# Echtzeit Synchronisierung von UML-Modellen unterschiedlicher technischer Basis am Beispiel von UML Lab und Fujaba

## Master-Arbeit

Fachgebiet Software Engineering

Prof. Dr. Albert Zündorf

Universität Kassel

28.09.2010

von

Andreas Koch

Betreuer: Dipl. Inf. Ruben Jubeh
Manuel Bork, M.Sc.
Prof. Dr. Albert Zündorf

## Zusammenfassung

CASE-Tools haben sich zur Unterstützung der Softwareentwicklung etabliert. Sie werden in verschiedenen Phasen des Entwicklungsprozesses eingesetzt und haben sich in ihrem Funktionsumfang ausdifferenziert. Durch die Kombination verschiedener spezialisierter Tools kann der Anwender weiteren Nutzen ziehen. Somit entsteht das Problem ein Modell zur jeweiligen Bearbeitung untereinander austauschen zu müssen. Sollen die Funktionalitäten zudem parallel eingesetzt werden, müssen die Modelle zur Laufzeit zueinander konsistent gehalten werden. Für diese Tool-Integration ist der Weg über die Modellsynchronisierung naheliegend.

Im Rahmen dieser Arbeit wird die Synchronisation von Modellen basierend auf dem Fujaba-Metamodell (basierend auf dem UML-Metamodell 1.4) und dem UML-Metamodell 2.2 vorgestellt. Auch wenn der Ansatz auf Erweiterbarkeit ausgelegt ist, beschränkt sich die Umsetzung auf alle Elemente, die Klassendiagramme betreffen. Darauf aufbauend ist ein Eclipse-Plugin entstanden, welches sich in Fujaba4Eclipse und UML Lab einbettet, um zusätzlich zur Modellsynchronisierung eine enge Verknüpfung der Funktionalitäten dieser zwei CASE-Tools zu ermöglichen.

# Inhaltsverzeichnis

# 1 Motivation

CASE-Tools (Computer-aided Software Engineering) werden zur Unterstützung in der Softwareentwicklung eingesetzt. Beispielsweise können sie während der Planungs- bzw. der Entwurfsphase oder bei der Erstellung einer verständlichen Dokumentation helfen. Als Standard für diese Tools hat sich die UML[1]-Spezifikation durchgesetzt. Auf Grund der zahlreichen spezifizierten Diagrammarten in UML, kann der Entwickler in verschiedenen Bereichen unterstützt werden. Dies betrifft sowohl die Modellierung von Struktur als auch die des Verhaltens von Programmen. Dementsprechend reicht die Spanne in CASE-Tools von der Einbindung weniger Diagrammarten für ausgewählte Teilschritte der Entwicklung bis hin zur Umsetzung der gesamten Spezifikation für ein möglichst breites Funktionsspektrum.

Bei sehr jungen oder besonders spezialisierten CASE-Tools tritt häufig die Problematik auf, dass nicht alle Diagramme unterstützt werden, die der Anwender benötigt. In diesem Fall möchte er die Funktionalitäten aus verschiedenen Tools miteinander kombinieren. Nutzen diese ein gemeinsames Metamodell (beispielsweise das UML-Metamodell) unterstützt dies den Ansatz, ein selbst erstelltes Projekt oder zumindest Teile davon in unterschiedlichen Tools zu bearbeiten. Hierzu kommt häufig eine Import/Export-Funktion zum Einsatz, die die Umwandlung der Projektdaten in ein allgemeines Format ermöglicht. Auf diesem Weg wird der Austausch zwischen den Tools ermöglicht.

Durch den Import/Export eines Projektes können aber nicht alle Anwendungsszenarien abgedeckt werden. Soll ein Modell zur Laufzeit parallel in verschiedenen Tools genutzt werden, müssen die Änderungen unmittelbar von einem in das andere Tool übertragen werden können. Der wiederholte Import/Export stellt einen Umweg dar. Werden Funktionalitäten aus einem CASE-Tool genutzt, deren resultierende Daten nicht durch das andere Metamodell abgebildet werden können, ist zudem der standardisierter Import/Export von diesen Daten nicht möglich. Um dem entgegen zu wirken, wird eine direkte Kommunikation zwischen den Tools aufgebaut, d.h. ein direkter Abgleich, eine Synchronisation, der Metamodelle und

[1] Unified Modeling Language

entsprechend auch der Modelle. Durch diese Verbindung ist es möglich die Stärken beider Tools miteinander zu kombinieren.

Dieser Arbeit beschreibt die Synchronisation des Fujaba-Metamodells (UML1), eingesetzt in Fujaba, und des UML2-Metamodells am Beispiel von UML Lab. Die Motivation für die Synchronisierung von Fujaba- auf UML2-Metamodell liegt vor allem in der Möglichkeit auf diesem Weg einzelne Bereiche aus Fujaba Schritt für Schritt auf das UML2-Metamodell umzustellen ohne die Funktionsweise der übrigen Bereiche während dieser Zeit einzuschränken. Somit können ältere, mit Fujaba erstellte Projekte auch in dem neu entwickelten UML Lab weiterverwendet werden. Die Motivation für die Gegenrichtung zeigt sich in der Nutzung von Funktionalitäten aus Fujaba in UML Lab, die dort (noch) nicht umgesetzt sind. Dazu gehört besonders das *Story Driven Modeling* (SDM).

# 2 Grundlagen

## 2.1 Modelle

Der Begriff des Modells findet seine Verwendung in vielen Anwendungsgebieten. Neben der Softwaretechnik wurde es vorher bereits in vielen anderen Bereichen genutzt und geprägt. So bezeichnet im Maschinenbau ein Modell ein maßstabsgerechtes, aber minimalisiertes Abbild einer geplanten Konstruktion, um beispielsweise verschiedene Verhaltensweisen unter realen Bedingungen zu testen. In der Physik wird mit Hilfe eines Modells hingegen versucht ein reales physikalisches Phänomen zu erklären bzw. mathematisch abzubilden und berechenbar zu machen. Trotz der unterschiedlichen Anwendungen, haben Modelle stets eins gemeinsam: Sie dienen dazu komplexe Probleme derart abzubilden oder nachzuahmen, dass sie in der gewünschten Anwendungsdomäne lösbar sind. Zu diesem Zweck haben sie auch ihren Weg in die Softwaretechnik gefunden.

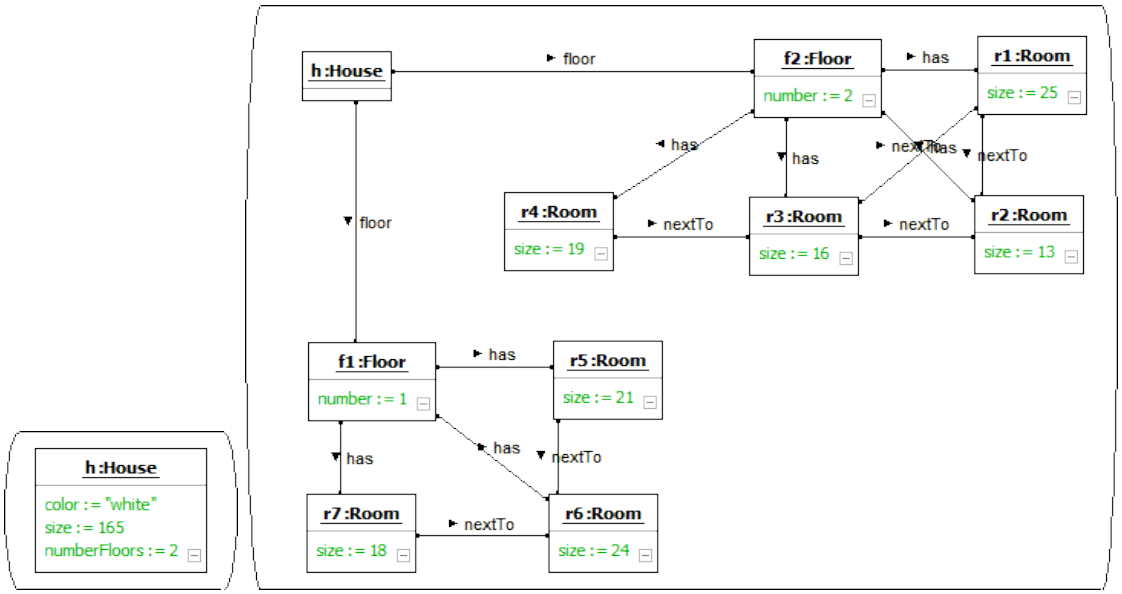
Für eine Anwendung ist es häufig nötig einen Teil der realen Welt nutzbar zu machen,

**Abbildung 2.1:** Bild eines zweistöckigen Einfamilienhauses

d.h. ihn in einer für den Computer verständlichen Form zu repräsentieren. Hierfür werden alle benötigten Elemente modelliert, d.h. man abstrahiert so weit wie möglich und so viel wie nötig von den realen Gegebenheiten. Betrachtet man beispielsweise ein einfaches Einfamilienhaus wie in Abbildung 2.1. Um hierfür ein Modell zu erstellen, müssen zuerst die

Rahmenbedingungen festgelegt werden. Es ist also nicht praktikabel das Haus auf atomarer Ebene, also möglichst genau, zu modellieren, sondern, wie oben erläutert, soll es den Anforderungen entsprechend explizit für die Anwendungsdomäne nachgebildet werden.

Wie man das Haus am besten umsetzt, hängt von den Anforderungen an die Genauigkeiten




(a) Einfaches Beispieldiagramm

(b) Komplexes Beispieldiagramm

**Abbildung 2.2:** Objektdiagramme für ein Einfamilienhaus

der benötigten Daten ab. Sind diese sehr gering und auf einige wenige, äußerliche Merkmale beschränkt, so zeigt Abbildung 2.2(a) ein mögliches Objektdiagramm. Diese Informationen können für eine Anwendung oder ein Spiel, das sich mit Stadtplanung beschäftigt, bereits ausreichend sein. Neben einem Objekt für das Haus selbst, besitzt es nur wenige primitive Attribute wie Größe und Außenfarbe. Ist mehr Detailwissen über die Innenräume nötig, wie beispielsweise in einer Architektursoftware, so zeigt Abbildung 2.2(b) ein einfaches Objektdiagramm. Hier gibt es neben einem Objekt der Klasse *House*, mehrere Instanzen der Klassen *Floor* bzw. *Room*, welche zusätzlich miteinander verknüpft sind. Somit können genauere Aussagen über den Aufbau des Hauses von außen (Anzahl der Etagen) und von Innen (Anzahl und Größe der Räume) getroffen werden. Hierzu ist es nötig Instanzen verschiedener Klassen mit unterschiedlichen Assoziation zu erzeugen. Damit ein solches Objektdiagramm möglich ist, muss in einem zugehörigen Modell festgelegt werden, welche Klassen man überhaupt nutzen kann und welche Beziehungen diese zueinander haben dürfen. In Abbildung 2.3 ist das zugehörige Modell als Klassendiagramm zu sehen. Über

die drei Klassen, ihre Attribute und Assoziationen ist die Struktur aus Abbildung 2.2(b) wiederzuerkennen.

Ebenso wie man aus Abbildung 2.1 das Objektdiagramm in Abbildung 2.2(b) erstellen

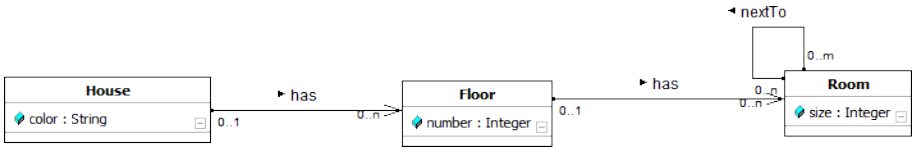



**Abbildung 2.3:** Klassendiagramm eines einfachen Einfamilienhauses

kann, ist es auch möglich ein Objektdiagramm aus dem Modell in Abbildung 2.3 zu erzeugen. In Abbildung 2.4 ist ein passendes Diagramm zu sehen. Zu Gunsten der Übersichtlichkeit ist es auf die wesentlichen Teile gekürzt. So fehlen beispielsweise Kardinalitäten bei den Assoziationen oder die Typen der primitiven Attribute. Weiterhin ist die Assoziation „nextTo“ aus Abbildung 2.3 nicht enthalten. In dem Restdiagramm werden im Programmieralltag bekannte Begriffe wie *Class* oder *Property* verwendet, welche die Verbindungen zwischen beiden Diagrammen verdeutlichen. So wird die Klasse *House* aus dem Modell im Objektdiagramm über das Objekt c1 der Klasse *Class* und zwei Objekte (p1 und p2) der Klasse *Property* beschrieben. Eine vergleichbare Struktur weisen die Klassen *Floor* mit den Objekten c2, p3 und p4 bzw. *Room* mit den Objekten c3 und p5. Durch den Wert des Attributs *name* kann die jeweilige Zuordnung zum Klassendiagramm erfolgen. Die in Ab-

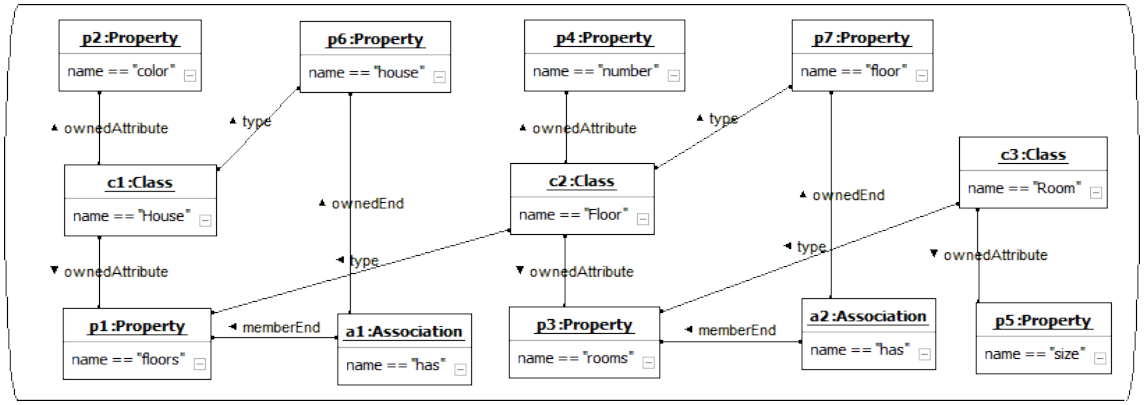



**Abbildung 2.4:** Objektdiagramm des Klassendiagrams aus Abbildung 2.3

bildung 2.3 durch Pfeile dargestellten Assoziationen bestehen im Objektdiagramm aus drei Objekten. Betrachtet man beispielsweise die Referenz *has* von *House* auf *Floor*, wird diese im Objektdiagramm über p1, p6 und a1 dargestellt. p1 ist das zu *House* zugehörige Attribut, über das die Referenzen auf *Floor*-Objekte verwaltet werden. Für die Gegenrichtung gibt es p6 und a1 als verbindendes Element.

Betrachtet man die bisherigen Beispiele wird deutlich, dass mit jedem Schritt die Abstraktion zunimmt. So existiert ebenso wie zu dem Objektdiagramm aus Abbildung 2.2(b) auch zu jenem aus Abbildung 2.4 ein Modell. Weil sich dieses eine Abstraktionsebene höher befindet als das vorherige Modell, bezeichnet man das Modell in Abbildung 2.5 als Metamodell von diesem. Auch dieses kann wieder selbst auf einem Metamodell basieren, was sich theoretisch beliebig weiter abstrahieren lässt. Wird für eine oder mehrere dieser Meta-Ebenen eine Spezifikation festgelegt (beispielsweise UML), können diese für Entwickler als gemeinsame Grundlage bei der Erstellung eigener Modelle dienen.

Unabhängig davon wie ein Modell aufgebaut ist, stellt sich die Frage, wie es dargestellt

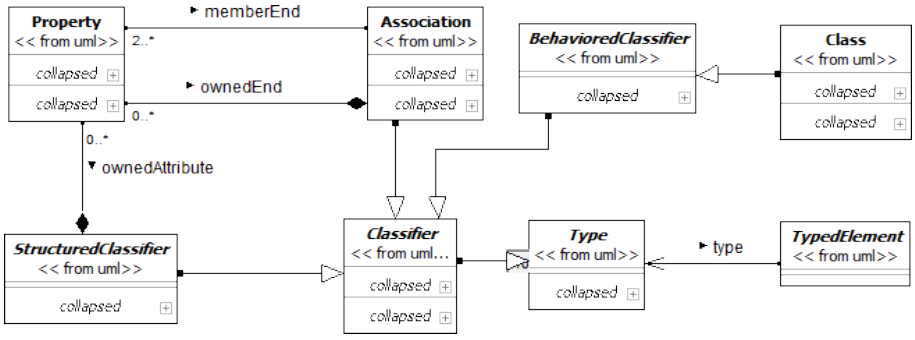



**Abbildung 2.5:** Klassendiagramm für das Objektdiagramm aus Abbildung 2.4

werden soll. Eine gute Visualisierung kann das Verständnis für ein Problem bei einem Entwickler deutlich verbessern. Betrachtet man zum Vergleich das Objektdiagramm in Abbildung 2.4 und das Klassendiagramm in Abbildung 2.3, wird deutlich, wie durch eine Visualisierung Semantik hinzugefügt werden kann. Klassen erkennt man durch ihre rechteckige Umrahmung. Attribute werden nicht eigenständig dargestellt, sondern innerhalb der Klassen untereinander positioniert. Auch Assoziationen setzen sich nicht aus mehreren separaten Konstrukten zusammen, sondern bestehen aus einer schlichten Linie zwischen zwei Klassen. Jeder der sich dieser Semantik bewusst ist, also sozusagen die Legende der Diagramme kennt, kann sie problemlos verstehen. Für Außenstehende ist es ohne Erklärung dagegen kaum möglich Sinn in diese Darstellung zu bringen.

Damit genau dieses Problem möglichst ausgeschlossen werden kann und Entwickler unbekannte Modelle verstehen können, ist es nötig, eine solche Semantik möglichst in Verbindung mit einheitlichen Metamodellen zu verwenden. Im Endeffekt durchgesetzt und heutzutage Standard unter den Modellierungssprachen ist UML (**U**nified **M**odelling **L**anguage). Die Idee mit UML eine visuelle Modellierungssprache einzuführen, hatten Grady Booch, Ivar Jacobson und James Rumbaugh während ihrer gemeinsamen Arbeit bei „Rational Software“. Weil sie bereits Erfahrung auf diesem Gebiet hatten, konnten sie ein überzeugendes

Grundkonzept aufstellen, welches im Laufe der Zeit von der Object Management Group gepflegt und in der Entwicklung vorangetrieben wurde. Während sich über verschiedene Zwischenversionen die aktuelle UML-Spezifikation zurzeit in Version 2.3 befindet, werden im Folgenden die UML 1.4- und 2.2-Spezifikationen, im Besonderen Klassendiagramme betreffende Abschnitte, genauer beleuchtet, weil diese die Grundlage für diese Arbeit bilden.

### 2.1.1 Fujaba-Metamodell (basierend auf UML 1.4)

Das Fujaba-Metamodell basiert auf dem UML-Metamodell, Version 1.4[11]. Es wird daher nicht die originale UML-Spezifikation, sondern die Umsetzung in Fujaba vorgestellt.

Im Folgenden werden die Grundsätze in drei Abschnitten näher beleuchtet. Der Übersichtlichkeit halber sind die primitiven Attribute und alle Methoden ausgeblendet, so dass die Konzentration auf den Beziehungen der Elemente untereinander liegt.

