodeGenVisitor so angepasst, dass dieser auch Code für die neuen ASG-Elemente generieren kann.

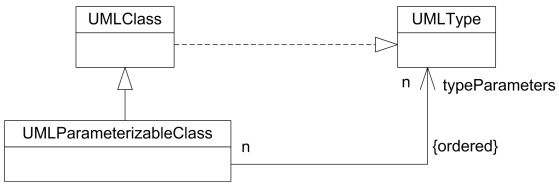

Abbildung 5.2.: Spezialisierung von UMLClass für parametrisierte Typen

Die vorhandene Code-Generierung kann nicht mit generischen Klassen und Methoden umgehen. Damit für jede Assoziation spezielle generische Rollenklassen generiert werden können, wird ein neues ASG-Element eingeführt, das in der Klasse

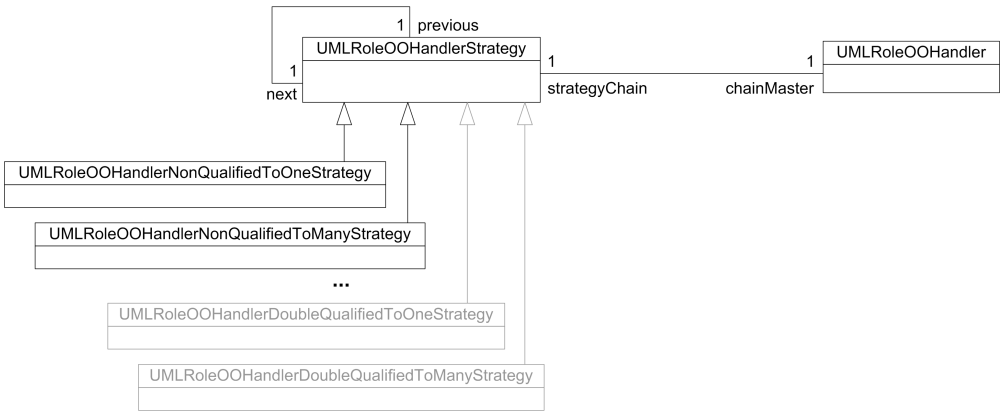

Abbildung 5.3.: Implementierung der Rollen-Strategien

UMLParameterizableClass implementiert wird (siehe Abb. 5.2). Dieses Element repräsentiert einen parametrisierten Typ (z.B. eine Klasse mit Typparametern) oder eine gewöhnliche Klasse. Die einzige neue Funktionalität gegenüber der Klasse UMLClass ist das Merken der Typparameter, wenn welche festgelegt wurden. Da bei der Generierung von Rollenklassen keine Methoden mit Typvariablen generiert werden müssen, werden vorerst keine ASG-Elemente für Typvariablen implementiert.

Bei der Generierung von dynamischen Teilen eines Modells wird ein spezieller abstrakter Syntaxgraph von dem OOGenStrategyClient (Code-Generierungsstrategie für objektorientierte Programmiersprachen) und dem OOGenVisitor verwendet. Typen werden darin durch Objekte der Klasse OOType anstatt UMLClass oder UMLType repräsentiert. Deswegen wurde für die Verwendung von parametrisierten Typen innerhalb von Methodenrümpfen die Klasse OOType durch die Klasse OOParameterizedType analog zu UMLParameterizableClass erweitert.

Ein UMLRoleOOHandler hat eine Liste von Strategien (siehe Abb. 5.3), ähnlich der Handler-Liste der CodeGenStrategy. Für jede Rollenart, z.B. nicht qualifizierte zu-n-Rolle, gibt es je eine Strategie, die für die Bearbeitung des UMLRole-Objekts zuständig ist. (Bisher sind nur Strategien für nicht-qualifizierte Rollen implementiert.) Der UMLRoleOOHandler und seine Strategien werden so angepasst, dass anstatt der Assoziationsmethoden innerhalb der Modellklassen nun die Rollenklassen und die zugehörigen Zugriffsmethoden generiert werden (siehe auch Abb. 4.1, S. 35 und Abb. 4.2, S. 36). Dazu werden für die entsprechenden Methoden, Attribute und Rollenklassen neue ASG-Elemente erstellt und zur weiteren Behandlung an die Code-Generierungsstrategie weitergegeben. Damit die erzeugten Elemente nach der Code-Generierung aus dem Modell wieder entfernt werden können, wird der UMLProjectOOHandler so angepasst, dass er sie zwischenspeichert und nach der Code-Generierung entfernt. Zusätzlich sammelt dieser alle zu generierenden Dateien ein – nämlich die für die Rollenklassen – und generiert den zugehörigen Code nachdem alle anderen Dateien generiert wurden.

Auch die CodeGenFunctions werden angepasst, damit innerhalb der generier-

ten Methodenrümpfe die neuen Zugriffsmethoden und Rollen benutzt werden. Außerdem können aufgrund der typsicheren Assoziationsimplementierung bisher notwendige Typumwandlungen (*type casts*) weggelassen werden. Die verwendeten Iteratoren sind parametrisiert mit dem Typ der Elemente, über die iteriert wird. Diese Typparameter müssen ebenfalls bei der Code-Generierung ergänzt werden.

Der CodeGenVisitor wird erweitert, damit dieser mit parametrisierten Typen umgehen kann. Insbesondere wird die Methode createClassExtendsDeclaration überschrieben, um bei einer Klassendeklaration mit einem `extends`-Ausdruck auch Typparameter angeben zu können. Das ist für die Generierung der Rollenklassen notwendig, wie die Abb. 4.2 (S. 36) zeigt.

Schließlich wird eine Plug-In-Klasse implementiert, die bei der Initialisierung den CodeGenVisitor, die Handler und die CodeGenFunctions durch angepasste Versionen ersetzt.

Der generierte Code kann zusammen mit der in Abschnitt 5.1 (S. 47) beschriebenen Rollenbibliothek kompiliert und verwendet werden.

6. Zusammenfassung und Ausblick

In dieser Studienarbeit wurden verschiedene Ansätze zur Implementierung von Assoziationen mit dem Ziel der Code-Generierung aus einem UML-Modell vorgestellt und evaluiert. Die bisher verwendete Fujaba-Code-Generierung für Assoziationen und ein neuer an der Universität Kassel entwickelter Ansatz [MZ04] wurden verglichen und auf die Realisierung von Typsicherheit hin überprüft. Dabei stellte sich heraus, dass die Fujaba-Code-Generierung zwar typsicher gemacht werden könnte, der erzeugte Code aber durch sehr viele generierte Methoden innerhalb der Modellklassen nur schwer lesbar und wartbar ist. Obwohl der Ansatz aus [MZ04] bereits Java Generics verwendet, kann bei dieser Art der Assoziationsimplementierung aufgrund der Verwendung des Reflection-Mechanismus keine Typsicherheit erreicht werden. Es zeigten sich aber deutliche Vorteile bzgl. Wartbarkeit und Lesbarkeit des generierten Codes.

Aufbauend auf der Idee aus [MZ04] wurde eine typsichere Implementierung von Assoziationen erarbeitet und beschrieben. Sie erfüllt alle genannten Anforderungen, insbesondere wird die Konsistenz bei bidirektionalen Assoziationen gewahrt und die Wartbarkeit und Lesbarkeit des generierten Codes erhöht. Für einen praktischen Test wurde ein Fujaba-Plug-In implementiert, das alle nicht-qualifizierten in Fujaba modellierbaren Assoziationen unterstützt.

In Zukunft kann die zur Zeit nicht vollständig implementierte Rollenbibliothek um die Rollenklassen für einseitig qualifizierte und beidseitig qualifizierte Assoziationen erweitert und der Öffentlichkeit zugänglich gemacht werden. Da der Ansatz Fujaba-unabhängig ist, kann die Bibliothek auch ohne Code-Generierung oder innerhalb einer anderen Umgebung als Fujaba verwendet werden.

Die bisher als Plug-In implementierte typsichere Assoziationsimplementierung könnte komplett in den Code-Generierungsmechanismus von Fujaba integriert werden. Da große Teile von Fujaba mit Fujaba entwickelt wurden, ist auch eine Neugenerierung des vorhandenen Fujaba-Codes unter Verwendung der neuen Assoziationsimplementierung denkbar. Dadurch könnte der Fujaba-Code typsicher, aber auch übersichtlicher und lesbarer gemacht werden. Der Aufwand für die Wartung des Fujaba-Codes könnte reduziert werden und Typfehler bei der Implementierung von Assoziationen verhindert werden.

In einem nächsten Schritt könnte die Modellierung von generischen Klassen und Methoden innerhalb von Fujaba realisiert werden. Dadurch wäre eine Generierung von vollständig[1] typsicherem Code aus einem Modell möglich.

[1] Das realisierte Plug-In ermöglicht die Generierung von einem nur für Assoziationen typsicheren Code .

A. Anhang

In diesem zusätzlichen Kapitel werden einige Fragen bzgl. Performance bei der Implementierung von Assoziationen geklärt.

In dem ersten Abschnitt wird untersucht, wie hoch der Laufzeit- und Speicheraufwand für das Erzeugen von Iteratoren ist, die auch bei Veränderung des Container-Inhalt während einer Iteration keine Laufzeitfehler (ConcurrentModificationExceptions) erzeugen. Das ist insbesondere für die Verwendung der Standard-Container aus der Java-Bibliothek wichtig, wie das z.B. bei dem in Abschnitt 4.1 (ab S. 33) beschriebenen Ansatz der Fall ist.

Bei dem zweiten Abschnitt wird der Aufwand für Methodenaufrufe mit Hilfe des Reflection-Mechanismus von Java dem Aufwand für direkte Methodenaufrufe gegenübergestellt. Dieser Vergleich ergänzt die Evaluation des Ansatzes zur Assoziationsimplementierung in Abschnitt 3.3 (ab S. 23).

Alle in diesem Kapitel vorgestellten Tests sind in Java implementiert. Dabei wird die Version 1.5 verwendet (Java 2 SE 5 bzw. Java Development Toolkit 1.5). Um Daten, wie Speicherverbrauch und Laufzeit der Tests, zu sammeln, wird das Tool *JProfiler* in der Version 3.1.2 der Firma ej-Technologies verwendet [ej-04].