**Klassen:** Die Abbildung 2.6 zeigt Teile des Metamodells, Klassen bzw. das Äquivalent in

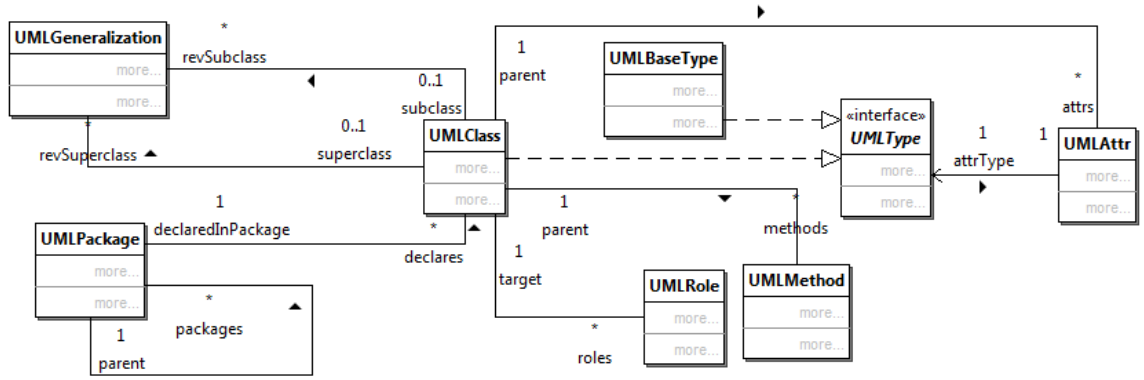



**Abbildung 2.6:** Ausschnitt aus dem Fujaba-Metamodells betreffend Klassen

Fujaba *UMLClass* betreffend. *UMLPackage* ermöglicht die Modellierung von Paketstrukturen und die Einordnung der Klassen in diese. Mit Hilfe von *UMLGeneralization* kann die Vererbung zwischen Klassen modelliert werden, wobei explizit zwischen Super- und Subklassen unterschieden wird. *UMLMethod* entspricht den Methoden einer Klasse und wird später genauer behandelt. Die Klassen *UMLAttr* und *UMLRole* stehen für die Attribute einer Klasse, wobei erstere die primitiven und letztere die komplexen Attribute, also die Beziehungen zwischen Klassen, repräsentieren. Damit ist *UMLRole* ein Teil von Assoziationen, welche ebenfalls im späteren Verlauf erläutert werden.

**Methoden:** Die Methoden, repräsentiert durch *UMLMethod*, bilden den Kern der Abbil-

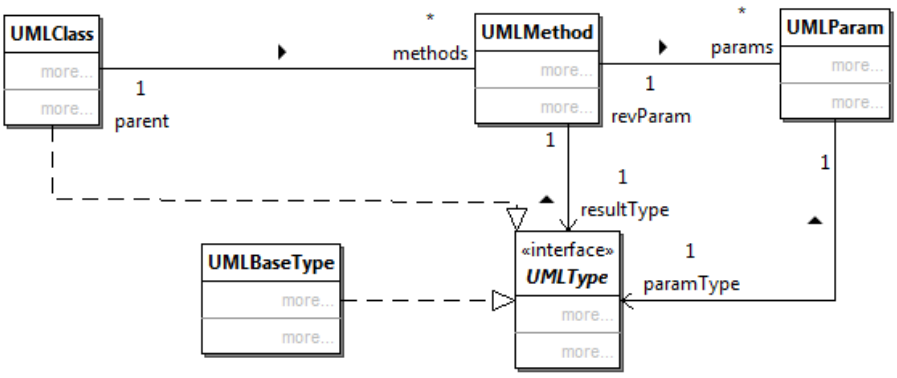



**Abbildung 2.7:** Ausschnitt aus dem Fujaba-Metamodells betreffend Methoden

dung 2.7. Dort ist neben ihrer Beziehung zu *UMLClass* die Klasse *UMLParam* zur Modellierung für die Parameter der Methoden zu erkennen. *UMLType* als Basisinterface für Datentypen erfüllt die Aufgaben als Rückgabewert für Methoden und Typ der Methodenparameter. Durch die Vererbungspfeile von *UMLClass* und *UMLBaseType* (Grundlage für primitive Datentypen und Datenarrays in Fujaba) werden die zwei Implementierungen des Interfaces definiert.

**Assoziationen:** Die Abbildung 2.8 zeigt die Struktur von Assoziationen in Fujaba. Neben der Kernklasse *UMLAssoc* dient *UMLRole* zur Modellierung der Klassenattribute an den jeweiligen Enden. *UMLCardinality* dient zur Festlegung der oberen und unteren Grenzen der Assoziationsenden, womit beispielsweise 1..1 oder 1..n-Assozationen definiert werden können. *UMLQualifier* ermöglicht die Qualifizierung der Assoziationsenden, d.h. sie können durch ein zusätzliches Attribut kategorisiert werden

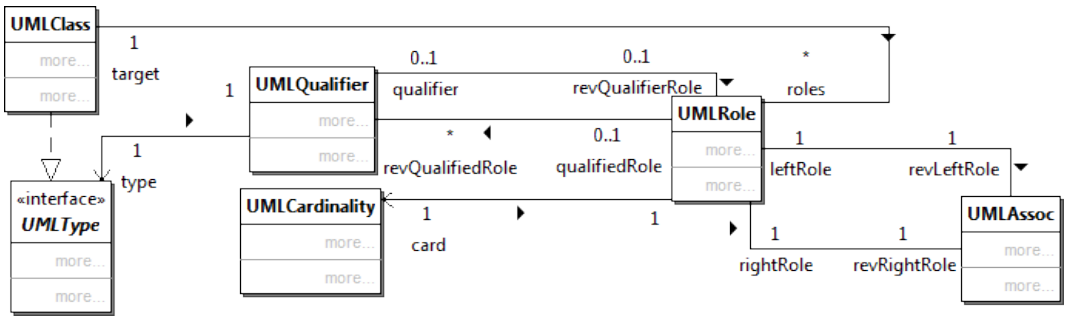



**Abbildung 2.8:** Ausschnitt aus dem Fujaba-Metamodells betreffend Assoziationen

**Implementationsdetails des Fujaba-Metamodells**

Neben der Struktur sind auch einige Bereiche in der Implementation des Metamodells für diese Arbeit relevant. Daher sind im Folgenden einige Anmerkungen zu diesem Thema aufgeführt.

Über die gesamte Implementation des Fujaba-Metmodells hinweg werden Änderungen an einem Attribut bzw. einem Objekt über Events an angemeldete Listener gesendet. Dabei handelt es sich um *PropertyChangeEvent*s bzw. Erweiterungen von diesen wie das *CollectionChangeEvent*. Diese enthalten, neben dem übergeordneten Objekt des Attributes, unter anderem einen Bezeichner, um das geänderte Attribut zu identifizieren. Zusätzlich wird der neue und, wenn vorhanden, alte Attributwert mitgeliefert. Diese Events werden nur bei einer wirklichen Änderung verschickt, d.h. sind der alte und der potentiell neue Attributwert identisch, wird weder der „neue“ Wert gesetzt, noch ein Event erzeugt.

Durch die Struktur des Metamodells sind viele der Beziehungen zwischen den Elementen bidirektional angelegt, d.h. betrifft eine Änderung eine solche Assoziation wird jeweils ein Event für jedes der beiden Assoziationsenden erzeugt.

### 2.1.2 UML 2.2-Metamodell

Das UML-Metamodell ist zu Version 2.2[13] in vielen Bereichen verändert und gegenüber den Vorgängerversionen erweitert worden. Auch in Bezug auf Klassendiagramme gibt es Anpassungen, die bei der Erläuterung des Metamodells erwähnt werden. Wie im vorherigen Abschnitt werden die drei Bereiche „Klassen“, „Methoden“ und „Assoziationen“ unterschieden. Zusätzlich werden Attribute und Methoden für eine bessere Übersichtlichkeit ausgeblendet.

**Klassen:** In Abbildung 2.9 ist ein Ausschnitt aus dem Metamodell von UML 2.2 zu sehen, der sich auf den Bereich der Klassen bezieht. Im Vergleich zu der vergleichbaren Abbildung im vorherigen Abschnitt sind einige Bereiche differenzierter ausgeprägt. So werden Klassen und Interfaces durch *Class* und *Interface* separat behandelt. Auf die Modellierung von Paketstrukturen hat dies keinen Einfluss, weil die Einsortierung in das richtige *Package* über die gemeinsame Oberklasse *PackageableElement* erfolgt. Attribute (*Property*) und Methoden (*Operation*) werden dagegen über unterschiedliche, aber gleichnamige Assoziationen ausgehend von *Interface* bzw. *Class* und die Superklasse *StrcuturedClassifier* verwaltet. Außerdem werden Attribute nichtmehr in primitiv und komplex unterteilt, sondern

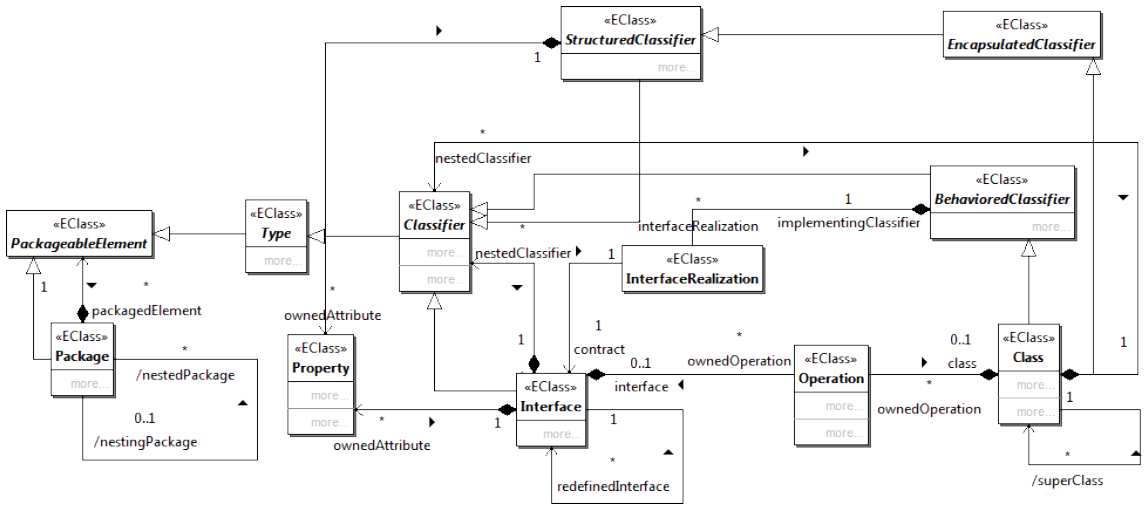



**Abbildung 2.9:** Ausschnitt aus dem UML 2.2-Metamodell betreffend Klassen

einheitlich über (*Property*) definiert. Merklich aufwendiger ist auch die Darstellung von Vererbung geworden. So gibt es drei verschiedene Varianten: zum Ersten von *Interface* auf *Interface* über die Assoziation „redefinedInterface“; zum Zweiten von *Class* auf *Class* über die Assoziation „superClass“; zum Dritten von *Interface* auf *Class* über die Hilfsklasse *InterfaceRealization*.

**Methoden:** Die Abbildung 2.10 zeigt einen Ausschnitt mit allen relevanten Elementen

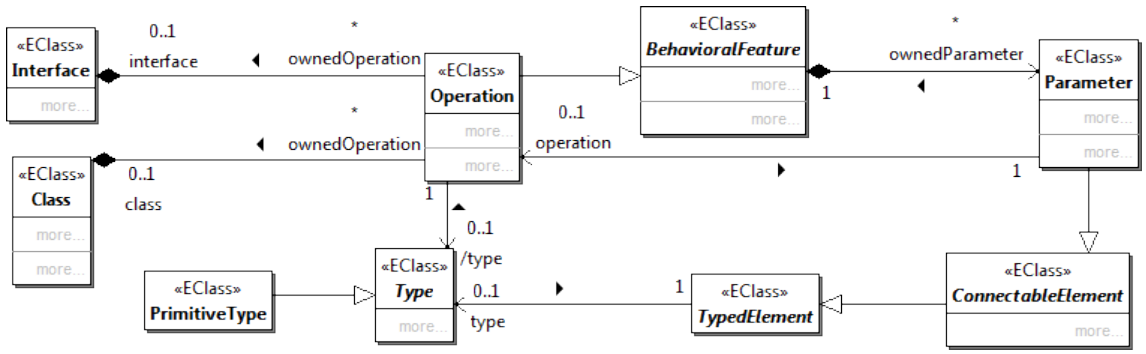



**Abbildung 2.10:** Ausschnitt aus dem UML 2.2-Metamodell betreffend Methoden

für *Operation*, welches zur Modellierung von Methoden in UML 2.2 verwendet wird. Zum einen wird die Klasse *Parameter* für die Methodenparameter inklusive des zugewiesenen Datentyps genutzt. Zum anderen ist die Assoziation „type“ zu erkennen, welche dem Rückgabetyp der Methode entspricht.

**Assoziationen:** Der Diagrammausschnitt betreffend Assoziationen in UML 2.2 ist in Abbildung 2.11 zu sehen. Hervorzuheben sind *Association* und *Property*, welche den Kern jeder

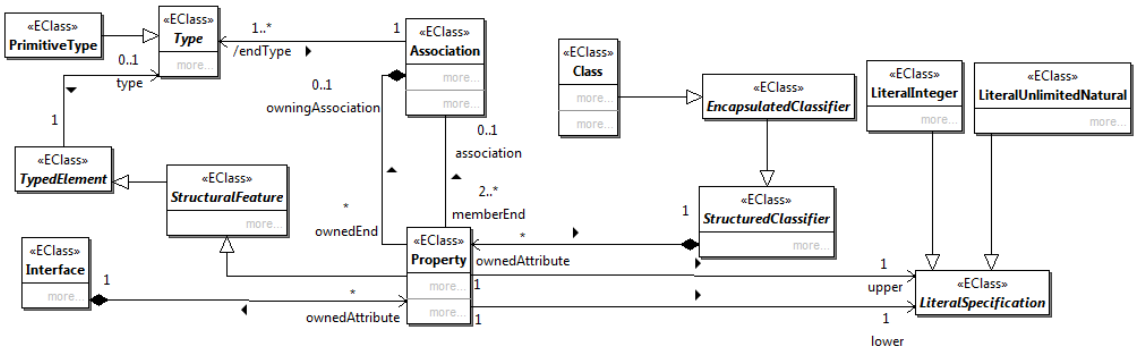



**Abbildung 2.11:** Ausschnitt aus dem UML 2.2-Metamodell betreffend Assoziationen

Assoziation bilden. Im Gegensatz zu dem ersten Metamodell gibt es keine Unterscheidung mehr zwischen den beiden Enden der Assoziation; beide werden über dieselbe Kante verwaltet („member end“). Die Kardinalitäten sind nun auf zwei Objekte aufgeteilt. Sowohl für die obere („upper“) als auch die untere („lower“) Grenze jedes Assoziationsendes gibt es ein eigenes Objekt.

### Implementationsdetails des UML2-Metamodells

Für diese Arbeit kommt die auf EMF[1] basierende Implementierung der UML2-Spezifikation für die Eclipse-Plattform zum Einsatz. Implementationsdetails, die für diese Arbeit relevant sind, werden im Folgenden genauer beleuchtet.

Um auf Änderungen an Objekten bzw. deren Attributen in einem Modell reagieren zu können, bietet die Implementation die Möglichkeit sich über Listener an verschiedenen Bereichen anzumelden und auf diesem Weg sogenannte Notifications zu erhalten. Diese enthalten alle notwendigen Informationen über die Änderungen. Dazu gehört beispielsweise ein Identifizierungswert für das angepasste Attribut oder das Elternobjekt des Attributes. Zusätzlich wird auch der neu zugewiesene Wert und, wenn ein vorheriger gesetzt war, dessen alter Wert mitgesendet.

Bedingt durch die UML2-Spezifikation sind viele Beziehungen zwischen den Metamodellelementen sehr genau und differenziert dargestellt. Das resultiert in einer großen Anzahl an unidirektionalen Assoziationen zwischen diesen. Somit werden Änderungen häufig ausschließlich vom Ursprungsobjekt aus verbreitet.

[1] Das Eclipse Modeling Framework ermöglicht die Generierung von Quellcode aus strukturierten Daten, siehe http://www.eclipse.org/modeling/emf/

## 2.2 Import/Export

Für die Bereitstellung einer Im-/Export Funktion in einer Applikation gibt es verschiedene Beweggründe; sei es um Daten in ein übersichtliches (und für andere lesbares) Format zu bringen, beispielsweise der PDF-Export in Textverarbeitungsprogrammen; sei es um Einstellungen für eine spätere Wiederherstellung zu sichern, beispielsweise der Lesezeichenexport in einem Browser. Ein häufiger Grund liegt darin, die Möglichkeit zu haben in einer Applikation ein unter Umständen proprietäres, auf die eigenen Ansprüche abgestimmtes Dateiformat einzusetzen, um gleichzeitig die verwendeten Daten auch anderen Programmen über verständliche Formate zur Bearbeitung zur Verfügung stellen zu können. So verwendet man den Export in das CSV-Format, um Daten strukturiert in einer einfachen Textdatei abzulegen.

Auf diesem Prinzip basierend können Daten über unterschiedliche Applikationen hinweg bearbeitet werden. Hierbei erfolgt der Export im Allgemeinen durch vollständige Übertragung der Daten in eine Datei und der Import durch Einlesen derselben Datei. Grundsätzlich kann man dies in zwei Szenarien unterteilen. Im ersten Szenario basieren beide Anwendungen auf dem gleichen Datenmodell, d.h. es erfolgt beim Ex-/Import keine Umwandlung in bzw. von einer Struktur, welche nicht dem internen Modell gleicht. Der Vorteil liegt darin, dass auf keiner Seite zusätzliche Informationen durch das Modell abgebildet werden und die Daten somit verlustfrei übertragen werden können. Ein vollständiger Export bedingt demnach einen ebenso vollständigen Import des Datensatzes. Im zweiten Szenario basieren beide Anwendungen auf einem unterschiedlichen Modell und der Ex-/Import erfolgt entweder über eines der beiden existierenden Formate oder über ein zusätzliches Format, welches beide Anwendungen verarbeiten können. Je größer hierbei die Unterschiede zwischen den Modellen sind, umso größer werden die Verluste, weil davon ausgegangen werden muss, dass vieles aus dem Ursprungsmodell in das Zielmodell (und umgekehrt) nicht integriert werden kann. Gehen die Verluste über informelle Daten (beispielsweise die exportierte Dokumentation) hinaus, wird ein effektives Arbeiten erschwert, weil bei jedem Ex-/Import zusätzliche Wiederherstellungsarbeit zu leisten ist.

### 2.2.1 XMI

XMI (**X**ML **M**etadata **I**nterchange) ist ein offenes von der Object Management Group (OMG) gepflegtes und auf dem XML Standard aufbauendes Format zum Austausch von

Daten. Es gründet sich in den Bemühungen der OMG ein Standardaustauschformat für Modelle zu etablieren, die auf der MOF[2]-Spezifikation basieren. Neben der allgemeinen Struktur einer XMI-Datei bietet die Spezifikation selbst kein weiteres Vokabular an, sondern ist als Anleitung zu sehen, wie ein XMI-konformes Vokabular erstellt werden soll. Ein solches wird von der OMG beispielsweise für UML bereits zur Verfügung gestellt. Bis Version 1.x konnte ein Vokabular nur über DTDs (**D**ocument **T**ype **D**efinitions) definiert werden, seit Version 2.x sind auch die flexibleren XML Schemata nutzbar. Durch die enge Verknüpfung mit dem MOF-Standard, bedingt dessen stetige Weiterentwickelung eine gleichzeitige Anpassung der XMI-Spezifikation, so dass sich deren Versionsschritte beinahe parallel entwickeln.