Die Evaluation einer Anwendung mit JProfiler erfordert, dass diese Anwendung noch aktiv (nicht beendet) ist. Aus diesem Grund wird in allen Tests am Ende des Test-Programms ein AWT-Fenster[1] erzeugt, das die Programmausführung für die Dauer der Evaluation hinauszögert. Dadurch ist der gemessene Speicherbedarf evtl. etwas erhöht.

A.1. Aufwand für das Erzeugen von Iteratoren mit exklusivem Kontext

Bei der Implementierung von Assoziationen wird meist Gebrauch von Containern gemacht. Diese dienen der Verwaltung von sich gegenseitig referenzierenden Objekten. Im Zusammenhang mit Containern werden auch Iteratoren verwendet (`java.util.Iterator`), um die enthaltenen Elemente zu durchlaufen. Dabei können ConcurrentModificationExceptions auftreten, falls sich der Inhalt eines Containers während der Iteration ändert, z.B. wenn ein Element hinzugefügt wird.

[1] *AWT* steht für „Abstract Window Toolkit“. Das ist ein Teil der Java-Bibliothek und stellt diverse Hilfsmittel zur Programmierung von grafischen Benutzungsschnittstellen zur Verfügung. Hier wurde ein Frame aus dem Paket `java.awt` verwendet.

```
public class TestPerformance
{
   private static HashMap<Integer, String> map = new HashMap<Integer, String>();
   public static void main(String[] args)
   {
      // Initialisieren der Variablen...

      // Einfügen der Einträge in die HashMap...

      // TEST: Erzeugen der Iteratoren...

      // AWT-Fenster erstellen, damit das Programm nicht gleich beendet wird
      Frame f = new Frame();
      f.setVisible(true);
      f.addWindowListener(new WindowAdapter()
         {
            public void windowClosing(WindowEvent e)
            {
               System.exit(0);
            }
         });
   }
}
```

Abbildung A.1.: Aufbau eines Testprogramms

Um diese Exceptions in sequentiellen Anwendungen zu vermeiden,[2] ist es möglich, das Container-Objekt zu klonen und einen Iterator des geklonten Containers für die Iteration zu verwenden. Dadurch erhöhen sich der Speicherverbrauch und die Laufzeit, was in diesem Abschnitt genauer untersucht wird.

Um den Container-Inhalt bei jeder Iteration exklusiv zu benutzen, können die Container-Objekte vor der Iteration geklont und anstatt des Originals für die Iteration verwendet werden. Es wurde aber auch eine weitere Möglichkeit erarbeitet, die ebenfalls eine Iteration über exklusive Container-Inhalte erlaubt. In diesem Fall wird nicht ein gesamter Container geklont, sondern nur sein Inhalt kopiert (genauer: nur die Referenzen auf den Inhalt). Dadurch soll der Speicherbedarf reduziert werden.

Realisiert wird das mit Hilfe einer typsicheren Implementierung der Iterator-Schnittstelle von Java. Der CollectionIterator (eigene Entwicklung) bekommt einen Container vom Typ `java.util.Collection` übergeben, dessen Inhalt beim Erzeugen des Iterators in eine vom Iterator exklusiv gehaltene ArrayList (Paket `java.util`) kopiert wird.

Die `clone`-Methode gibt immer nur ein Objekt vom Typ `java.lang.Object` zurück und nicht vom speziellen tatsächlich geklonten Typ. Dadurch werden Typumwandlungen (*type casts*) nötig. Diese verhindern aber Typsicherheit, da die Typprüfung zur Laufzeit erfolgt.

Um Aussagen zu Speicherverbrauch und Laufzeit machen zu können, wurden einige kurze Java-Programme geschrieben und mit JProfiler evaluiert. In jedem

[2] In nebenläufigen Anwendungen bedarf es eines weiteren Mechanismus, um ConcurrentModificationExceptions zu vermeiden. Damit sich der Container-Inhalt während des Klonvorganges nicht ändert (z.B. durch das Einfügen oder Entfernen von Objekten durch andere Threads), muss das Klonen des Containers durch einen exklusiven Zugriff geschützt werden.

Test wird eine HashMap (Paket `java.util` mit mehreren Einträgen (Integer als *Key* und String als *Value*) erzeugt. Anschließend wird eine bestimmte Anzahl von Iterator-Objekten für die Iteration über die Value-Einträge der HashMap erzeugt und in einem Array oder einer Variable gespeichert. Mit JProfiler wird die Laufzeit für das Kopieren des Container-Inhalts (bzw. Klonen des Containers) und das Erzeugen der Iterator-Objekte gemessen. Zusätzlich wird der verbrauchte Speicher direkt vor dem Erstellen der Iterator-Objekte und danach festgehalten. Die Abb. A.1 zeigt den Aufbau eines solchen Testprogramms.

Für die Testergebnisse werden nur die relevanten Werte betrachtet. Für die Ermittlung des Speicherverbrauchs werden nur die Objekte betrachtet, die zusammen den meisten Speicher verbrauchen oder eindeutig von den verwendeten Datenstrukturen instanziiert werden. Beim Laufzeitvergleich wird nur die Laufzeit der beim Test verwendeten Methoden betrachtet, sodass die Einfüge-Operationen in die HashMap sowie das Erstellen des AWT-Fensters keine Auswirkung haben. Es wird immer die gesamte Laufzeit betrachtet, d.h. wenn eine Methode zehn Mal aufgerufen wurde, wird die Laufzeit für diese zehn Aufrufe zusammen verwendet. Es werden nicht alle gesammelten Ergebnisse in dieser Arbeit präsentiert, da das den Rahmen sprengen würde. Stattdessen werden zu jedem der Tests nur die Laufzeitergebnisse präsentiert und einige Hinweise bzgl. des Speicherverbrauchs gegeben. Anschließend (ab S. 61) werden der Gesamtspeicherbedarf und die Laufzeit verglichen, allerdings ohne ins Detail zu gehen.

Mit jeder der beiden genannten Methoden (Klonen und CollectionIterator) werden drei Tests durchgeführt. Bei dem ersten Test enthält die HashMap 100.000 Einträge und es wird ein Iterator erstellt. Bei dem zweiten Test hat die HashMap 100.000 Einträge und es werden 10 Iterator-Objekte erstellt. Beim dritten Test enthält die HashMap 10 Einträge, es werden aber 100.000 Iterator-Objekte erstellt.

In beiden Ansätzen werden ausschließlich der Container-Inhalt und zugehörige

```
// Initialisieren der Variablen
Iterator<String> iter = null;


// Einfügen der Einträge in die HashMap
for (int i = 0; i < 100000; i++)
{
   map.put(new Integer(i),
      Integer.toString(i));
}

// TEST: Erzeugen der Iteratoren
iter = ((HashMap) map.clone())
   .values().iterator();

// AWT-Fenster erstellen...
```

```
// Initialisieren der Variablen
HashMap<Integer, String>[] maps = new HashMap[10];
Iterator<String>[] iterators = new Iterator[10];

// Einfügen der Einträge in die HashMap
for (int i = 0; i < 100000; i++)
{
   map.put(new Integer(i),
      Integer.toString(i));
}

// TEST: Erzeugen der Iteratoren
for (int i = 0; i < maps.length; i++)
{
   maps[i] = (HashMap) map.clone();
   iterators[i] = maps[i].values().iterator();
}

// AWT-Fenster erstellen...
```

Abbildung A.2.: Einmaliges bzw. 10-maliges Erzeugen eines Iterators nach dem Klonen der HashMap mit 100.000 Einträgen

Tabelle A.1.: Speicherverbrauch vor dem Klonen (10 Iteratoren, 100.000 Einträge)

Objekt-Typ	**Instanzen**	**Byte**
HashMap$Entry	101.170	2.428.080
HashMap$ValueIterator	0	0
HashMap$Values	0	0
HashMap$EntryIterator	0	0
HashMap	60	2.400
String	102.356	2.456.544
Integer	100.316	1.605.056
`<class>[ ]`	1.615	1.703.080
`char[ ]`	102.720	2.559.880
Summe		**10,25680542 MB**

Tabelle A.2.: Speicherverbrauch nach dem Klonen und Erzeugen der Iteratoren (10 Iteratoren, 100.000 Einträge)

Objekt-Typ	**Instanzen**	**Byte**
HashMap$Entry	1.101.170	26.428.080
HashMap$ValueIterator	10	320
HashMap$Values	10	160
HashMap$EntryIterator	2	64
HashMap	70	2.800
String	102.356	2.456.544
Integer	100.316	1.605.056
`<class>[ ]`	1628	11.664.832
`char[ ]`	102.720	2.559.880
Summe		**42,64615631 MB**

Hilfsobjekte (zur Verwaltung der enthaltenen Objekte) geklont bzw. kopiert. Die verwalteten Objekte bleiben unverändert und werden nicht mitgeklont. [3] Dadurch beschränkt sich der erhöhte Speicherbedarf auf den von den geklonten Containern bzw. erstellten Iteratoren benötigten Speicher, der aber von der Anzahl der verwalteten Objekte abhängt (jede Referenz auf ein verwaltetes Objekt benötigt eine bestimmte Menge Speicher).

[3] Es gibt auch Klonoperationen, die eine so genannte *tiefe Kopie* eines Objekts erstellen. In diesem Fall werden alle referenzierten Objekte ebenfalls geklont, wodurch der Speicherbedarf erhöht wird.

Klonen von Containern

Die Tests für das Klonen eines Containers und das anschließende Erstellen des Iterators enhalten den Code aus Abb. A.2 in der `main`-Methode des Testprogramms. In der Abbildung ist der Code für die Tests mit 100.000 Einträgen und einem bzw. 10 Iteratoren. Der Quellcode für den Test mit 10 Einträgen und 100.000 Iteratoren ist analog und unterscheidet sich nur durch die Array-Indizes.