Eine XMI-Datei besteht aus bis zu drei Bereichen: dem Inhalt, den Erweiterungen und der Dokumentation. Für andere Anwendung interessant ist vor allem der Inhalt, weil dort die exportierten Modelle serialisiert sind. In Listing 2.1 ist der Ausschnitt aus dem Inhaltsbereich einer XMI-Datei zu sehen. Dabei handelt es sich die Klasse *House* aus der Abbildung 2.3 in Abschnitt 2.1. Die exportierte Datei wurde mit Hilfe ArgoUML [3] erzeugt. Wegen dem großen Umfang des generierten Modells ist der Ausschnitt auf nur eine Klasse und der Übersichtlichkeit halber manuell verkürzte IDs reduziert. Alle benutzten Tags in diesem Listing liegen in dem Namespace uml, welcher der von der OMG für den Export von UML-Modellen bereitgestellt wird.

```
1  <UML:Class xmi.id = 'class1'
2    name = 'House' visibility = 'public' isSpecification = 'false' isRoot
       = 'false'
3    isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
4    <UML:Classifier.feature>
5      <UML:Attribute xmi.id = 'attr1'
6        name = 'color' visibility = 'public' isSpecification = 'false'
           ownerScope = 'instance'
7        changeability = 'changeable' targetScope = 'instance'>
8        <UML:StructuralFeature.multiplicity>
9          <UML:Multiplicity xmi.id = 'mult1'>
10           <UML:Multiplicity.range>
11             <UML:MultiplicityRange xmi.id = 'multrange1'
12               lower = '1' upper = '1'/>
13           </UML:Multiplicity.range>
14         </UML:Multiplicity>
```

[2]Die MetaObject Facility-Spezifkation stellt grundlegende Strukturen zur Entwicklung miteinander kompatibler Modell zur Verfügung, siehe http://www.omg.org/mof/

[3]zu finden unter www.argouml.org

```
15        </UML:StructuralFeature.multiplicity>
16        <UML:StructuralFeature.type>
17          <UML:DataType href =
                'http://argouml.org/profiles/uml14/default-uml14.xmi#color'/>
18        </UML:StructuralFeature.type>
19      </UML:Attribute>
20    </UML:Classifier.feature>
21  </UML:Class>
```

**Listing 2.1:** Ausschnitt einer XMI-Datei mit verkürzten IDs

Die Beschreibung der Klasse erfolgt in den Zeilen 1 bis 3 mit Attributen wie Name oder Sichtbarkeit. Ihr Attribut *color* beginnt ab Zeile 5 und wird mit der Kardinalität (Zeile 8 bis 15) und seinem Typ (Zeile 16 bis 18) vervollständigt. Die Assoziation zu *Floor* ist an einer anderen Stelle der XMI-Datei aufgeführt und wird zur besseren Übersichtlichkeit ausgelassen.

Möchte man exportierten Dateien zusätzlich anwendungsspezifische Informationen anhängen, können in XMI die „Extensions“ genutzt werden. Damit hat man die Möglichkeit beliebige Metadaten hinzuzufügen, solange sie validem XML entsprechen. Beispielsweise nutzen dies manche Anwendungen um GUI-Koordinaten von exportierten Komponenten zu serialisieren. Auch wenn andere Programme versuchen können auch diese Daten auszuwerten, müssen gemäß Spezifikation keine Schemata oder Dokumentationen bezüglich der Bedeutung dieser bereitgestellt werden. Um die Interoperabilität zu gewährleisten sollten nicht verwertbare „Extensions“ importierter Modelle zumindest vollständig in eigene Exporte integriert werden. Das kann die Dateigröße merklich erhöhen, hat aber für den Benutzer den Vorteil, dass anwendungsspezifisches nicht durch das Benutzen verschiedener Programme verloren geht.

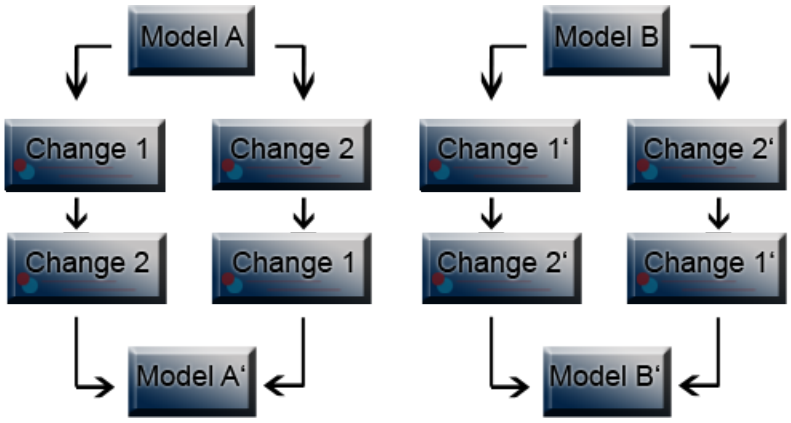
Im dritten Bereich, der Dokumentation, können erklärende Anmerkungen hinzugefügt werden. So können eine Bezeichnung und die Version der Anwendung, mit der die exportierte Datei erstellt wurde, hinzugefügt werden. Des Weiteren sind erläuternde Beschreibungen oder genauere Informationen zu den enthaltenen Modellen ergänzbar.

## 2.3 Modellsynchronisation

Modellsynchronisation bedeutet zwei (oder mehrere) Modelle zueinander konsistent zu halten, d.h. Änderungen an einem Modell werden auf ein anderes übertragen, falls dieses

den angepassten Bereich abbilden kann, die Änderungen also durch das Zielmodell überhaupt beschrieben werden können. Dementsprechend darf es nicht möglich sein, für ein Ursprungsmodell zwei verschiedene Zielmodelle zu erzeugen. Wird beispielsweise ein Modell A, wie in Abbildung 2.12(a), über zwei verschiedene Wege aus dem gleichen Startzustand in den gleichen Endzustand überführt, müssen sich auch bei dem synchronisierten Modell B, wie in Abbildung 2.12(b), beide Zustände jeweils gleichen, wenn eine korrekte Synchronisierung vorliegt. Dabei beschreiben „Change 1“ und „Change 1’“ bzw. „Change 2“ und „Change 2’“ jeweils die gleiche Änderung im jeweiligen Modell. Unterschiedliche Zustände in einem Modell können hingegen in gleichen Zuständen im Zielmodell resultieren, wenn die verschiedenen Bereiche durch das Zielmodell nicht ausgedrückt werden können.

Bei der Umsetzung einer Synchronisierung von Modellen ist es wünschenswert möglichst




(a) Anpassung Modell A (b) Synchronisierung Modell B

**Abbildung 2.12:** Zwei Modelle resultieren unabhängig von der Reihenfolge der Änderungen im jeweils gleichen Zielmodell

unabhängig von der Struktur und der Größe eines Modells zu bleiben; werden Teile des Modells geändert oder erweitert, muss der Aufwand die Synchronisierung anzupassen überschaubar bleiben. Vor allem sollen die Modelle vollständig unabhängig von der Synchronisierungslogik bleiben; es dürfen also für die Implementierung keine Anpassungen an diesen nötig sein. Häufig ist dies schon ausgeschlossen, weil kein direkter Zugang zu dem Modell existiert, beispielsweise bei der Nutzung einer vorgefertigten Bibliothek. In diesem Fall können grundsätzlich keine Änderungen eingepflegt werden. Des Weiteren dient solche zusätzlich eingefügte Funktionalität ausschließlich dem Zweck die Synchronisierung zu ermöglichen und gehört daher nicht in ein Datenmodell. Ein weiteres großes Problem ist, dass auf dem ursprünglichen Modell existierende Projekte nicht grundsätzlich weiter bearbeitet werden können. Nur durch Anpassungen, wie eine Umwandlung in das erweiterte Modell,

können diese weiter genutzt werden. Doch gerade dies ist häufig, wie auch im Rahmen dieser Arbeit, wünschenswert. Auf Grund dieser Voraussetzung fallen bereits verschiedene Herangehensweisen heraus, die Synchronisierung umzusetzen.

Im Folgenden werden einige Ansätze Modelle zu synchronisieren genauer erläutert.

### 2.3.1 Observer/Listener

Das Observer-Muster [4] beschreibt ein Programmierparadigma um Statusänderungen eines Objektes an eine beliebige Anzahl anderer Objekte auszuliefern, ohne dass das Ursprungsobjekt genauere Informationen über die Empfänger haben muss. In Abbildung 2.13 ist eine Beispielimplementierung für das Muster zu sehen. Die Basis bilden das Interface *Observer* und die abstrakte Klasse *Observable*. Beide sind durch eine unidirektionale Assoziation miteinander verbunden, die bedeutet, dass sich beliebige Implementationen von *Observer* an einem *Observable* registrieren können. Ein *Observer* weiß nicht, an welchen Objekten er registriert ist. *Observable* stellt zudem die Methode `notifyObservers(value:Object)` bereit. Dort werden alle Attributänderungen an die angemeldeten Objekte weitergeleitet. Dazu wird die Methode `update(observable:Observable, value:Object)` von jedem *Observer* aufgerufen und sowohl das geänderte Objekt als auch der neue Wert übertragen.

In der Beispielimplementierung erweitert die Klasse *House Observable*. Neben ihrem ein-

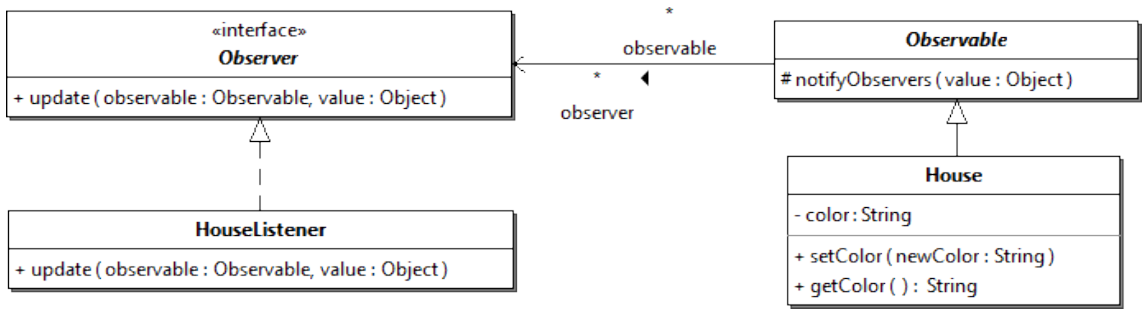



**Abbildung 2.13:** Einfaches Beispiel einer Observer-Struktur

zigen Attribut *color*, enthält sie zwei Methoden. Erstens die Methode zum Auslesen des Attributs (`getName()`) und zum Zweitem eine Methode zum Setzen eines neuen Wertes (`setColor(newColor:String)`). Zur Realisierung des Musters wird in `setColor` die Methode `notifyObservers(value:Object)` aufgerufen, um den neuen Attributwert an alle Listener weiterzuleiten. Die Implementierung *HouseListener* von *Observer*

soll an jedes erzeugte Objekt vom Typ *House* angemeldet werden. Dazu muss die Methode `update(observable:Observable, value:Object)` implementiert werden, um auf die Attributänderungen reagieren zu können. Wie im obigen Beispiel wird dieses Muster häufig genutzt, wenn Änderungen an dem Datenmodell einer Anwendung Auswirkungen in anderen Programmteilen haben sollen. Diese Programmlogik sollte nicht in das Modell integriert, sondern in einen eigenen Programmteil ausgelagert werden. Damit folgt man auch dem Model-View-Controller-Muster (MVC) [14], das einen modularen Programmaufbau mit voneinander unabhängigen Bereichen beschreibt. Ein weiteres Einsatzgebiet liegt in Bibliotheken. Diese bieten häufig für Bereiche, die für auf ihnen aufbauende Anwendungen relevant sein können, die Möglichkeit für die Nutzung von *Observern*. Dadurch können sie, sobald sie in das Programm eingebunden sind, dieses über Statusänderungen informieren.

Auf die Modellsynchronisation bezogen wird durch dieses Muster ermöglicht, auf Änderungen in den Modellelementen reagieren zu können, ohne dessen Strukturen zu beeinflussen oder erweitern zu müssen. Einzige Voraussetzung ist, dass die Modellimplementation das Muster unterstützt, also eine Anmeldung von Listenern ermöglicht und Attributänderungen an diese weitergibt. Durch diese unmittelbare und selektive Weitergabe der Änderungen kann gewährleistet werden, dass zum einen Veränderungen schnell synchronisiert werden können und zum Zweiten nur der abgeänderte Bereich auch behandelt wird.

### 2.3.2 Graph Grammatiken

Eine Möglichkeit zur Umsetzung von Modellsynchronisationen durch Modelltransformationen bieten Graph Grammatiken. Ein als Graph dargestelltes Modell besteht aus einer Menge an Knoten und Kanten, wobei jede Kante einen Start- und einen Zielknoten besitzt. Neben dem Startgraphen gehören zu einer Graph Grammatik auch eine beliebige Menge von Graphersetzungsregeln. Jede Regel besteht aus einer linken und einer rechten Seite. Auf der linken Seite wird der Ursprungszustand als Objektstruktur dargestellt, d.h. auf jeden zur linken Regelseite isomorphen Teilgraph eines untersuchten Graphs wird diese Regel angewandt. Die rechte Seite stellt den Zielzustand dar. Dieser wird durch Änderung oder das Löschen existierender Objekte bzw. das Anlegen neuer Objekte erreicht. Eine Regelanwendung besteht aus der Suche nach allen isomorphen Teilgraphen und deren Ersetzung durch die Zielgraphen auf der rechten Regelseite. Diese Regeln sind somit darauf beschränkt, Änderungen an der Objektwelt eines Modells abzubilden.

Die folgenden drei Abbildungen zeigen einen Startgraphen, eine Graphersetzungsregel und

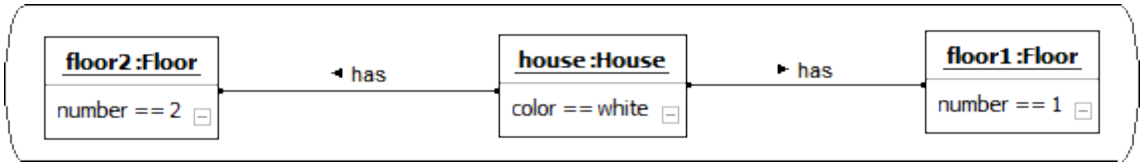



**Abbildung 2.14:** Graph vor der Regelanwendung

einen daraus resultierenden Graph. Der Startgraph in Abbildung 2.14 greift das Beispiel aus den vorherigen Abschnitten wieder auf und zeigt das Objektdiagramm für ein Haus mit zwei Etagen. Die Graphersetzungsregel in Abbildung 2.15 sucht über die linke Seite alle Teilgraphen, die ein Haus mit assoziierter Etage enthalten. Durch die rechte Seite wird jede gefundene Etage durch zwei neue Räume ergänzt. Der Endgraph in Abbildung 2.16 resul-

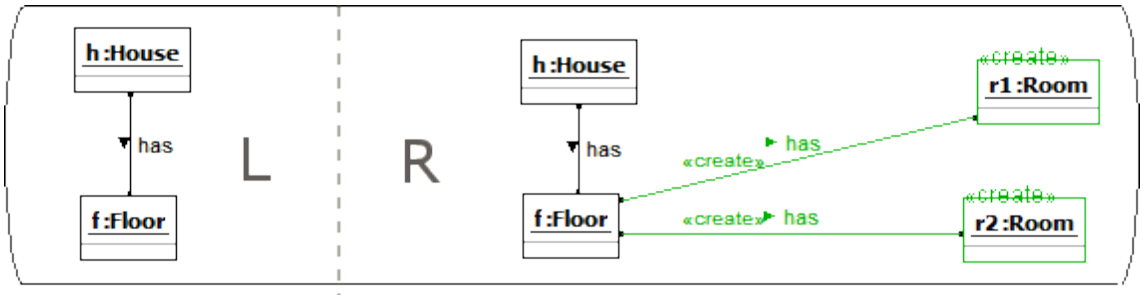



**Abbildung 2.15:** Beispielregel für eine Graph Grammatik

tiert aus dem Startgraph und der zweimaligen Anwendung der Graphersetzungsregel, d.h. es wurden auf Grund der beiden Etagen zwei unterschiedliche Teilgraphen gefunden und diesen jeweils zwei Räume hinzugefügt.

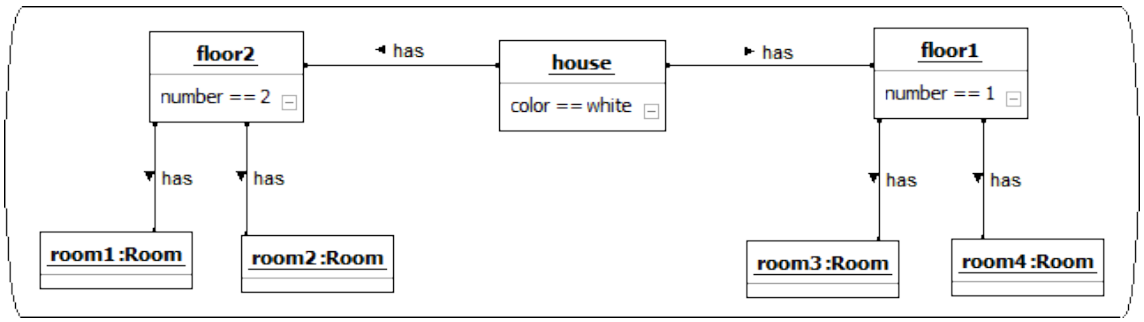
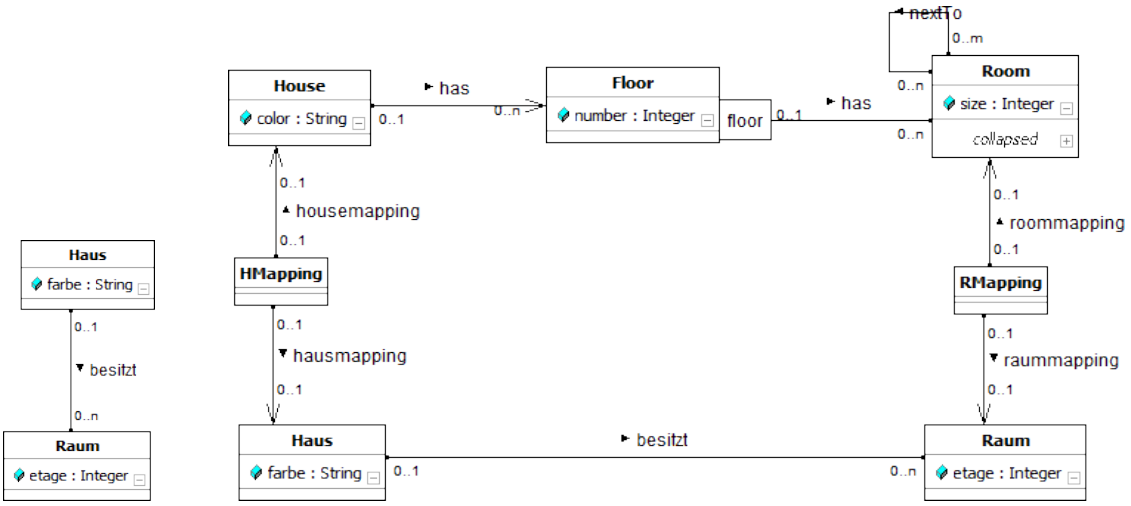



**Abbildung 2.16:** Graph nach allen möglichen Anwendungen der Regel

### 2.3.3 Triple Graph Grammatiken

Auf den Graph Grammatiken aufbauend stellte Andy Schürr [16] 1994 eine zusätzliche Technik für Modelltransformationen vor, die Triple Graph Grammatiken (TGG). TGGs erweitern die Regeln der Graph Grammatiken um eine zusätzliche mittlere Spalte. Dadurch können innerhalb dieser Regeln verschiedene Modelle bzw. Modellarten genutzt und ineinander transformiert werden.