Einige beispielhafte Ergebnisse zu einem der drei durchgeführten Tests sind in den Tabellen A.1 und A.2 zu sehen. Hier wird gut deutlich, dass beim Klonen sehr viele HashMap$Entry-Objekte erstellt werden, um die 100.000 Einträge in jedem neuen Container-Klon zu verwalten. Außerdem steigt der Speicherverbrauch für Arrays von Objekten (`<class>[ ]`). Der Speicherbedarf dieses Ansatzes wird ab Seite 61 der Verwendung des CollectionIterators gegenübergestellt.

In allen drei Tests wächst der Speicherverbrauch nach dem Klonen und Erstellen der Iteratoren hauptsächlich durch die Hilfsobjekte der HashMap. Besonders deutlich wird das bei den HashMap$Entry-Objekten. Bei besonders häufigem Klonen – z.B. 100.000 Mal, wie bei dem dritten Test – wächst der Speicherverbrauch auch durch die HashMap$Values-, HashMap$ValueIterator-, HashMap$EntryIterator-, HashMap- und `<class>[ ]`-Objekte, da für jeden Klon je ein solches Objekt erstellt wird.

Tabelle A.3.: Laufzeit für das Klonen und Erzeugen der Iteratoren (in Sekunden)

Methode/Test	**1I/100.000E**	**10I/100.000E**	**100.000I/10E**
HashMap.clone	0,507	7,537	7,747
HashMap.values	nm	nm	0,42
Collection.iterator	nm	nm	0,835
Summe	0,507	7,537	9,002

Die mit *nm* gekennzeichneten Werte sind kleiner als 1 Millesekunde und damit nicht mehr messbar. *I* steht für „Iteratoren“ und *E* für „Einträge“.

Die Laufzeitmessungen ergaben, dass die `clone`-Methode mit Abstand die meiste Rechenzeit benötigt (siehe Tabelle A.3). Die beiden anderen Methoden verbrauchen weniger als 1 Millesekunde.

Insgesamt erweist sich das Klonen von Containern als nicht effizient, vor Allem durch den hohen Speicherverbrauch. Außerdem ist bei der `clone`-Operation keine Typsicherheit gegeben.

CollectionIterator-Ansatz

Bei diesem Ansatz wird versucht, Speicher für Hilfsobjekte zur Verwaltung von Container-Inhalten einzusparen. Anstatt den gesamten Container zu klonen wird hier nur sein Inhalt in einen anderen Container kopiert, nämlich in eine ArrayList

```
// Initialisieren der Variablen
Iterator<String> iter = null;

// Einfügen der Einträge in die HashMap
for (int i = 0; i < 100000; i++)
{
   map.put(new Integer(i),
      Integer.toString(i));
}

// TEST: Erzeugen der Iteratoren
iter = new CollectionIterator<String>(
   map.values());

// AWT-Fenster erstellen...
```

```
// Initialisieren der Variablen
HashMap<Integer, String>[] maps = new HashMap[10];
Iterator<String>[] iterators = new Iterator[10];

// Einfügen der Einträge in die HashMap
for (int i = 0; i < 100000; i++)
{
   map.put(new Integer(i),
      Integer.toString(i));
}

// TEST: Erzeugen der Iteratoren
for (int i = 0; i < iterators.length; i++)
{
   iterators[i] = new CollectionIterator<String>(
      map.values());
}

// AWT-Fenster erstellen...
```

Abbildung A.3.: Einmaliges bzw. 10-maliges Erzeugen eines CollectionIterators mit dem Inhalt der HashMap mit 100.000 Einträgen

(Paket `java.util`). Da der ArrayList-Container auf einem Array basiert, werden hier nur sehr wenige Hilfsobjekte zur Verwaltung des Container-Inhalts benötigt. Der Konstruktor des CollectionIterators bekommt ein Collection-Objekt, dessen Inhalt kopiert wird. Anschließend kann der CollectionIterator über die kopierten Objektreferenzen iterieren.

Wie im vorhergehenden Abschnitt wird hier der Quellcode für die drei durchgeführten Tests in Abb. A.3 dargestellt. Wieder ist der Quellcode für die letzten beiden Tests analog (Unterschied nur in den Indizes).

Dadurch, dass nur der Container-Inhalt kopiert wird, benötigt dieser Ansatz wesentlich weniger Speicher. Durch die Verwendung der ArrayList werden aber zusätzliche Objekte erzeugt, z.B. AbstractList$Itr, die ebenfalls Speicher verbrauchen. Trotzdem werden z.B. bei Test zwei (10 Iteratoren, 100.000 Einträge) anstatt der ca. 32 MB beim Klonen-Verfahren ($42,6$ MB $- 10,3$ MB $= 32,3$ MB) hier nur ca. 4,2 MB benötigt. Um nicht weiter ins Detail zu gehen, wird der Gesamtspeicherbedarf dieses Ansatzes ab Seite 61 dem Klonen von Containern gegenübergestellt.

Tabelle A.4.: Laufzeit für das Erzeugen von CollectionIteratoren (in Sekunden)

Methode/Test	**1I/100.000E**	**10I/100.000E**	**100.000I/10E**
CollectionIterator.<init>	0,064	0,71	3,776
HashMap.values	nm	nm	0,177
Summe	0,064	0,71	3,953

Die mit *nm* gekennzeichneten Werte sind kleiner als 1 Millesekunde und damit nicht mehr messbar. *I* steht für „Iteratoren“ und *E* für „Einträge“.

Auch bei der Laufzeit zeigen sich enorme Vorteile gegenüber dem Klonen von Containern wie die Tabelle A.4 im Vergleich zur Tabelle A.3 auf Seite 59 zeigt.

Insgesamt betrachtet ist dieser Ansatz wesentlich effizienter als das Klonen von Containern.

Vergleich

Alle gesammelten Werte bzgl. Speicherverbrauch aufzuzeigen, würde diese Arbeit mit vielen Tabellen und überflüssigen Informationen füllen. Um die beiden Ansätze dennoch vergleichen zu können, wird in diesem Abschnitt der Speicherverbrauch der durchgeführten Tests in Diagrammen illustriert. Auch die Laufzeiten werden gegenübergestellt.

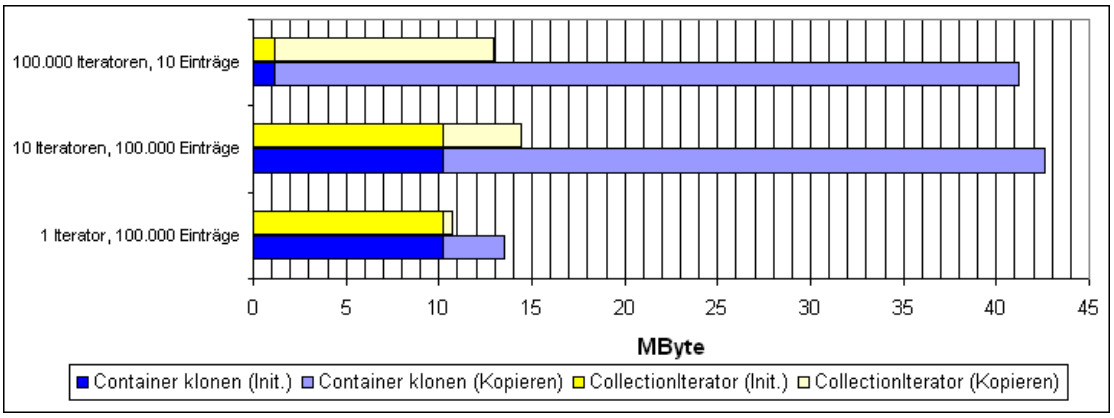

Abbildung A.4.: Gesamtspeicherverbrauch

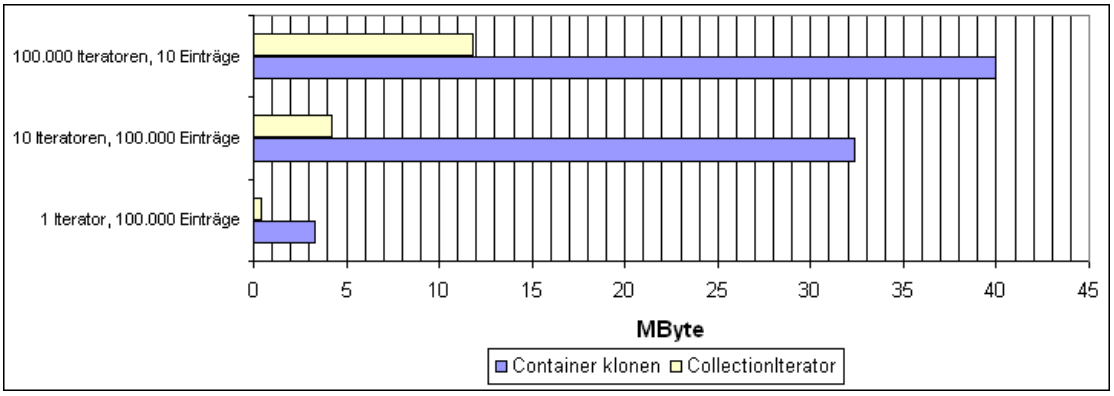

Abbildung A.5.: Speicherverbrauch für erzeugte Objekte

Die Abbildung A.4 zeigt für die beiden Ansätze und die drei Tests pro Ansatz den insgesamt verbrauchten Speicher. Da die Initialisierung bei allen Ansätzen gleich ist, ergibt sich dafür auch der gleiche Speicherbedarf. In der Abbildung A.5