Die folgenden Beispiele wurden durch [10] inspiriert. Man möchte zwei Modelle A (be-




(a) Modell B (b) Modell TGG-Mapping

**Abbildung 2.17:** Beispielmodelle

kanntes Modell aus Abbildung 2.3) und B (siehe Abbildung 2.17(a)) ineinander transformieren. Der Hauptunterschied zwischen beiden Modellen ist ein fehlendes Äquivalent zu der Klasse *Floor* in Modell B. Im Gegenzug weiß dort jeder Raum über das Attribut „etage“ in welchem Stockwerk er liegt. Eine der TGG-Regeln für dieses Beispiel zeigt die Abbildung 2.18. Wie bereits erwähnt ist diese in drei Spalten eingeteilt. Die linke Spalte enthält einen Teilgraph basierend auf Modell A, der Teilgraph der rechten Spalte basiert auf Modell B. Die mittlere Spalte enthält ein drittes Modell (siehe Abbildung 2.17(b)) mit Mapping-Klassen, die zueinander gehörende Objekte aus den Modellen A und B verbinden. Dazu werden von dem mittleren Objekt ausgehend unidirektionale Kanten zu den Zielobjekt geführt.

Die Verwandtschaft zu den Graph Grammatiken zeigt sich durch die Ähnlichkeit der linken und rechten Seite in der Regelbeschreibung. Diese Regel ist allerdings bidirektional anwendbar, d.h. wird ein isomorpher Teilgraph zur linken Seite gefunden, wird der Graph für

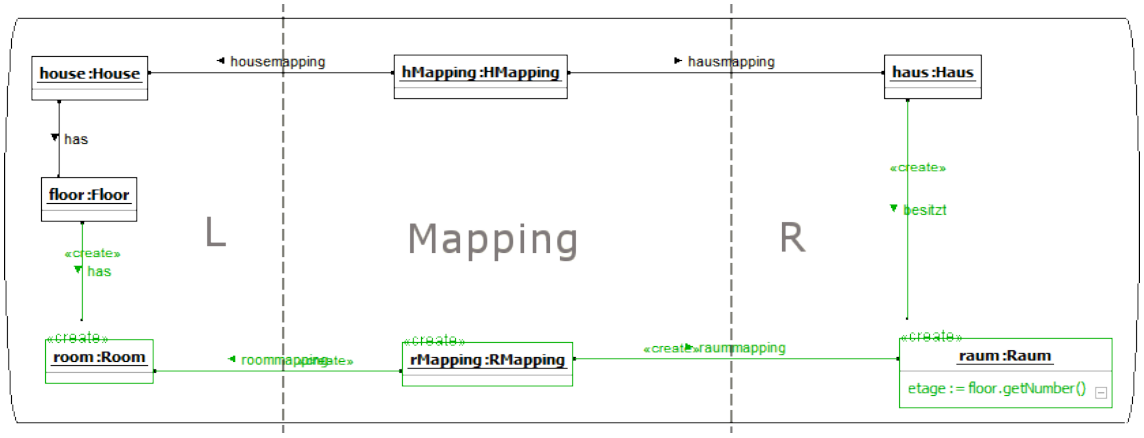



**Abbildung 2.18:** Beispielregel für eine Triple Graph Grammatik

das andere Modell durch die Objekte auf der rechten Seite ergänzt; existiert ein Teilgraph zur rechten Seite, erfolgt die Anpassung auf die linke Seite der Regel. Die Mapping-Objekte in der mittleren Spalte helfen einmal gefundene bzw. erzeugte zueinander gehörige Objekte in den beiden Modellen dauerhaft zu markieren. Somit wird diese existierende Verbindung auch bei späteren Regelausführungen wiedergefunden und verhindert Fehler während den Transformationen. So werden keine Objekte unnötig erzeugt oder Regeln fälschlicherweise mehrfach angewendet. Wird also ein *Raum* gefunden, für den kein Gegenstück im anderen Modell existiert, wird dieser erzeugt und beide Objekte mit Hilfe des Mapping-Modells verbunden. Je nach Modell wird anschließend entweder der *Raum* dem richtigen *Floor*-Objekt zugewiesen oder sein Attribut „etage“ wird in den entsprechenden geändert.

Für die Verwendung von TGG-Regeln sind verschiedene Anwendungsszenarien denkbar. Zum Ersten kann mit ihnen ein Modell A in ein (noch nicht existierendes) Modell B transformiert werden, indem solange Regeln auf Modell A angewendet werden, bis sich in dem neuen Modell B keine Änderungen mehr zeigen. Zum Zweiten ist die eine Integration zweier Modelle möglich. In diesem Szenario existieren beide Modelle bereits und mit Hilfe der Regeln wird versucht zueinander passende Teilgraphen zu finden und mit Hilfe dieser die Mapping-Objekte zu erstellen. Falls dies nicht mit allen Objekten gelingt, gelangt man zum Dritten Szenario, der Modellsynchronisation. Dort existieren beide Modelle und die Mapping-Objekte für passende Elementen aus den Modellen bereits. Dieses Szenario kommt beispielsweise bei einer unvollständigen Modellintegration oder der nachträglichen Veränderung eines oder beider Modelle zum Tragen. Durch die Anwendung der Regeln sollen diese wieder synchronisiert werden. In [16] werden die Grundlagen und Erweiterungen von TGGs genauer ausgeführt.

## 2.4 Fujaba

Bei der Fujaba Tool Suite handelt es sich um ein CASE-Tool, das in Zusammenarbeit verschiedener Universitäten (unter anderem Paderborn und Kassel) entstanden ist und weiterentwickelt wird. Professor Zündorf erläutert in [20] Softwareentwicklung mit UML mittels *Story Driven Modeling* (SDM). Dieser in Abschnitt 2.4.1 näher erläuterte Ansatz hebt Fujaba von anderen UML-Modellierungs-Tools ab, weil er über die Festlegung der statischen Strukturen hinaus zusätzliche Unterstützung bei der Analyse, dem Design und der Implementation bietet. Das Fujaba-Metamodell, welches das erste zu synchronisierende Modell dieser Arbeit darstellt, basiert auf einer der frühen UML-Spezifikationen um Version 1.4. In Abschnitt 2.1.1 wurde bereits der für Klassendiagramme relevante Ausschnitt erläutert.

Wird die Abkürzung Fujaba auf ihre vollständige Bezeichnung „**F**rom **U**ML to **J**ava **A**nd **B**ack **A**gain“ erweitert, zeigt sich dessen zu Grunde liegender Ansatz. Sowohl Quelltext aus in UML modellierten Strukturen generieren, als auch der umgekehrte Weg soll möglich sein. Eine template-basierte Codegenerierung sorgt hierbei für eine einfach zu verwaltende und erweiterbare Lösung. Diese ermöglicht durch Austausch der Templates entweder eine Anpassung, um generierten Quellcode auf eine bestimmte Problemstellung hin zu optimieren oder die Generierung in eine beliebige andere Programmiersprache.

Im Folgenden werden Aspekte von Fujaba genauer beleuchtet, welche für diese Arbeit eine besondere Relevanz haben und in späteren Abschnitten wieder aufgegriffen werden.

In Fujaba werden alle Objekte des Metamodells in Factories[4] erzeugt. Für jede Klasse ist eine eigene Factory registriert. Diese sind in *UMLFlyweightFactory* und *UMLHeavyweightFactory* unterteilt. Die Ersteren erzeugen Objekte automatisch bei Bedarf. In diese Kategorie fallen unveränderliche Objekte, die nur einmal existieren und vielfach wiederverwendet werden sollen, wie primitiven Typen, Stereotypen oder Kardinalitäten. Möchte man beispielsweise einem Attribut den Typ *boolean* zuweisen, wird die Factory nach einem entsprechenden Objekt gefragt. Existiert kein passendes wird es automatisch erzeugt und für die nächsten Anfragen gesichert, bevor es zurückgegeben wird. In einer *UMLHeavyweightFactory* werden die übrigen veränderlichen Objekte wie *UMLClass*, *UMLMethod* oder *UMLAttr* angelegt. Deren Erzeugung muss manuell angeregt werden.

[4] Ein Entwicklungsmuster, das einen Ansatz für die Erzeugung von Objekten beschreibt[4]

#### „Java Feature Abstraction“-Bibliothek

Die „Java Feature Abstraction“-Bibliothek wird von Dr. Christian Schneider in seiner Dissertation[15] vorgestellt. Sie dient als eine abstrakte Zugriffsschicht auf ein Modell, um Lese- und Schreiboperationen auszuführen. Dazu gehört das Anlegen und Löschen von Objekten in einem Modell, die Änderung eines Attributwertes oder das Setzen bzw. Entfernen eines Links zu einem anderen Objekt. Für diese Operationen gibt es verschiedene Handler, wie den *ClassHandler* oder den *FeatureHandler*, die als Anlaufpunkt für alle Zugriffe dienen. Diese Bibliothek bietet durch ihre reflektiven Zugriffsmethoden die Möglichkeit bei Wertabfragen oder - änderungen unabhängig vom angefragten Modell zu bleiben.

In dieser Arbeit wird die „Java Feature Abstraction“-Bibliothek in verschiedenen Bereichen für lesende Zugriffe auf Fujaba-Modelle genutzt.

### 2.4.1 Story Driven Modeling

Das *Story Driven Modeling* (SDM) umfasst von der Analyse über das grundlegende Design bis zur eigentlichen Implementierung fast alle Bereiche des Software-Entwicklungsprozesses. Vor allem durch die leicht verständliche grafische Notation bietet SDM auch Fachfremden einen leichten Einstieg in (existierende) Projekte. In Abbildung 2.19 sind die einzelnen Teilbereiche und die Vorgehensweise grob aufgeschlüsselt zu sehen.

Man beginnt bei der Analyse mit Use-Cases, um die Rahmenbedingungen festzulegen. Es werden die grundlegenden Bereiche und Aufgaben definiert und für die unterschiedlichen Anwendergruppen beschrieben inwiefern der Zugriff bzw. die Arbeit mit einem Bereich möglich sein soll. Diese Use-Cases werden anschließend in Beispielszenarien genauer definiert. Diese werden beim *Story Driven Modeling* als *Story Boards* bezeichnet. Dazu werden mehrere Diagrammarten von UML kombiniert um in graphen-ähnlichen Strukturen eine Objektwelt für das Beispielszenario zu modellieren. Die Veränderungen während eines Szenarios werden in Einzelschritten dargestellt. Dazu wird zuerst das Startszenario mit Objekten, Attributen, Assoziation usw. inklusive dem Aufruf von (möglicherweise noch nicht existierenden) Methoden angelegt. Die gewünschten Auswirkungen dieser Methodenaufrufe resultieren in einem Endszenario. Zwischenschritte können zum besseren Verständnis der Vorgänge ebenfalls modelliert werden. In Abbildung 2.20 ist ein Beispiel für ein *Story Board* mit Start- und Zielszenario zu sehen.

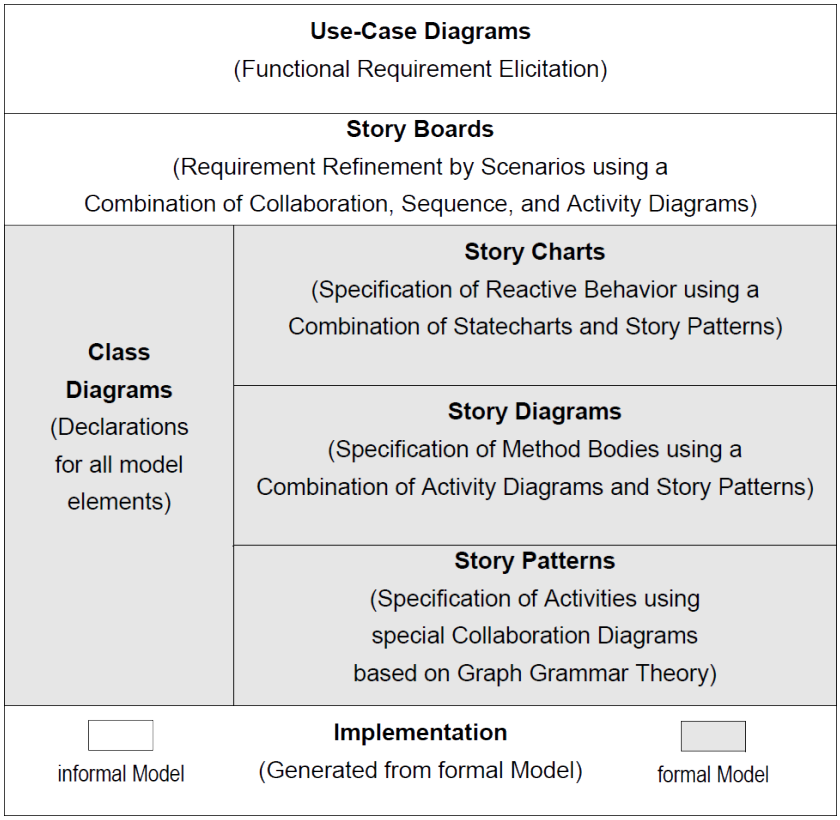



**Abbildung 2.19:** Übersicht des Story Driven Modeling (Quelle: [20], Seite 13)

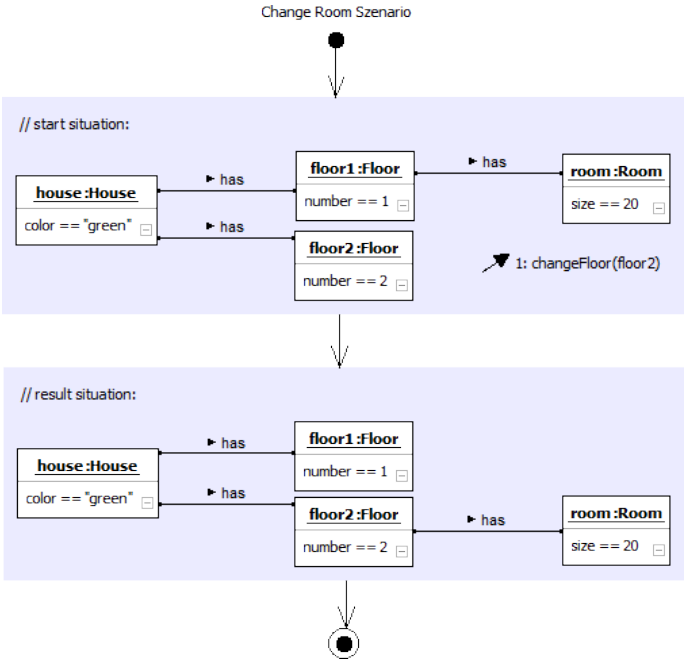



**Abbildung 2.20:** Beispiel für ein *Story Board*

Ist das Entwicklungsziel durch ein *Story Board* deutlicher umrissen, können als nächstes die in den Szenarien benötigten Methoden mit Inhalt gefüllt werden. Dafür werden sogenannte *Story Diagrams* verwendet, eine Kombination aus Aktivitätsdiagramm und den *Story Pattern*. Während der aus den Aktivitätsdiagrammen übernommene Teil für den Kontrollfluss zuständig ist, sorgen die *Story Patterns* für die Modifikationen an den Objekten, beispielsweise Attributänderungen. In Abbildung 2.21 ist ein einfaches *Story Diagram* dargestellt. Dort ist zu erkennen, wie mit Hilfe von Objektdiagrammen in den einzelnen *Story Pattern* ausgewählte Bereiche der Objektwelt dargestellt werden. Die einzelnen Module sind aber nicht strikt voneinander getrennt, sondern bauen aufeinander auf. Wird beispielsweise in einem *Story Board* eine bisher unbekannte Klasse angelegt, ist sie anschließend auch in dem parallel aufgebauten Klassendiagramm zu finden und somit in allen übrigen Bereichen nutzbar. Dieses Klassendiagramm kann zusätzlich auch separat gepflegt und erweitert werden, um es an geänderte Ansprüche anzupassen.

Am Ende dieser einzelnen Schritte steht die Generierung des kompilier- und ausführbaren

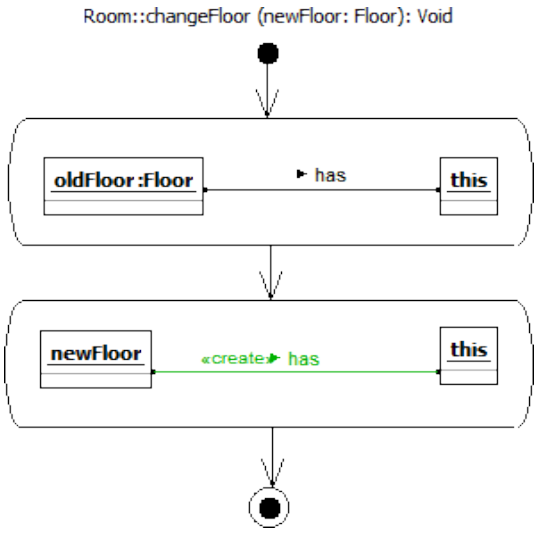



**Abbildung 2.21:** Beispiel für ein *Story Diagram*

Quelltextes aus den Klassendiagrammen und *Story Diagrams*. Die *Story Boards* hingegen können zum Testen der Implementierung verwendet werden. Dazu wird das Startszenario erzeugt und hierauf die implementierten Methoden aufgerufen. Auf das Beispiel in Abbildung 2.20 bezogen bedeutet dies, dass die vier Objekte, wie dargestellt, erzeugt werden, bevor auf room die Methode „changeFloor(floor2)“ aufgerufen wird. Die resultierende Objektwelt wird anschließend mit dem Zielszenario verglichen um die Korrektheit der Implementierung abzusichern.

## 2.5 UML Lab

UML Lab ist eine von Yatta Solutions GmbH[5] entwickelte Modeling IDE. Dabei handelt es sich um ein kommerzielles Produkt, welches einen besonderen Schwerpunkt auf ein ausgefeiltes Round-Trip-Engineering (siehe Abschnitt 2.5.1) legt.

Der zentrale Arbeitsschwerpunkt mit UML Lab liegt derzeit im Klassendiagramm-Editor. Dieser wird durch eine template-basierte und somit auf die eigenen Ansprüche anpassbare Codegenerierung ergänzt. Durch ein Reverse-Engineering bereits existierenden oder des generierten Quellcodes wird der Kreis für ein Round-Trip-Engineering geschlossen.

Als Metamodell wird bei UML Lab die Implementation der UML2-Spezifikation auf EMF-Basis eingesetzt. Dieses wurde in Abschnitt 2.1.2 genauer vorgestellt.

### 2.5.1 Round-Trip-Engineering

Mit Round-Trip-Engineering wird die (üblicherweise automatische) Erhaltung der Konsistenz zwischen Modell und zugehörigem Quelltext bezeichnet. Das erfolgt über eine Codegenerierung für den Weg von Modell nach Quelltext und durch das Reverse Engineering des Quelltextes für die Gegenrichtung. Auf diesem Weg wird idealerweise versucht Änderungen automatisch beidseitig einzupflegen, d.h. eine Anpassung des Quelltextes modifiziert das Modell und Änderungen im Modell können durch die Codegenerierung auf den Quelltext übertragen werden. Als weiterer Vorteil zeigt sich, dass es nicht erforderlich ist eine 1-zu-1 Abbildung zwischen Modell und generiertem Code vorzuhalten, d.h das abstrakte Modell darf sich von den Implementationsdetails abheben. Während sich das abstrakte Modell auf die Kernaspekte der Anwendungsdomäne beschränkt, bleiben die Details im Quelltext davon unberührt. Zu diesen Details können beispielsweise Dokumentationseinträge oder Methodenrümpfe zählen. Diese müssen, falls sie beim Reverse Engineering nicht interpretiert werden, bei der Codegenerierung aber bedacht werden, um das Überschreiben und somit den Informationsverlust zu verhindern.

[5] Homepage unter http://www.yatta.de

## 2.6 Verwandte Arbeiten

Betrachtet man ausschließlich Modelltransformationen, so gibt es viele existierende Ansätze. Dazu gehören beispielsweise die ATLAS Transformation Language (ATL)[1], VIsual Automated model TRAnsformations (VIATRA)[17], Graph Rewrite And Transformation (GReAT)[7] oder QVT (Query/View/Transformation)[12]. Erweitern sich die Anforderungen auf die Modellsynchronisation sind vor allem die Triple Graph Grammatiken (TGG)[16] zu erwähnen.