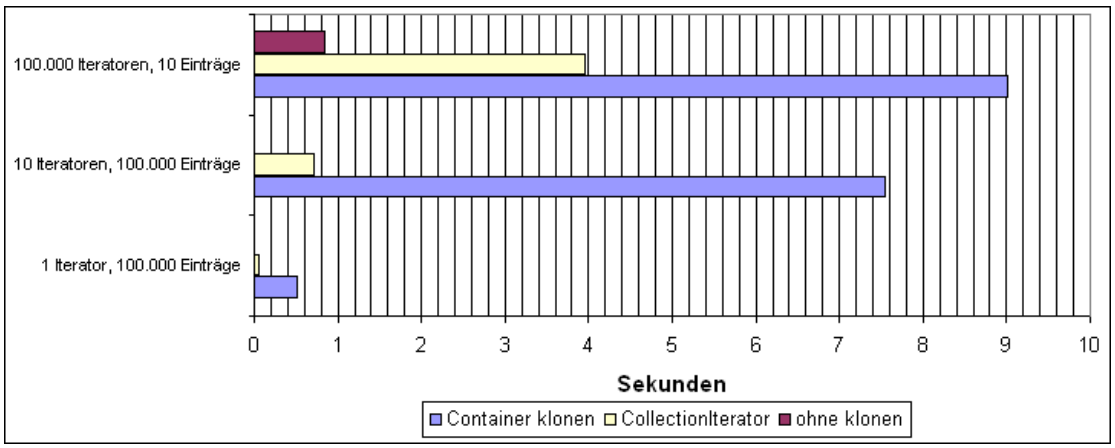

Abbildung A.6.: Laufzeit der beiden Ansätze im Vergleich zum Erstellen eines Iterators ohne Klonen

wird nur der Speicherbedarf für die beim Klonen bzw. Iterator-Erzeugen instanziierten Objekte dargestellt. Das verdeutlicht den unterschiedlichen Speicherbedarf.

Die Laufzeit der beiden Ansätze wird in der Abb. A.6 gegenübergestellt. Zusätzlich wird die Laufzeit abgebildet, die für die Rückgabe eines Iterators ohne Kopieren des HashMap-Inhaltes benötigt wird. Die Benutzung eines CollectionIterators ist eindeutig schneller als das Klonen von Containern. Natürlich benötigt das aber eine höhere Laufzeit, als einen Iterator von einem Container zurückgeben zu lassen, ohne den Inhalt vorher zu kopieren.

Bei allen durchgeführten Tests wird deutlich, dass sich das Klonen der Container nicht lohnt. Stattdessen sollte nur ihr Inhalt kopiert werden.

A.2. Aufwand für Methodenaufrufe mit Hilfe von Reflection

Um den Aufwand für die Verwendung von Reflection bei der Implementierung von Assoziationen zu messen, wurde ein kleines Testprogramm geschrieben, das 100.000 Mal eine parameterlose Methode per Reflection aufruft. Zum Vergleich wurde dieser Test auch mit einem 100.000-fachen direkten Aufruf der gleichen Methode durchgeführt und mit einem 100.000-fachen Aufruf der Methode per Reflection, wobei aber die Methode gecached wurde.

In Abbildung A.7 ist der Code für den ersten Test zu sehen. Hier wird 100.000 Mal per Reflection die Methode trimToSize eines ArrayList-Objekts geholt und per invoke-Anweisung aufgerufen.

Besonders lange braucht der Aufruf der Methode `java.lang.Class.getMethod`, nämlich 1.702 Millesekunden (siehe Abb. A.8, S. 63). Um diese Laufzeit bei wiederholten Aufrufen zu verkürzen, wurde bei dem nächsten Test (Abb. A.9) die zurückgegebene Methode gecached, um später nur noch aufgerufen zu werden.

```
public class TestPerformanceReflection
{
   static List list = new ArrayList();

   public static Method getMethod()
   {
      Method result = null;
      try
      {
         result = list.getClass().getMethod("trimToSize", null);
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
      return result;
   }

   public static void main(String[] args)
   {
      for (int i=0; i<100000; i++)
      {
         try
         {
            getMethod().invoke(list,null);
         }
         catch (Exception e)
         {
            e.printStackTrace();
         }
      }
   }
}
```

Abbildung A.7.: 100.000-facher Aufruf einer Methode per Reflection

Durch diese kleine Anpassung spart man einen großen Teil der Laufzeit, da die Methode nur noch einmal rausgesucht werden muss, sodass sich die Nachteile von Reflection bei mehrfachem Aufruf der gleichen Methode minimieren (siehe Abb. A.8, S. 63). Zum Vergleich wurde das gleiche Testprogramm mit direktem Methodenaufruf ausgeführt (Abb. A.10).