In [18] wird auf Basis von Triple Graph Grammatiken eine Technik zur inkrementellen Modellsynchronisation beschrieben. Dabei geht es vor allem darum, die Stärken der TGG bei Modellintegration, -transformation und -synchronisation in einem festgelegten Rahmen einzusetzen, d.h. es wird ein Algorithmus beschrieben, der eine Synchronisation durch die Anwendung von TGG-Regeln möglichst effizient durchführt. Zusätzlich werden Werkzeuge zur automatischen oder manuellen Erstellung der nötigen TGG-Regeln vorgestellt; vergleichbar dazu auch [6].

In wird [19] ein Ansatz zur Modellsynchronisation auf Basis von ATL aufgezeigt. Dazu wird neben dem Ursprungs- und dem Zielmodell zusätzlich ein verändertes Ursprungs- und Zielmodell mit zu synchronisierenden Inhalten benötigt. Während eine Modelltransformation die Neugenerierung des Zielmodells anstößt und damit im Verlust des zusätzlichen Inhalts resultiert, wird dies mit dem beschriebenen Algorithmus umgangen. Dazu werden die Änderungen miteinander kombiniert, indem jeweils Unterschiede zwischen zwei Modellen extrahiert und anschließend auf ein drittes Modell angewendet werden. Dadurch ergibt sich Schritt für Schritt ein synchronisiertes Ursprungs- bzw. Zielmodell.

Eine weitere Alternative wird in [8] präsentiert. Dieser basiert auf der auf Graphen anwendbaren Anfragesprache UnQL[3]. Der vorgestellte Ansatz beruht auf der Anreicherung des Ursprungsgraphen um zusätzliche Informationen, um etwaige Veränderungen im Ursprungs- oder Zielgraphen auf den Gegengraph übertragen zu können. Dadurch kann ein bidirektionaler Abgleich, also die Synchronisation, ermöglicht werden.

# 3 Anwendungsszenarien für die Synchronisation von zwei Modellen

In den folgenden Abschnitten werden drei verschiedene Anwendungsszenarien für den Synchronisierungsmechanismus vorgestellt. Im ersten Use-Case werden Änderungen an einem Fujaba-Modell in dem zugehörige in UML2-Modell synchronisiert. Im zweiten erfolgt die Synchronisierung in entgegengesetzter Richtung. Der dritte Use-Case beschäftigt sich mit der Vorgehensweise beim Laden von geänderten Projekten.

## 3.1 Synchronisierung ausgehend von dem Fujaba-Metamodell auf das UML2-Metamodell

**Szenario:** Das Ausgangsszenario ist in den Abbildungen 3.1 und 3.2 dargestellt. Dabei handelt es sich um zwei Objektdiagramme. Das Erste basiert auf dem Fujaba-Metamodell, das Zweite auf dem Metamodell von UML2. Die Diagramme haben einen identischen Inhalt und sind als verknüpft anzusehen, d.h. alle zukünftigen Änderungen werden synchronisiert. Dargestellt wird eine Klasse *House* als Objekt c1, die in *Package* p1 enthalten ist. Diese

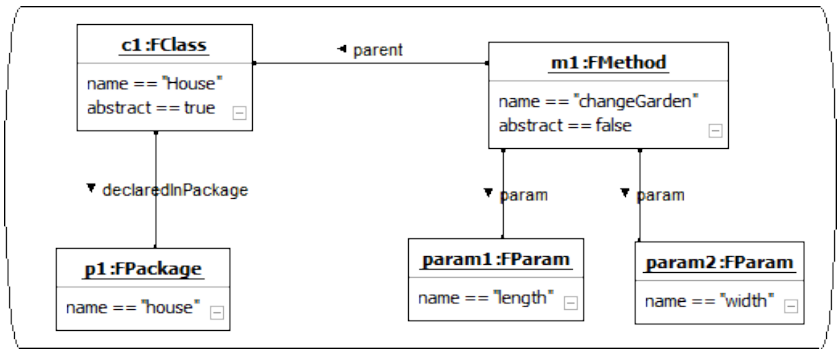



**Abbildung 3.1:** Ausgangsszenario im Fujaba-Metamodell als Objektdiagramm

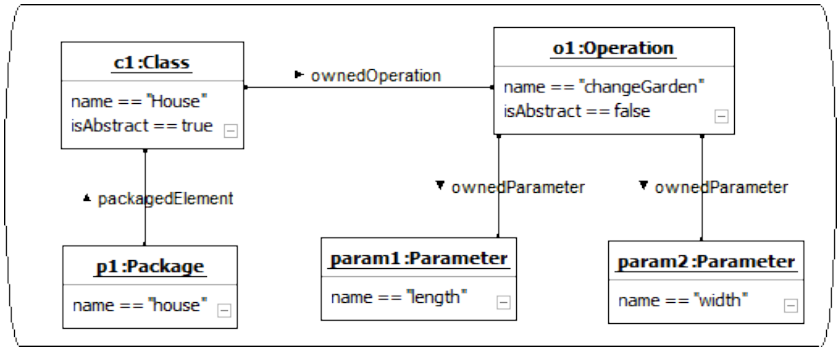



**Abbildung 3.2:** Ausgangsszenario im UML2-Metamodell als Objektdiagramm

Klasse besitzt die Methode „changeGarden“ (Objekte m1 bzw. o1) mit den zwei Parametern „length“ und „width“. Die dargestellten Informationen wurden auf ein Minimum reduziert, um die wichtigsten Aspekte herausarbeiten zu können. So werden neben vielen Attributen beispielsweise die Typen der Parameter oder der Typ des Rückgabewerts der Methode eingespart, weil diese für den Use-Case nicht relevant sind.

**Ablauf:** Veränderungen werden in diesem Use-Case ausschließlich am Modell in Fujaba gemacht. Die Anpassungen auf Seiten von UML2 werden automatisch durch die Synchronisierung eingefügt. Daher ist zuerst der Endzustand in Fujaba relevant. Dieser ist in Abbildung 3.3 zu sehen. In Bezug auf das Ausgangsszenario sind zwei Unterschiede zu sehen. Zum Ersten ist das Attribut „name“ des Objektes c1 und zum Zweiten das Attribut „abstract“ des Objektes m1 geändert, d.h. die Klasse *House* wird umbenannt und die Methode „changeGarden“ zu einer abstrakten umgewandelt. Der Zustand der Objektwelt auf Seiten

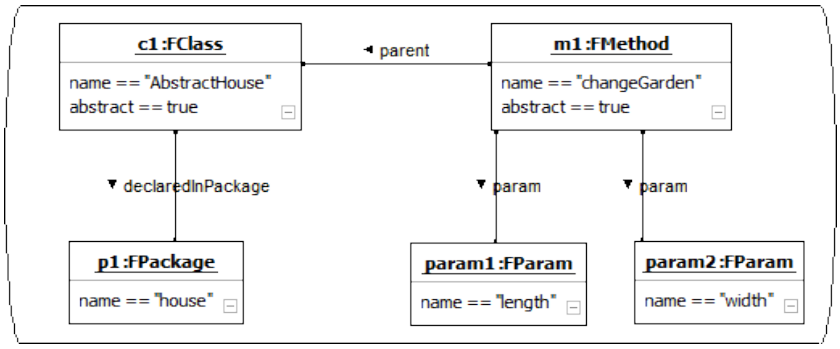



**Abbildung 3.3:** Endzustand im Fujaba-Metamodell als Objektdiagramm

von UML2 nach der Synchronisierung ist in Abbildung 3.4 zu sehen. Im Vergleich zum Fujaba-Modell sind also automatisch die gleichen Attribute angepasst worden.

Dieser Use-Case ist als Beispiel zu sehen wie sich eine Synchronisation von Strukturen,

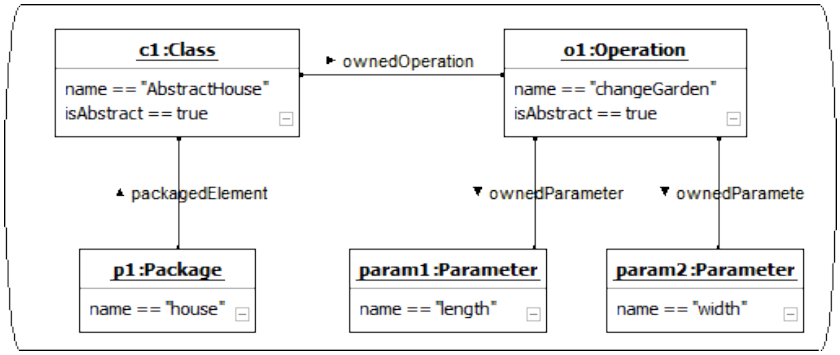



**Abbildung 3.4:** Endzustand im UML2-Metamodell als Objektdiagramm

die in beiden Metamodellen äquivalent sind, auswirkt. In diesem Fall handelt es sich um Attribute, die einen vergleichbaren Namen tragen und, vor allem, desselben Typs sind.

## 3.2 Synchronisierung ausgehend von dem UML2-Metamodell auf das Fujaba-Metamodell

**Szenario:** Das Ausgangszenario ist in den Abbildungen 3.5 (UML2-Metamodell) und 3.6 (Fujaba-Metamodell) dargestellt. In beiden Diagrammen ist die gleiche Struktur modelliert. Sie sind zusätzlich als vollständig synchronisiert und miteinander verknüpft zu betrachten. Zueinander gehörige Objekte tragen jeweils denselben Bezeichner, beispielsweise die Objekte c1, einmal als Instanz von *FClass* und einmal als Instanz von *Class*.

Das Basismodell besteht aus der Paketstruktur „house.elements.windows“, in das die Klas-

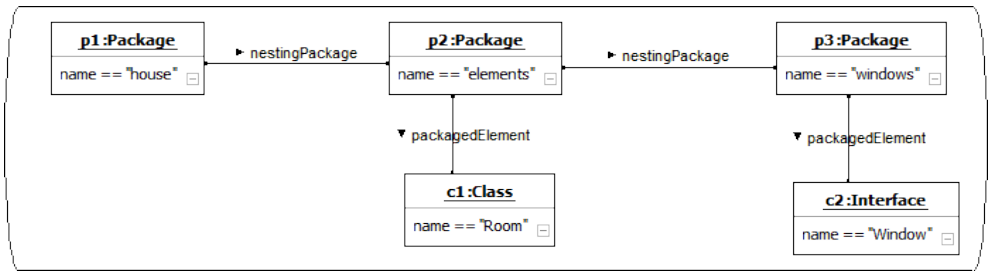



**Abbildung 3.5:** Ausgangsszenario im UML2-Metamodell als Objektdiagramm

se *Room* und das Interface *Window* eingebettet ist. Während für Interfaces in UML2 eine eigenständige Klasse existiert, wird diese in Fujaba durch die Markierung einer *FClass*-Instanz mit dem Stereotyp „interface“ erzeugt.

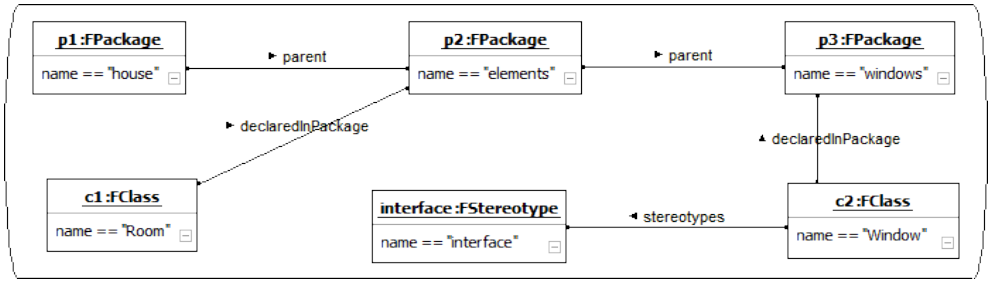



**Abbildung 3.6:** Ausgangsszenario im Fujaba-Metamodell als Objektdiagramm

**Ablauf:** Die manuellen Veränderungen erfolgen in diesem Use-Case in UML2 und werden durch die Synchronisierung auf das Modell in Fujaba übertragen. Betrachtet man daher zuerst den Endzustand in UML2 als Objektdiagramm in Abbildung 3.7, ist eine Erweiterung um die sieben Objekte auf der rechten Seite zu erkennen. Diese resultieren daraus, dass eine bidirektionale Assoziation zwischen *Room* und *Window* gezogen wird. Neben dem Objekt a1 für die Assoziation, sind die beiden Objekte prop1 und prop2 vom Typ *Property* neu. Dabei handelt es sich um die Attribute, die jeweils die Instanzen der Klasse am gegenteiligen Ende der Assoziation verwalten. Durch die Objekte o1 bis o4 werden schließlich die Kardinalitäten der Attribute, unterteilt in untere und obere Grenze, beschrieben.

Während der Erzeugung der Assoziation in UML2 erfolgt der parallele Aufbau in Fuja-

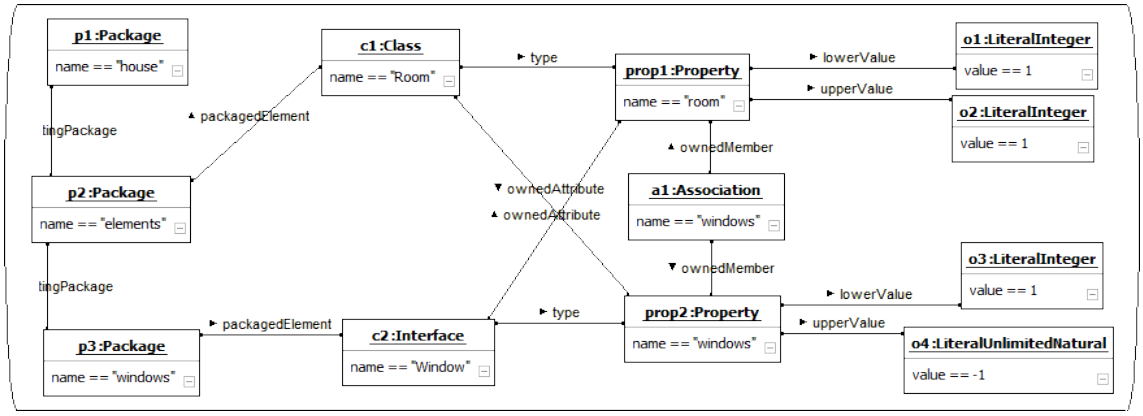



**Abbildung 3.7:** Endzustand im UML2-Metamodell als Objektdiagramm

ba, welcher in dem Endzustand aus Abbildung 3.7 resultiert. Die Anzahl der neu hinzugekommen Objekte beschränkt sich auf fünf, weil die Kardinalitäten auf eine andere Weise ausgedrückt werden. Die markantesten Unterschiede zwischen den beiden neuen Modellbereichen befinden sich zum Ersten in der expliziten Unterscheidung zwischen linker und rechter Seite einer Assoziation über die Kanten „rightRole“ bzw. „leftRole“. Zum Zweiten wird in Fujaba zwischen Attributen mit einem primitiven Typ (Klasse *FAttr*) unterschieden

und solchen, die Teile einer Assoziation sind. Letztere werden durch die Klasse *FRole* modelliert. Als Drittes werden Kardinalitäten über ein einzelnes Objekt dargestellt, welches die obere und untere Begrenzung beinhaltet.

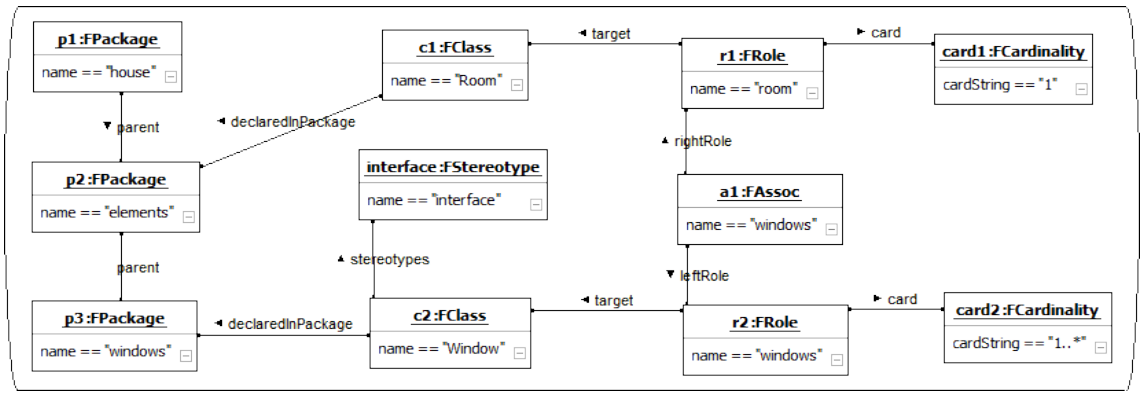



**Abbildung 3.8:** Endzustand im Fujaba-Metamodell als Objektdiagramm

## 3.3 Separat veränderte Projekte neu laden

**Szenario:** Das Ausgangszenario ist in den Abbildungen 3.9 und 3.10 dargestellt. Auch bei diesen zwei Objektdiagrammen basiert das erste auf dem Fujaba-Metamodell und das zweite auf dem Metamodell von UML2. Es handelt sich um zwei teilweise bereits synchronisierte Modelle. Diese werden anschließend separat bearbeitet und im Anschluss wieder eingeladen um ein zweites Mal miteinander synchronisiert zu werden.

Das Objektdiagramm in Abbildung 3.9 besteht aus den beiden Klassen *House* (Objekt c1)

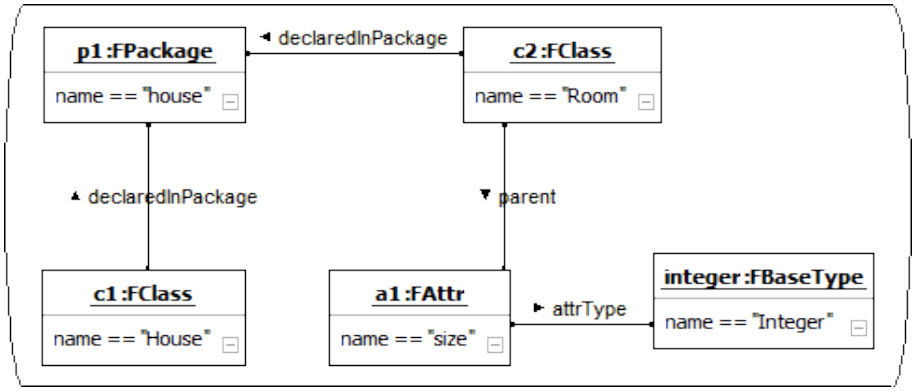



**Abbildung 3.9:** Ausgangsszenario im Fujaba-Metamodell als Objektdiagramm

und *Room* (Objekt c2), welche im gemeinsamen Paket p1 liegen. Die Klasse *Room* wird

durch das Attribut „size“ des Typs *Integer* ergänzt. Im Objektdiagramm in Abbildung 3.10 wird die Klasse *House* (Objekt c1) im Paket p1 um die Methode „addNeighbour“ (Objekt m1) inklusive einem zugehörigen Parameter (Objekt param1) ergänzt.

Die Gemeinsamkeiten der beiden Diagramme beschränken sich auf die Objekte c1 und p1.