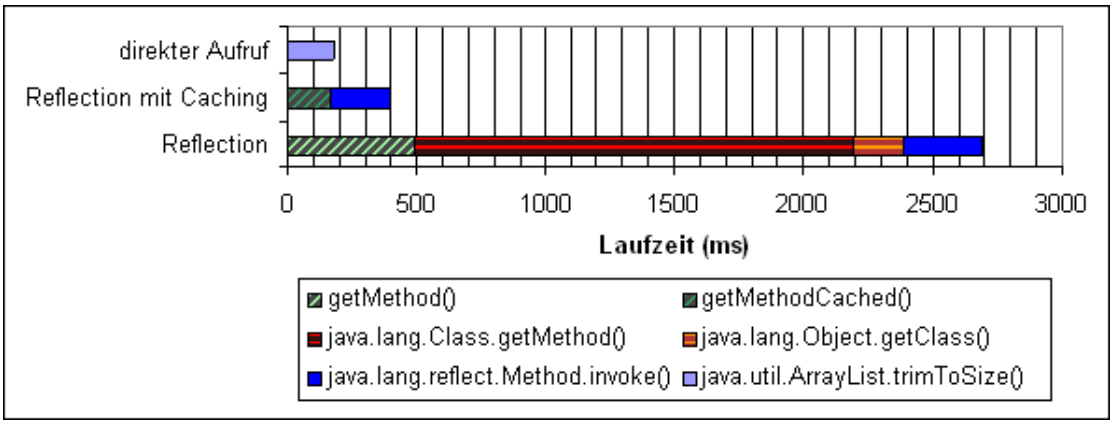

Abbildung A.8.: Laufzeitaufwand für Reflection

```
public class TestPerformanceReflection
{
   static List list = new ArrayList();
   static Method method = null;

   public static Method getMethodCached()
   {
      if (method != null)
         return method;
      try
      {
         method = list.getClass().getMethod("trimToSize", null);
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
      return method;
   }

   public static void main(String[] args)
   {
      for (int i=0; i<100000; i++)
      {
         try
         {
            getMethodCached().invoke(list,null);
         }
         catch (Exception e)
         {
            e.printStackTrace();
         }
      }
   }
}
```

Abbildung A.9.: 100.000-facher Aufruf einer Methode per Reflection mit Caching

Die Ergebnisse der drei Tests im Vergleich stellt die Abbildung A.8 dar (S. 63). Hier werden die Laufzeiten der einzelnen Methoden aufgeführt. Sie werden so dargestellt, dass auch die Gesamtlaufzeiten der drei Ansätze verglichen werden können. Die dargestellten Zeiten sind die Laufzeiten für die 100.000 Aufrufe der Methoden (bei Reflection mit Caching werden die Methoden `java.lang.Class.getMethod` und `java.lang.Object.getClass` nur einmal aufgerufen). Ein einzelner Aufruf der Methoden `java.lang.Class.getMethod` und `java.lang.Object.`

```
public class TestPerformanceReflection
{
   static ArrayList list = new ArrayList();

   public static void main(String[] args)
   {
      for (int i=0; i<100000; i++)
      {
         list.trimToSize();
      }
   }
}
```

Abbildung A.10.: 100.000-facher direkter Aufruf einer Methode

`getClass` innerhalb der Methode getMethodCached benötigte weniger als 1 ms. In dem Test von Reflection ohne Caching benötigt der Aufruf der Methode getMethod insgesamt 2.389 ms, wovon 1.702 ms für den enthaltenen Aufruf von `java.lang.Class.getMethod` und 197 ms für den Aufruf von `java.lang.Object.getClass` benötigt wird. Für die Methode getMethod ist in dem Diagramm nur die Laufzeit dargestellt, die nicht von den darin enthaltenen Methodenaufrufen benötigt wird, nämlich $2.389 - 1.702 - 197 = 490$ ms.

Insgesamt zeigt sich deutlich, dass bei der Verwendung von Reflection auch Caching genutzt werden sollte, um die Laufzeit zu minimieren, wenn eine Methode mehrfach verwendet wird. Am wenigsten Laufzeit (und Code) benötigt aber der direkte Aufruf einer Methode.

Literaturverzeichnis

[BCK+01] Bracha, Gilad, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup und Philip Wadler: *Adding Generics to the Java Programming Language: Participant Draft Specification*, April 2001.

[Bra04] Bracha, Gilad: *Generics in the Java Programming Language*, Juli 2004. Tutorial.

[ej-04] ej-Technologies GmbH: *JProfiler*. Online unter: http://www.ej-technologies.com/products/jprofiler/overview.html, 2004. Version 3.1.2.

[FNT98] Fischer, Thorsten, Jörg Niere und Lars Torunski: *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*. Diplomarbeit, Universität Paderborn, Juli 1998.

[FNTZ98] Fischer, Thorsten, Jörg Niere, Lars Torunski und Albert Zündorf: *Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language and Java*. Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), 1998.

[Fuj04] Fujaba Development Group: *Fujaba ToolSuite*. Online unter: http://www.fujaba.de, 2004.

[GHJV95] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[HBR00] Harrison, William, Charles Barton und Mukund Raghavachari: *Mapping UML Designs to Java*. Proceedings of the 15th Annual ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'2000), Volume 35: Seiten 178–188, Oktober 2000. Minneapolis, Minnesota, USA.

[Lea05] Lea, Doug: *Java Specification Request (JSR) 166: Concurrency Utilities*. Online unter: http://gee.cs.oswego.edu/dl/jsr166/dist/docs/, 2005. Schnittstellendefinition (APIs).

[Mai04] Maier, Thomas: *Associations*. Online unter: http://sourceforge.net/projects/associations/, November 2004. Version 0.4.

[Moa02] Moat, Susannah: *Fujaba Code Generation for static UML Models*. Universität Paderborn, Dezember 2002.

[MZ04] Maier, Thomas und Albert Zündorf: *Yet Another Association Implementation*. Fujaba Days 2004, 2004.

[Obj03] Object Management Group, Inc.: *OMG Unified Modeling Language Specification*, März 2003. Version 1.5.

[Wat96] Watt, David A.: *Programmiersprachen – Konzepte und Paradigmen*. Hanser, 1996. ISBN 3-446-17992-5.