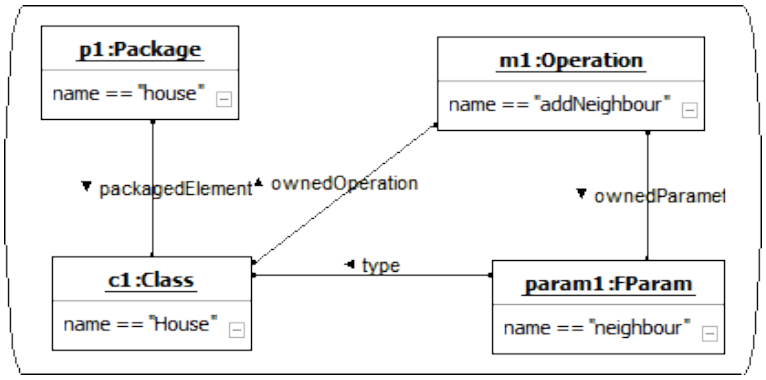



**Abbildung 3.10:** Ausgangsszenario im UML2-Metamodell als Objektdiagramm

Dies bedeutet, dass diese als synchronisiert markiert sind und auch nach dem Schließen und erneuten Laden automatisch als zueinander passend erkannt werden. Alle übrigen Objekte werden den jeweiligen Projekten hinzugefügt, während keine aktive Synchronisierung zwischen den Modellen existiert.

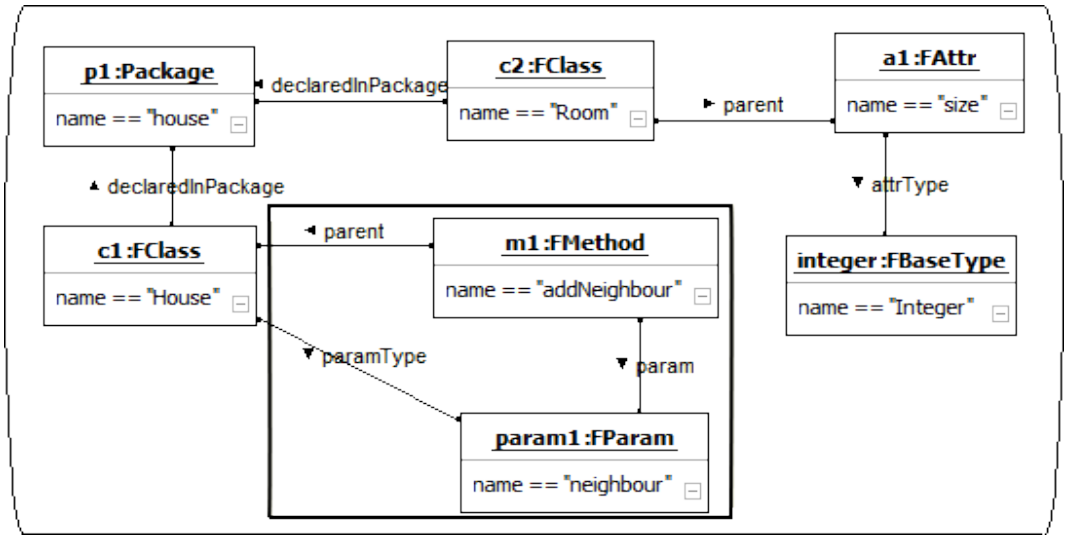



**Abbildung 3.11:** Endzustand im Fujaba-Metamodell als Objektdiagramm

**Ablauf:** Nach dem Laden beider Projekte werden die bereits bekannten Objekte c1 und p1 aus den Modellen einander zugeordnet. Für alle übrigen Objekte wird ein passendes Gegenstück im anderen Modell gesucht bzw. erzeugt. Daraus resultieren die Abbildungen 3.11

und 3.12.

Das Objektdiagramm in Abbildung 3.11 ist im Vergleich zu dem Ausgangsszenario um die Objekte m1 und param1 erweitert worden. Die im UML2-Diagramm zusätzlich hinzugefügten Elemente werden also inklusive aller Attribute und Assoziationen ergänzt. Abgesehen von den Bezeichnern ist die Struktur der neuen Elemente in beiden Modelle vergleichbar.

Das Objektdiagramm von UML2 wird, wie in Abbildung 3.12 zu sehen, um die Objekte c2,

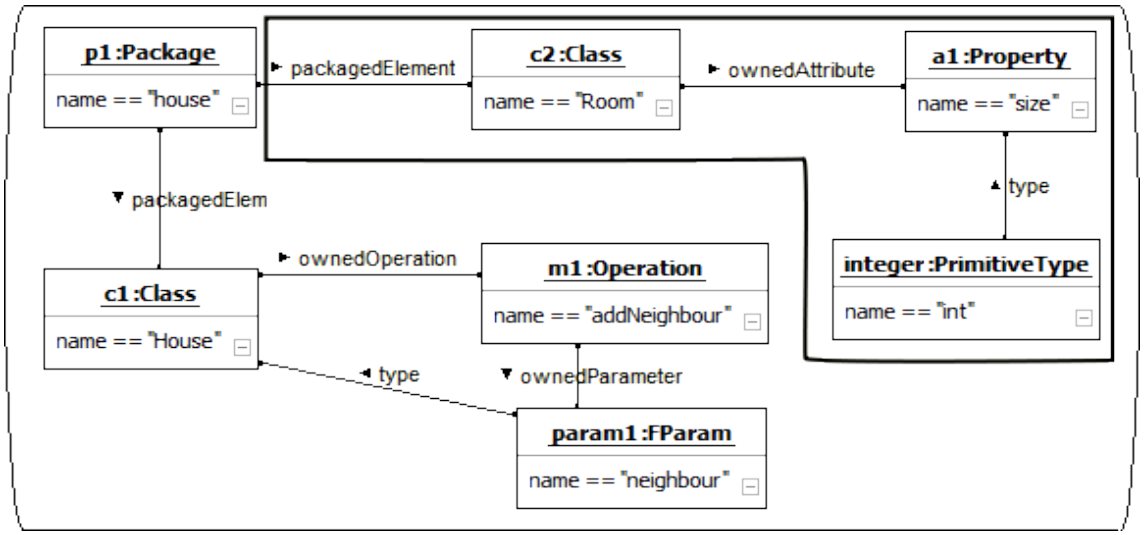



**Abbildung 3.12:** Endzustand im UML2-Metamodell als Objektdiagramm

a1 und integer erweitert. Auch in diesem Fall werden die in Fujaba hinzugefügten Elemente inklusive der Attribute und Assoziationen synchronisiert.

# 4 Umsetzung

In den folgenden Abschnitten werden die zentralen und wichtigsten Themen der Implementierung erläutert. Dabei werden verschiedene mögliche Ansätze der einzelnen Bereiche und die Gründe für die Wahl einer bestimmten Alternative aufgezeigt.

## 4.1 Alternative Ansätze für den Abgleich von Änderungen zwischen den Modellen

Betrachtet man die Problemstellung fällt zuerst die Frage ins Auge, wie man Änderungen an einem Objekt des einen Modells auf sein Gegenstück im zweiten Modell übertragt. Für die Lösung kommen verschiedene Strategien als alternative Grundlage in Frage: die Klassenadapter, die Objektadapter und die Triple Graph Grammatiken. Im Folgenden werden diese drei Ansätze kurz im Allgemeinen und in Bezug auf die Problemstellung inklusive der Vor- und Nachteile ihres Einsatzes erläutert.

### 4.1.1 Klassenadapter

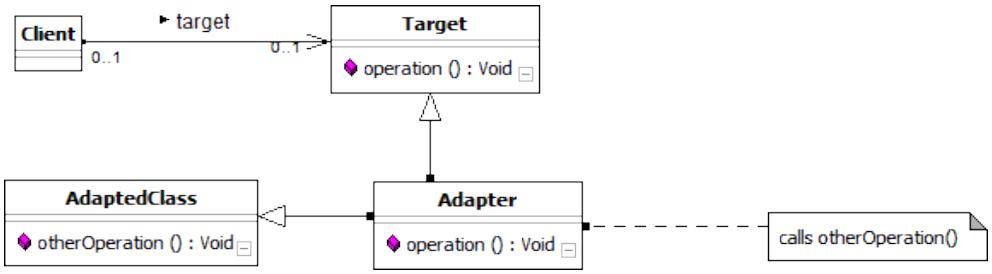



**Abbildung 4.1:** Klassendiagramm für einen Klassenadapter

Ein Klassenadapter als Entwicklungsmuster nach Definition in [4] beschreibt eine Adapterklasse, welche durch Mehrfachvererbung zum einen die zu adaptierende Klasse und zum

anderen die zu nutzende Schnittstelle erweitert. Hierbei werden in der Regel die Methoden der Schnittstelle von außen erreichbar gehalten, die zu adaptierende Klasse hingegen wird versteckt. Eingehende Aufrufe von außerhalb werden innerhalb der Klasse weiter delegiert. Hierzu muss eine Klasse aus dem Modell zu einer Adapterklasse erweitert werden. Auf diesem Weg kann direkt im Modell zum einen ein verändertes Verhalten und zum Anderen erweiterte Funktionalität beschrieben werden. Man gelangt über diesen Weg zu einer umfangreichen Einflussnahme auf das Modell und hat vor allem Möglichkeiten Funktionen einer Klasse anzupassen oder zu beschneiden.

In Abbildung 4.1 ist der Klassenadapter in einer beispielhaften Umsetzung als Klassendiagramm zu sehen. Abbildung 4.2 zeigt zusätzlich den Ablauf für den Aufruf der Methode `operation()` als Sequenzdiagramm, im Besonderen die Weiterleitung auf die Methode `otherOperation()` durch Selbstdelegation.

Auf die gegebene Problemstellung übertragen kommen zwei Hauptprobleme zum Tra-

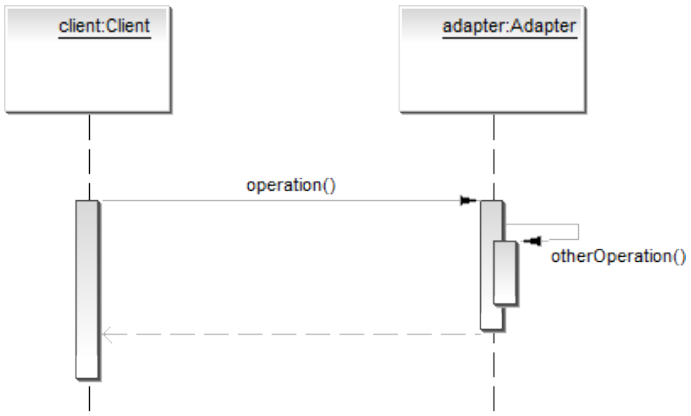



**Abbildung 4.2:** Sequenzdiagramm für einen Klassenadapter

gen. Zum einen unterstützt die für diese Arbeit verwendete Programmiersprache Java keine Mehrfachvererbung. Als Hilfslösung muss die zu implementierende Schnittstelle als Interface vorliegen, um die benötigte Vererbungsstruktur umzusetzen. Somit kann allerdings keine vorimplementierte Schnittstelle erweitert werden, was bei umfangreichen Schnittstellen den Aufwand der Umsetzung erhöht. Diese Schnittstellen sind beispielsweise bei Fujaba mit FClass, FMethod, etc. vorhanden, aber, wie erwähnt, entsprechend umfangreich und aufwendig zu implementieren. Zum Zweiten handelt es sich bei Schnittstelle und zu adaptierender Klasse um die zwei Metamodelle der Anwendungen, welche um Funktionen bzw. durch eigene Klassen erweitert werden müssten. Dies kann zu Kompatibilitätsproblemen mit bereits existierenden Projekten bzw. zu einer Einschränkung der Nutzbarkeit neu erstellter Projekte ohne den Adapter führen. Dazu gehört unter anderem, dass bei der Persi-

stenz bestimmte konkrete Metamodell-Klassen erwartet werden. Beispielsweise werden in abgespeicherten Fujaba-Modellen die konkreten Klassen (also *UMLClass* statt dem Interface *FClass*) für die Wiederherstellungen hinterlegt.

### 4.1.2 Objektadapter

Ein Objektadapter als Entwicklungsmuster nach Definition in [4] beschreibt eine Adapterklasse, welche eine Schnittstelle implementiert und auf eine zu adaptierende Klasse delegiert. Somit ist im Gegensatz zu einem Klassenadapter keine Mehrfachvererbung notwendig. Die Delegation der eingehenden Aufrufe erfolgt explizit auf die zu adaptierende Klasse. Durch die Vererbung der Schnittstelle kann direkter Einfluss auf diese genommen werden um ihre Funktionen anzupassen oder zu beschneiden.

Die Abbildung 4.3 zeigt ein Beispiel für einen Objektadapter als Klassendiagramm. Zusätz-

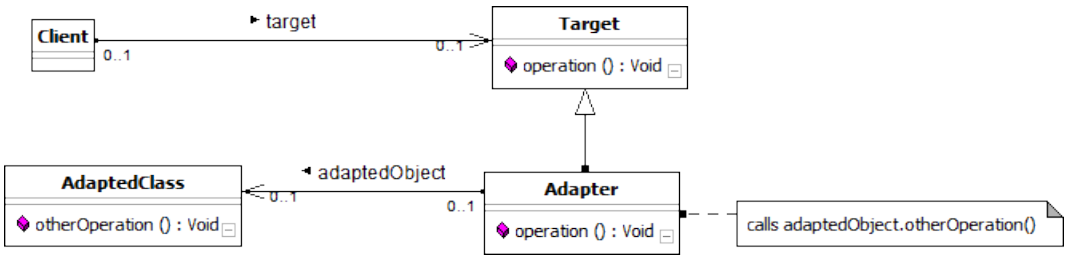



**Abbildung 4.3:** Klassendiagramm für einen Objektadapter

lich wird in Abbildung 4.4 der Ablauf eines Methodenaufrufs als Sequenzdiagramm aufgezeigt. Dabei erfolgt zuerst der Aufruf der Methode `operation()` auf „adapter“. Dieser wird über das adaptierte Objekt „adapted“ auf dessen Methode `otherOperation()` delegiert, bevor anschließend ein möglicher Rückgabewert zurückgereicht wird.

Betrachtet man diese Strategie in Bezug auf die Problemstellung, erscheint die Umsetzung nur in eine Richtung praktikabel: Die Erweiterung des Fujaba-Metamodells als Schnittstellen mit Delegation auf die entsprechenden Klassen des UML2-Metamodells. Für diese Variante können durch den Austausch von Factories (zuständig für die Erzeugung der Objekte) die benötigten, erweiterten Modellklassen einpflegt werden. Die Gegenrichtung birgt Probleme mit der Erweiterung der UML2-Metamodell-Klassen, weil es nicht vorgesehen ist diese durch eigene Implementationen zu ersetzen.

Ein großer Nachteil bei den Objektadaptern, wie schon bei den Klassenadaptern, ist, dass Kompatibilität zu bereits existierenden Projekten verloren geht. So können ältere Projekte

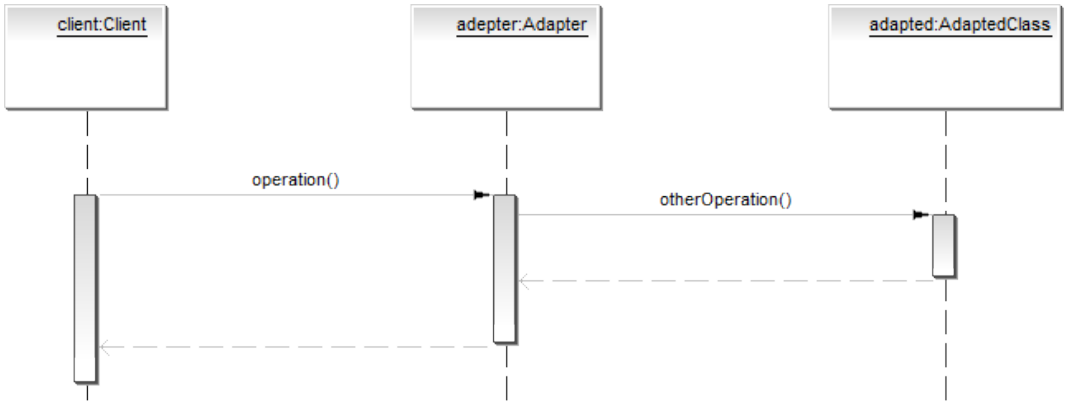



**Abbildung 4.4:** Sequenzdiagramm für einen Objektadapter

nicht ohne Aufwand verwendet, sondern müssen in die erweiterte Modellstruktur eingepasst werden. Alle erweiterten Modelle können im Gegenzug nichtmehr ohne das neue Metamodell eingeladen werden - die Flexibilität wird demnach eingeschränkt. Zusätzlich wirkt sich nachteilig aus, dass die Objektadapter nur für eine Richtung der Synchronisation eingesetzt werden können. Daraus folgen zwei unterschiedliche Techniken für die Synchronisierung der beiden Modelle.

### 4.1.3 Triple Graph Grammatiken

Die Grundlagen der Triple Graph Grammatiken(TGG) werden bereits in Abschnitt 2.3.3 erläutert. In der Aufgabenstellung dieser Arbeit wurde die Umsetzung mit TGGs wegen eines bereits existierenden und nicht zufriedenstellenden Ansatzes grundsätzlich ausgeschlossen.

## 4.2 Schwierigkeiten während der Umsetzung grundlegender Mechanismen

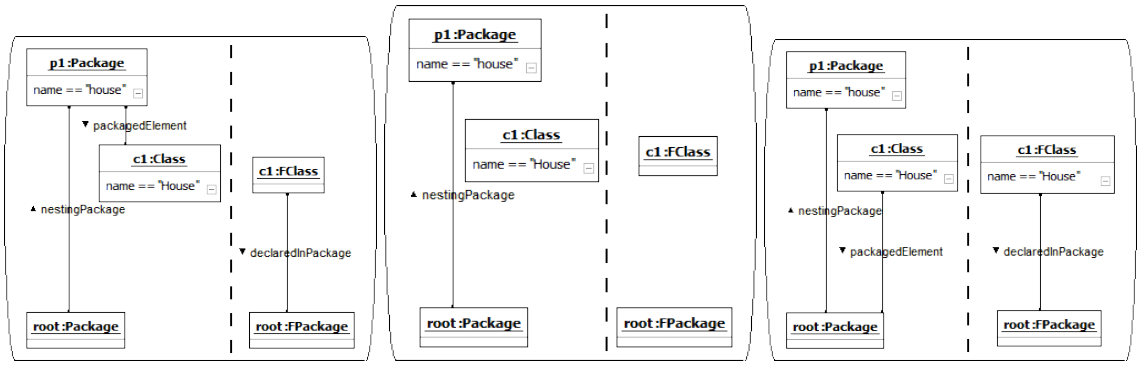
Bereits zu Beginn der Umsetzung zeichneten sich in einigen grundlegenden Mechanismen für die Synchronisierung Schwierigkeiten ab, die in den folgenden Abschnitten zu Beginn ausführlich beschrieben werden. Im Anschluss erfolgt die Analyse möglicher Lösungsansätze durch den Vergleich von deren Vor- und Nachteilen.

### 4.2.1 Seiteneffekte durch den Einsatz von Listenern

Änderungen an einem Modell erfolgen nicht immer separat, sondern sind häufig als Teil einer zusammengehörigen Transaktion zu sehen. So besteht beispielsweise die Erzeugung einer Assoziation zwischen zwei Klassen aus der Erzeugung mehrerer Objekte und dem Setzen verschiedener Attribute. Durch die Nutzung der Listener wird dieser Block aber nicht im Gesamten synchronisiert, sondern jeder Teilevent für sich. Daher muss gewährleistet werden, dass unabhängig von der Reihenfolge, in der die Events bearbeitet werden, am Ende immer derselbe Endzustand in beiden synchronisierten Modellen herrscht. Auch wenn in den Anwendungen die Reihenfolge bei der Erzeugung des Assoziation intern fest vorgegeben ist, kann sich diese in unterschiedlichen Programmversionen unterscheiden. Zudem wäre bei einer asynchronen Bearbeitung der Events kein fester Ablauf garantiert. Daher dürfen die Listener keine Seiteneffekte auslösen, die ungeplante Auswirkungen auf spätere Events oder das Ursprungsmodell haben. Obwohl nicht alle denkbaren Fälle überprüft werden können, werden durch Tests unterschiedlichste Konstellationen abgedeckt und auf den korrekten Endzustand hin überprüft. Zwei wichtige Fragen in diesem Zusammenhang sind „Wann wird ein nicht existierendes Gegenobjekt im anderen Modell erzeugt?“ und „Wie wird sichergestellt, dass Events nicht verlorengehen, wenn sie zum Zeitpunkt ihrer Erzeugung nicht bearbeitet werden können?“. Weil diese beiden Fragen auf verschiedene Aspekte dieser Arbeit Einfluss haben, werden sie innerhalb der Abschnitte 4.3 bzw. 4.4 separat behandelt.

Das bidirektionale Synchronisieren von zwei Modellen ist von Natur aus gefährdet Seiteneffekte auszulösen, welche unerwartete Auswirkungen auf das Ursprungsmodell haben. Ein besonders schwerer Fall ist die Bildung eines Zyklus, der unaufhörlich Änderungen von Modell zu Modell überträgt. Im Normalfall werden diese bereits in den Modellen selbst abgefangen. Soll beispielsweise ein Attributwert mit dem aktuell gesetzten Wert überschrieben werden, erzeugt das Modell durch die Übereinstimmung von altem und neuem Wert keinen Event und der Zyklus beendet sich; spätestens bei der ersten Wiederholung greift in diesem Fall die Unterbrechung. In komplexeren Szenarien kann dieser Mechanismus versagen. Bewirkt eine Kombination aus gesetzten Attributwerten eines oder mehrerer Objekte im Zielmodell ungewollte Änderungen im Ursprungsmodell, kann wiederum der Status des Zielmodells beeinflusst werden. Hierdurch kann eine Endlosfolge von sich wiederholenden Manipulationen in den Modellen ausgelöst werden. Auch wenn in den zur Entwicklungszeit verwendeten bzw. getesteten Modellen kein solcher Fall gefunden werden konnte, muss er in den Überlegungen berücksichtigt werden. Zusätzlich gibt es auch eine abgeschwächte Ver-

sion des beschriebenen Szenarios. Neben sich wiederholenden Zyklen, können „einfache“ Seiteneffekte vergleichbare Auswirkungen haben. Somit können schon durch die einmalige Änderungen in beiden Modellen nicht wiederherstellbare Informationen verloren gehen. Zur Erläuterung der Konsequenzen, wenn diese Problematik unbedacht bleibt, dient folgendes Beispiel.




(a) Ausgangssituation (b) Entfernen aus Paket (c) Setzen des Namens

**Abbildung 4.5:** Ablauf beim Aufruf von „setName“ auf die Implementierung von *FClass*

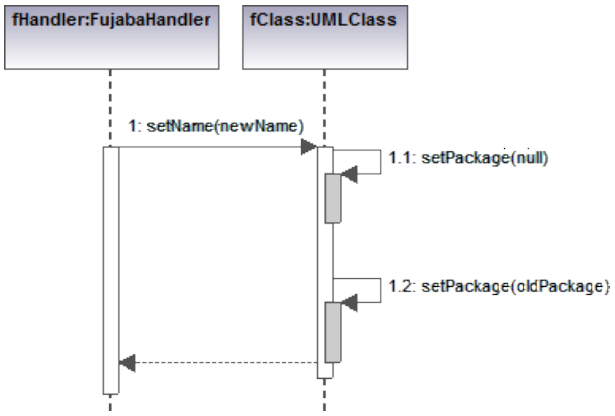



**Abbildung 4.6:** Sequenzdiagramm für das Setzen des Attributes „name“ in *UMLClass*

Wird der Name einer Klasse in Fujaba geändert, wird sie zwischenzeitlich aus ihrem aktuellen Paket entfernt. Während diesem Zustand werden einige Methoden ausgeführt, bevor sie zum Abschluss in ihr vorheriges Paket zurück verschoben wird. In Abbildung 4.6 wird dieser Vorgang noch einmal verdeutlicht. Ein Szenario, in dem diese Vorgehensweise Probleme hervorruft, betrifft die Synchronisierung nach dem Ladevorgang. Beispielsweise wird das UML2-Modell mit der Objektwelt aus Abbildung 4.5(a) (linke Seite) geladen. Hierfür existiert bisher kein entsprechendes Fujaba-Modell. Dieses wird daher angelegt, bevor die die beiden Modelle miteinander abgeglichen werden. Der Vorgang beginnt mit der Klasse c1. Diese wird zuerst in Fujaba angelegt, bevor die Attribute nacheinander synchronisiert werden. In Abbildung 4.5(a) ist diese Ausgangssituation vollständig zu sehen. Die Assoziation zwischen c1 und root auf Seiten des Fujaba-Modells liegt darin begründet, dass jedes neu angelegte

Objekt vom Typ *FClass* bei der Erzeugung automatisch dem Wurzelpaket, also root zugewiesen wird. Im nächsten Schritt wird der Name der Klasse c1 abgeglichen. Dazu ruft der *FujabaHandler* die entsprechende Methode auf (siehe Abbildung 4.6, Aufruf 1). Dies löst die Entfernung aus dessen alten Paket, d.h. ein Überschreiben des aktuellen Wertes mit `null`, aus (Aufruf 1.1). Weil durch diese Änderung im Fujaba-Modell ein Event erzeugt wird, führt dies zu einer Synchronisierung und resultiert in dem Diagramm in Abbildung 4.5(b), d.h. c1 wird im UML2-Modell ebenfalls aus seinem Paket entfernt. Schließlich wird der neue Name gesetzt und es erfolgt die Rückverschiebung in das vorherige Paket root (Aufruf 1.2). Auch diese Änderung wird synchronisiert und bewirkt im UML2-Modell die Verschiebung von c1 in das Paket root. Der resultierende Endzustand wird in Abbildung 4.5(c) dargestellt.Vergleicht man die Abbildungen 4.5(a) und 4.5(c), wird deutlich, dass die Information über das ursprüngliche Paket p1 der Klasse c1 im Laufe der Synchronisierung verloren geht.

Eine einzelne Sonderbehandlung für solche Fälle ist nicht praktikabel, weil zum Einen jede Sonderbehandlung selbst auf eigene Seiteneffekte überprüft werden muss und es zum Zweiten sehr unwahrscheinlich ist, dass alle gefährlichen Szenarien manuell erkannt und verhindert werden können. Daher ist es naheliegend die Einflussnahme auf das Ursprungsmodell während der Abarbeitung eines Events im Zielmodell für Manipulationen zu sperren. Das gesamte Modell währenddessen für die Bearbeitung zu blockieren ist unverhältnismäßig und erzeugt neue Probleme. Falls beispielsweise ein neues Objekt erzeugt werden muss, ist dies durch die Blockierung nicht möglich. Naheliegender ist die Sperrung auf Teile des Modells zu beschränken, insbesondere diejenigen Objekte, von denen das Gegenobjekt zurzeit bearbeitet wird. Daher werden diese direkt vor der endgültigen Bearbeitung eines Events intern als „nicht änderbar“ markiert, was unmittelbar nach dessen Beendigung wieder aufgehoben wird. Dadurch wird eine minimale Dauer der Sperrung garantiert. Auf das obige Beispiel bezogen, werden während dem Setzen des neuen Namens die zwischenzeitlichen Änderungen des Paketes in Bezug auf das Objekt c1 nicht auf das Modell in UML2 übertragen, so dass das ursprüngliche Paket nicht verloren geht und im Anschluss synchronisiert werden kann.

### 4.2.2 Erkennen zueinander gehörender Objekte aus zwei Modellen

Um zwei Modelle zueinander konsistent zu halten ist es unumgänglich für eine stets eindeutige Zuordnung von jeweils einem Objekt pro Modell zu garantieren. Eine verlorene oder

falsche Zuordnung kann mindestens in einem Modell unerwünschte Veränderungen hervorrufen bzw. Daten löschen. Wird beispielsweise ein Objekt durch eine falsche Zuordnung bei zwei voneinander unabhängigen Attributsynchronisierungen zwei verschiedenen Objekten im anderen Modell zugewiesen, werden diese Änderungen ungewollt auf diese Objekte aufgeteilt. Bleibt dies unbemerkt besteht die Gefahr, dass spätestens beim nächsten Einladen die Veränderungen auch auf das Ursprungsmodell übertragen werden. Auf zwei Wegen kann man dem entgegen wirken: Zum einen das jeweils neue Identifizieren der richtigen Zuordnung bei Bedarf und zum anderen das Cachen jedes existierenden Mappings zwischen den Modellen.

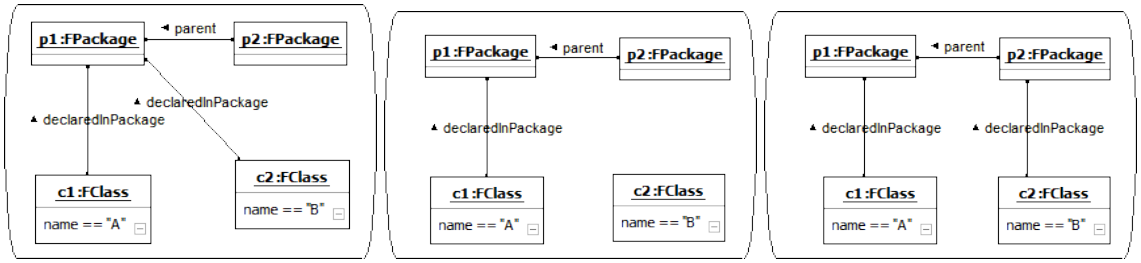
#### Identifizierung zueinander gehörender Objekte bei Bedarf

Bei dieser Variante muss für jede Klasse festgelegt werden, welche Attribute für eine eindeutige Identifizierung eines Objektes ausreichend sind. Werden daraufhin zwei Objekte miteinander verglichen, ob sie zueinander passen, wird mit Hilfe dieser Kriterien entschieden, ob diese als Mapping in Frage kommen. Wird ein Attributkriterium selbst im Wert geändert, zeigt sich der Nachteil in diesem Ansatz. Weil die Identifizierung vor der Synchronisierung stattfinden muss, besitzt das Objekt im Zielmodell noch den alten Attributwert. Daher muss das Partnerobjekt mit Hilfe dieses alten Wertes gesucht werden, bevor der neue Wert gesetzt werden kann. Dementsprechend können nicht in jedem Fall die aktuellen Werte des Objektausgelesen und zu Rate gezogen werden. Stattdessen müssen verschiedene Identifizierungsmethoden geboten werden. Besitzt eine Klasse beispielsweise drei Identifizierungsattribute, sollten zumindest vier Identifizierungsmethoden erzeugt werden; die erste, wenn die aktuellen Werte aus dem Objekt extrahiert werden können, die übrigen drei wenn eines der Attribute mit einem geänderten Wert gesucht werden muss. Damit diese unterschiedlichen Szenarien überhaupt abgedeckt werden können, müssen bei einer Änderung die ursprünglichen, also vorher gesetzten Werte, bekannt sein. Durch die Nutzung der Listener wird dies garantiert, weil jeder Event den neuen und alten Attributwert mitsendet.

Betrachtet man diese Vorgehensweise in Bezug auf die Metamodelle von Fujaba und UML2 zeigen sich bei primitiven Attributwerten keine Probleme. Derartige Änderungen werden stets über Events an alle Listener weitergeleitet. Bei nicht primitiven Werten, wie beispielsweise einer Assoziation zwischen zwei Klassen eines Modells, muss differenziert werden. Bei Fujaba zeigen sich auch hier keine Schwierigkeiten, weil beide durch die Assoziation verbundenen Klassen bei einer Wertänderung einen Event erzeugen. Im UML2-Metamodell sind die Assoziationen häufig nicht bidirektional ausgelegt. So kann es bei einer Eltern-Kind

Beziehung vorkommen, dass ein Elternelement über mehrere Assoziation mit deinem Kind-Objekt verbunden ist. Das Kind-Objekt besitzt allerdings nur eine rückwärtige Assoziation. Weil das Setzen der Assoziation von Kind- zu Eltern-Objekt nicht automatisch das Setzen aller vorhandenen rückwärtigen Assoziationen bedeutet, werden diese unidirektional angelegt. In diesem Fall wird also nicht auf beiden Seiten ein Event ausgelöst, sondern beispielsweise nur im Eltern-Objekt für jede neue Assoziation zum Kind-Objekt. Verpasst wird auch hierdurch keine Änderung, weil man die Listener auf die Problematik hin ausrichten kann, in welchem Fall, welche Events erzeugt werden. Muss aber das Gegenobjekt neu gefunden werden, zeigt sich die Problematik, welche im Folgenden an einem Beispiel erläutert wird.

Das Objektdiagramm in Abbildung 4.7(a) dient als Ausgangssituation der Veranschauli-




(a) Ausgangssituation (b) Nach Auswertung von Event E(p1:c2:null) (c) Gewünschtes Ergebnis nach E(p2:null:c2)

**Abbildung 4.7:** Beispiel für das Problem Objekte eindeutig nur bei Bedarf zu identifizieren

chung. In UML2 wird die Klasse c2 aus dem Paket p1 in das Paket p2 verschoben. Dies soll durch die Synchronisierung in Fujaba nachvollzogen werden. Die Änderungen in UML2 erzeugen die beiden Events **E(p1:c2:null)**[1] und **E(p2:null:c2)**. Erfolgt die Auswertung des Events **E(p1:c2:null)** zuerst, entsteht das Diagramm in Abbildung 4.7(b) - c2 wird in p1 durch `null` ersetzt, also entfernt. Im Anschluss soll **E(p2:null:c2)** verarbeitet werden und als Resultat das Diagramm in Abbildung 4.7(c) entstehen. Durch das vorherige Event ist aber die eindeutige Identifizierbarkeit verloren gegangen; in diesem Beispiel bestehend aus dem Elternelement (das Paket) und dem Attribut „name“. Durch das fehlende Paket von c2 kann es nichtmehr eindeutig als Gegenelement zu seinem UML2 Äquivalent identifiziert werden, weil elternlose Elemente nur einen notwendigen Zwischenzustand darstellen und solche Klassen mit dem gleichen Namen mehrfach vorkommen können. Um dem entgegen zu wirken, ist es nötig solche Events nur in Kombination zu verarbeiten. Nur auf diese Weise kann die Verschiebung von Kind-Objekten eindeutig nachvollzogen werden. Entschei-

[1] Die Schreibweise steht für Event(Quellobjekt des Events:alter Attributswert:neuer Attributswert). Das Attribut selbst ist irrelevant oder ergibt sich aus dem genutzten Kontext

det man sich für diesen Weg, müssen alle relevanten Events zwischengespeichert werden, bis sie bearbeitet werden können. Problematisch wird das Finden des Zeitpunkts, ab wann eine Kombination als vollständig erachtet werden darf. Wird ein Objekt aus seinem Elternelement gelöscht, aber zu keinem anderen neu hinzugefügt, wird ein „DELETE“-, aber kein „ADD“-Event erzeugt. Im Programmablauf ist es nur schwer bzw. mit viel Aufwand entscheidbar, ob noch ein weiteres relevantes Event kommt. Deswegen könnten zum einen manche Events unbearbeitet bleiben und die Modelle in unterschiedlichen Zustanden vorliegen. Zum anderen verbleiben diese Events auf Dauer in diesem unentscheidbaren Status und somit in einer stetig wachsenden Liste zwischengespeichert. Als zusätzlichen Nachteil gibt es Konstellationen, in denen ohne weitere Hilfsmittel eine eindeutige Identifizierung nicht möglich ist. Betrachtet man beispielsweise eine Klasse wie *FMethod*, die in ihren Identifikationsattributen auch die Liste von Objekten vom Typ *FParam* besitzt. Wird eines dieser Objekte aus der Liste gelöscht, weiß man zwar dass es vorher in der Liste vorhanden, aber nicht an welcher Stelle es genau platziert war. Gibt es mehrere ähnliche Methoden, die sich nur anhand der Positionierung der Parameter unterscheiden, kann man mehrere möglicherweise passende Gegenobjekte finden. Insgesamt sprechen also viele Punkte gegen diesen Ansatz.

Resultierend kann man sagen, dass dieser Ansatz speicherschonend ist, weil keine zusätzlichen Informationen für das Mapping vorgehalten werden müssen. Allerdings geht dieser Vorteil durch die sich häufig wiederholende Suche nach einem passenden Objekt im Zielmodell zu Lasten der Laufzeit.

#### Cachen existierender Mappings zueinander gehörender Objekte

Dieser Ansatz mittels Caching setzt auf die Speicherung aller bekannten zusammengehörigen Objekte in beiden Modellen. Im Gegensatz zu dem vorherigen Abschnitt bedeutet dies einen erhöhten Speicherbedarf. Dieser ist aber im Vergleich zu den gewonnen Vorteilen vernachlässigbar. Für die grundlegende Analyse dieser Variante sind genauere technische Details nicht nötig; diese werden in Abschnitt 4.3 genauer ausgeführt.

Geht man davon aus, dass von zwei zusammengehörigen Objekten ein gespeichertes Mapping existiert, werden die Nachteile aus der oberen Alternative ausgeräumt. Zum Ersten ist es nichtmehr nötig, zu jeder Gelegenheit das Mapping explizit auf den Identifikationskriterien basierend auszumachen, weil es zentral vorgehalten wird. Dementsprechend ist es auch unproblematisch wenn sich ein Identifikationsattribut ändert, weil es nicht zur Suche gebraucht wird, sondern direkt an das Gegenobjekt übertragen werden kann. Auch eine

logisch zusammenhängende Aktion aus mehreren Events kann sofort Event für Event verarbeitet werden.

Zusammenfassend gesagt werden alle Schwierigkeiten, die auf Grund von Problemen mit der Identifikation des Gegenelements auftreten, ausgeräumt. Zusätzlich ist dieser Ansatz schneller, weil nur beim Laden existierender Projekte zusammengehörige Objekte einmalig gesucht werden müssen. Anschließend ist es jeweils nur noch ein Zugriff (in konstanter Zeit) auf eine Map.

## 4.3 Mapping zueinander gehörender Objekte der Metamodelle

In Abschnitt 4.2.2 wurde als grundlegende Idee bereits erläutert, wie durch Mapping-Objekte zusammengehörige Objekte aus den beiden zu synchronisierenden Modellen zur Laufzeit gespeichert werden. Diese werden in einer bidirektionalen Map gesichert, d.h. auf die Einträge kann sowohl über das Fujaba- als auch über das UML2-Objekt zugegriffen werden. Die technischen Details dieser Map sind im Abschnitt 4.3.3 zu finden.

Mappings von zwei Objekten werden in zwei Szenarien benötigt. Das erste betrifft die Änderung eines Attributes von einem der Objekte und die Übertragung des neuen Wertes auf das andere Objekt. In diesem Fall muss wird durch das Mapping das Zielobjekt bestimmt. Das zweite Szenario tritt ein, wenn ein solches Objekt selbst ein Verweis ist und sein Gegenstück vergleichbar als Verweis gesetzt werden muss. Im ersten Fall handelt es sich immer um die Instanz einer Klasse aus einem der beiden zu synchronisierenden Metamodelle. Im zweiten Fall, welcher deutlich seltener auftritt, kann es sich zusätzlich auch um einen primitiven Typen wie String, Integer oder Boolean handeln, beispielsweise kann der Rückgabetyp einer Methode ein primitives oder komplexes Objekt sein. Unter anderem wegen diesem Missverhältnis in der Häufigkeit der beiden Szenarien werden Objekte in zwei Gruppen eingeteilt: primitive und Metamodell-Objekte.

### 4.3.1 Mapping von Metamodell-Objekten

Mappings von Objekten dieses Typs werden pro Modellpaar gesichert, d.h. für jeweils zwei miteinander verbundene Projekte existiert eine Map, die alle bestehenden Mapping-Einträge

enthält. Für ein solches Mapping gibt es drei relevante Fragen: Wie wird es erzeugt? Wie kann man auf es zugreifen? Wie erfolgt eine Löschung?

Die ersten beiden Fragen können kombiniert beantwortet werden. Erzeugung und Zugriff eines Mappings erfolgen bei Bedarf. Zuständig hierfür ist die Klasse *Mapping*, welche über verschiedene statische Methoden alle Anfragen bezüglich des Mapping behandelt und diese an die zuständigen Bereiche delegiert. Für die Abfrage des Mapping-Partners kommen zwei Methoden zum Einsatz: `getOrCreate(NamedElement element)` dient der Identifizierung eines passenden Objektes vom Typ *FElement*; um das entsprechende Objekt vom Typ *NamedElement* zu finden ist die Methode `getOrCreate(FElement element)` zu nutzen. Dort wird zuerst in allen bereits existierenden Mappings nachgeschaut. Ist diese Suche erfolglos, wird im nächsten Schritt das andere Modell im Gesamten für ein passendes Objekt durchforstet. Dazu sind für jede Klasse ausreichende Identifikationskriterien (beispielsweise für eine Klasse ihr Name und das Paket in dem sie sich befindet) festgelegt, mit denen ein passendes Objekt verifiziert werden kann. Führt auch dies nicht zum Erfolg, ist im anderen Modell kein passendes Objekt vorhanden und es wird versucht dort eines zu erstellen. Dazu müssen im Ursprungsobjekt zwei Kriterien erfüllt sein. Zum Ersten muss das Objekt einen Namen besitzen und zum Zweiten muss es einem Elternobjekt zugewiesen worden sein. Damit wird sichergestellt, dass das neu erzeugte Objekt eine Referenz in seinem Modell aufweist und deswegen nicht unmittelbar nach seiner Erstellung wieder durch den *GargabeCollector* aufgesammelt werden kann. Dies ist nur für das UML2-Metamodell von Belang, weil jedes Objekt aus dem Fujaba-Metamodell bei der Erstellung bereits eine Referenz auf sein Projekt gesetzt bekommt. Unmittelbar nach der Erzeugung wird für beide Objekte ein Mapping erstellt und für spätere Abfragen gespeichert.

Gelöscht werden Mappings, wenn sie nichtmehr benötigt werden. Dies ist immer dann der Fall, wenn eines bzw. beide Objekte aus ihrem Modell entfernt werden. Die Löschung erfolgt automatisch in der *BidiWeakMap* und wird im entsprechenden Abschnitt 4.3.3 genauer erläutert.

### 4.3.2 Mapping von primitiven Objekten

Diese Objektgruppierung wird aus mehreren Gründen separat behandelt. Neben dem weiter oben erwähnten eher selteneren Bedarf eines solchen Mappings, befinden sich die primitiven Typen bei UML2 in eigenen Bibliotheksprojekten, d.h. abseits der vom Benutzer

bearbeiteten Projekte. Bearbeitet man also beispielsweise mehrere Projekte parallel, besitzt, im Gegensatz zu Fujaba, nicht jedes geladene Modell seine eigenen Objekte für primitive Typen, sondern es werden in allen dieselben verwendet. Daher ist es ein unnötiger Speicherverbrauch für jedes Projektpaar explizit Mappings für primitive Typen zu speichern. Dadurch dass bei Fujaba jedes Projekt eigene Objekte für primitive Typen hat, ist es aber genauso wenig möglich diese Zuordnungen nur einmalig für alle Projekte anzulegen. Als Resultat bietet sich eine Alternative an, die vollständig auf gespeicherte Mappings verzichtet. Somit unterscheidet sich die Vorgehensweise beim Suchen und Verifizieren des richtigen Objekts grundlegend zum Ansatz bei Metamodell-Objekten.

Für die Suche nach dem passenden Gegenstück bieten `get(FBaseType type)` bzw. `get(PrimitiveType type)` den Einstiegspunkt. Um die Objekte des Modells nicht mehrfach durchlaufen zu müssen, werden diese in drei Kategorien eingeteilt. In die dritte Kategorie fallen alle Objekte, die auf keinen Fall zum Übergebenen passen können. In die zweite Kategorie fallen alle, die von dem Bezeichner her passen, sich aber in der Groß-/Kleinschreibung unterscheiden, beispielsweise „boolean“ und „Boolean“. In die erste Kategorie gehören ausschließlich exakte Treffer, welche vorrangig zurückgeliefert werden. Auf diesem Weg werden beinahe alle primitiven Typen abgedeckt. Es gibt allerdings drei Ausnahmen dieser Regel, die eine Anpassung der Vorgehensweise nach sich ziehen:

1. Arrays: Objektsammlungen werden normalerweise über Kardinalitäten wie 0..n modelliert. Das UML2-Metamodell beschränkt sich auch auf diese Vorgehensweise. Fujaba bietet zusätzlich einige feste Basistypen für Arrays, beispielsweise StringArray oder IntegerArray, an.

2. void: In UML2 bedeutet `void`, dass kein Typ gesetzt ist, es also den Wert `null` besitzt; in Fujaba gibt es den eigenen Basistyp *void*

3. Konstruktor: Der Basistyp *constructor* wird in Fujaba genutzt, um solche Methoden zu kennzeichnen. Die UML2-Spezifikation schreibt vor, dass ein Konstruktor als Rückgabewert das Objekt gesetzt haben muss, in dem er als Methode geführt wird, also seine Elternklasse. Zusätzlich wird das Keyword *create* gesetzt.

Betrachtet man die Spezialfälle kristallisieren sich zwei unterschiedliche Herangehensweisen heraus. Geht die Änderung von einem Objekt des UML2-Metamodells aus, kann man aus diesem Objekt alle Informationen extrahieren, um das passende Gegenstück in Fujaba zu finden, d.h. man muss nur einen passenden Datentyp finden, der anschließend auf das geänderte Attribut übertragen wird. Erfolgt die Veränderung in einem Objekt in Fu-

jaba, ist der gleiche Ansatz anwendbar, außer es handelt sich um ein Array oder einen Konstruktor. In beiden Fällen ist eine zusätzliche Änderung am Zielobjekt nötig (Kardinalität bzw. Keyword). Daher bietet sich der Einsatz des Strategy-Musters [4] an, um für die Spezialfälle ein gesondertes Verhalten festzulegen. Dies bedeutet, dass die Methode `get(FBaseType type)` kein Objekt vom Typ *PrimitiveType*, sondern eine *ITypeStrategy* zurück gibt.

### 4.3.3 Containerklasse der Mappings

Die *BidiWeakMap* verwaltet alle bidirektionalen Mappings der Metamodell-Objekte, d.h. jeder Eintrag besteht aus einem Objekt vom Typ *FElement* (Fujaba) und einem vom Typ *NamedElement* (UML2). In Abbildung 4.8 ist der Aufbau der *BidiWeakMap* inklusive einem Mapping zwischen den Objekten fElement und uml2Element zu sehen. Weil die Synchronisierung zwischen den Metamodellen bidirektional ist, müssen auch die Einträge über beide Komponenten zugänglich gemacht werden. Dazu besteht die Map intern aus zwei weiteren Maps, die zum einen die Beziehung *FElement* -> *NamedElement* (Objekt fEntry) und zum anderen die Beziehung *NamedElement* -> *FElement* (Objekt uEntry) abbilden. Diese werden beim Löschen oder Hinzufügen eines Eintrages parallel gepflegt, so dass sie stets konsistent zueinander sind.

Neue Einträge werden bei Bedarf über die Methode `getOrCreate` in der Klasse *Map-*

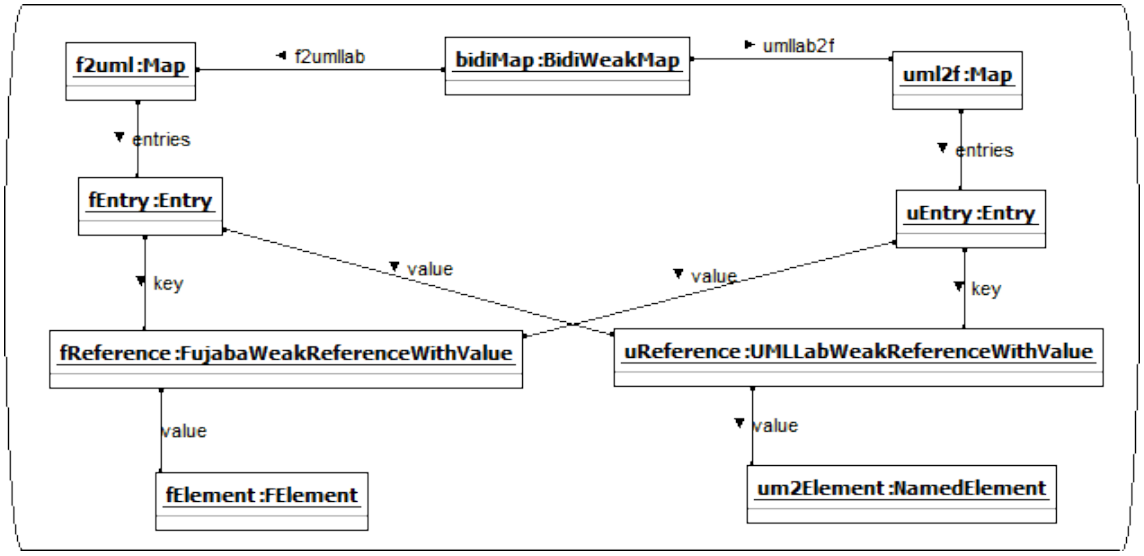



**Abbildung 4.8:** Aufbau der *BidiWeakMap* inklusive eines Mappings

*ping* hinzugefügt, d.h. zwischen dem Abrufen bereits existierender und dem erzeugen neuer Mappings muss nicht unterschieden werden. Für das Entfernen nichtmehr benötigter Objektpaare gibt es einen manuellen und einen automatisch durchgeführten Ansatz. Diese überflüssigen Einträge können nur dann entstehen, wenn ein Objekt des Paares (bzw. durch die Synchronisation im Endeffekt beide) gelöscht und somit auch das Mapping obsolet wird. Um diese manuell aus der Map zu entfernen muss es eine Möglichkeit geben, dass man über die Löschung eines der enthaltenen Objekte benachrichtigt wird. Bei Fujaba gibt es für diesen Fall ein spezielles Event (*RemoveYou*), welches erzeugt wird, nachdem alle von dem zu löschenden Objekt ausgehenden Referenzen entfernt wurden. Im Metamodell in UML2 ist nichts vergleichbares vorgesehen. Zwar gibt es für das Entfernen der einzelnen Referenzen eigene Events, aber daraus ist nicht zu schließen, ob das komplette Objekt oder nur ein einzelnes Attribut gelöscht wird. Dieser Ansatz ist also nicht umzusetzen.

Alternativ kann man die Verwaltung der Einträge auch in diesem Punkt durch die *BidiWeakMap* automatisiert durchführen lassen. Dazu muss ein Kriterium festgelegt werden, wann ein Eintrag als überflüssig angesehen wird. Naheliegend ist der Zeitpunkt, wenn eines der Objekte eines Eintrages durch den *GarbageCollector* eingesammelt wird. Dann hat das Objekt keinerlei Referenzen mehr aufzuweisen und es wird kein Mapping von diesem auf ein weiteres Objekt benötigt. Normalerweise ist die Beinhaltung in einer Map selbst eine Referenz, so dass der Gargabe Collector dort niemals aktiv werden kann. Daher gibt es die Möglichkeit Objekte in sogenannten schwachen Referenzen zu kapseln. Diese werden vom *GarbageCollector* nicht beachtet und gelöscht, wenn keine weiteren normalen Referenzen zu dem Objekt existieren. Nach ihrer Löschung werden diese Referenzen in eine Queue verschoben, die ihnen bei ihrer Erzeugung übergeben wird. Diese Queue wird ebenfalls in der *BidiWeakMap* verwaltet und regelmäßig abgefragt.

Zusätzlich zum Problem der zu löschenden Einträge kann gleichzeitig die Synchronisierung der Löschung ausgeführt werden. Dazu wird die Klasse *WeakReference* in die abstrakte Klasse *WeakReferenceWithValue* erweitert (jede Instanz dieser Klasse kapselt ein Objekt in einem Eintrag der Map, siehe Abbildung 4.8). Weiterhin implementiert sie das Interface *Runnable*. Hierdurch ist es möglich in der *BidiWeakMap* gelöschte, schwache Referenzen aus der Queue „auszuführen“: Die zwei konkreten Implementierungen *FujabaWeakReferenceWithValue* und *UML2WeakReferenceWithValue* legen das Verhalten während der Ausführung fest. In der ersten Variante wird das das zugehörige Objekt aus dem UML2-Metamodell gelöscht, in der zweite Variante der Objektpartner in Fujaba.

## 4.4 Vorverarbeitung eingehender Events

Die Änderungen in den Objekten der Modelle sind ausschließlich über die generierten Events zugänglich. Daher ist der Einsatz von Listenern für deren Analyse unumgänglich. Durch die große Vielfalt an Eventarten und -quellen ist eine Art Vorverarbeitung auf mehrere Arten hilfreich. Zum einen werden auf diesem Weg möglichst viele Events ausgefiltert, weil sie für die Synchronisierung nicht geeignet sind. Zum anderen muss die endgültige Auswertung der Events zwar in spezialisierten Handlern durchgeführt werden, grundlegende Vorverarbeitungsschritte können aber zentral für alle einheitlich vorgenommen werden. Auf den genauen Ablauf und die Unterschiede in Bezug auf das Fujaba- und das UML2-Metamodell wird im Folgenden eingegangen.

Für die Platzierung der Vorverarbeitungsschritte gibt es verschiedene Möglichkeiten. Weil alle Events aus demselben Metamodell während diesem Schritt identisch behandelt werden können, sollten sie an einer zentralen Stelle liegen und nicht in die spezialisierten Handler hinein kopiert werden. Eine einfache Lösung ist eine gemeinsame Superklasse von der alle Handler erben. Dort können die ersten Bearbeitungen vorgenommen werden, bevor die speziellen Verarbeitungsschritte individuell überschrieben werden. Ein Vorteil ist, dass man jeden Handler als Listener implementiert und nur an relevanten Objekten anmelden muss. Somit werden sehr selten Events unnötigerweise geschickt. Diese Tatsache ist gleichzeitig auch als Nachteil zu sehen, weil somit von vornherein festgelegt werden muss welche Events für welchen Handler relevant sind und dynamische Kriterien erschweren. Zudem müssen manche Events durch verschiedene Handler ausgewertet werden, so dass sie überflüssigerweise mehrfach die Vorverarbeitung durchlaufen. Als Alternative bietet sich ein einheitlicher Listener als zentrale erste Anlaufstelle für alle eingehenden Events an. Sowohl für das Fujaba- als auch für das UML2-Metamodell gibt es daher jeweils ein Objekt, welches sich an allen relevanten Stellen anmeldet, um Änderungsevents entgegen zu nehmen.

Bei jedem eintreffenden Event wird zuerst geprüft, ob das zugehörige Projekt zurzeit mit ei-

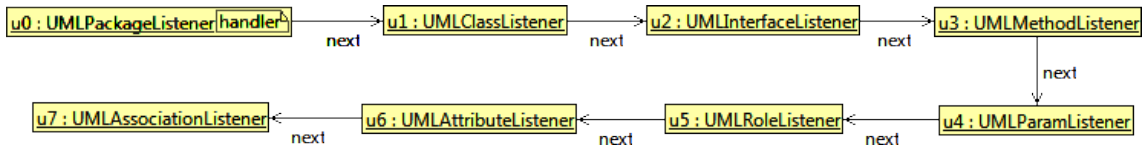



**Abbildung 4.9:** Aufbau der Chain-of-Responsibility der Fujaba-Handler

nem anderen Projekt verbunden, d.h. ob eine Synchronisation überhaupt möglich ist. Sollte dies nicht der Fall sein, wird an dieser Stelle bereits abgebrochen. Die weiteren Überprüfungen hängen von dem eingesetzten Metamodell ab und werden daher in den Abschnitten

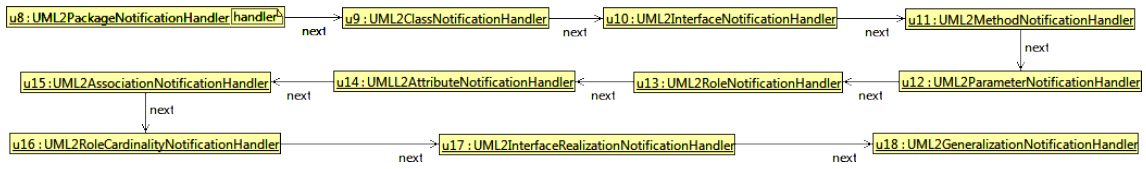



**Abbildung 4.10:** Aufbau der Chain-of-Responsibility der UML2-Handler

4.4.1 bzw. 4.4.2 besprochen. Im Anschluss erfolgt die Weitergabe des Events an spezielle Handler. Hierfür ist eine Variante des „Chain-of-Responsibility“-Musters [4] zur Verwaltung der Handler implementiert. In den Abbildungen 4.9 und 4.10 ist die Objektstruktur für die Fujaba- und UML2-Handler dargestellt. Jedes Element dieser Kette hat einen festen Vorgänger und Nachfolger (verbunden durch die „next“-Assoziation) und enthält logische Operationen um auszuwerten, ob ein ihm übergebenes Objekt bearbeitet werden kann oder nicht. In der ursprünglichen Version dieses Musters wird das zu untersuchende Objekt an das vorderste Kettenelement übergeben und solange an den Nachfolger weitergeleitet, bis es verarbeitet werden kann. An dieser Stelle wird die Kette unterbrochen und das Element führt automatisch seine Verarbeitungslogik aus. Auf Grund spezieller Anforderungen wird diese Vorgehensweise in zwei Punkten angepasst. Zum einen wird die Kette immer bis zum Ende durchlaufen, weil manche Events von mehreren Handlern verarbeitet werden müssen. Die feste Reihenfolge bekommt zusätzliche Relevanz, weil manche Handler zwingend in einer bestimmten Reihenfolge aufgerufen werden müssen. Zum Zweiten wird die Auswertung für jeden passenden Handler nicht automatisch ausgeführt. Weil noch verschiedene Zwischenschritte nach der Identifikation eines passenden Handlers nötig sind, erfolgt sie zu einem späteren Zeitpunkt.

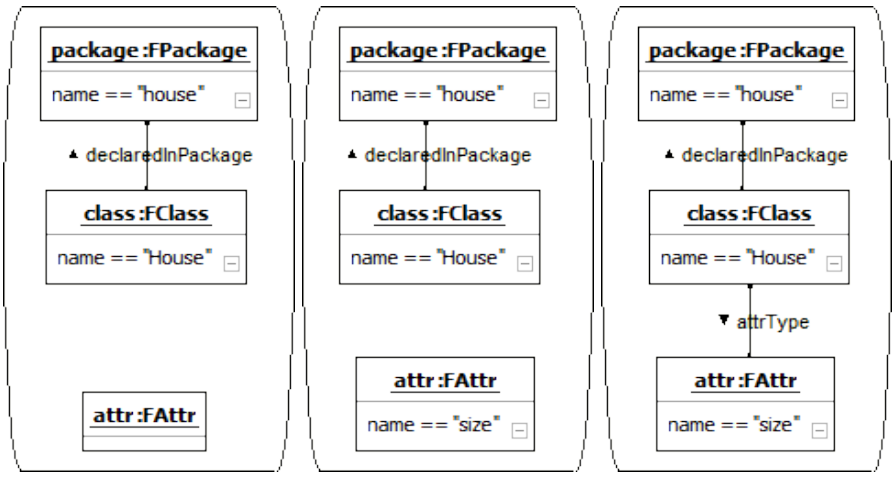
Neben dieser allgemeinen Vorgehensweise unterscheidet sich der genaue Ablauf zwischen Fujaba und UML2 stark, so dass dies in den folgenden Abschnitten separat betrachtet wird.

### 4.4.1 Spezielle Verarbeitungsschritte für Events aus dem Fujaba-Metamodell

Damit alle Events eines Objektes empfangen werden können, muss ein *PropertyChangeListener* direkt nach dessen Erzeugung angemeldet werden. Weil man in Fujaba jedes Objekt über registrierte Factories erzeugt, wird der Umweg über einen zusätzlichen Listener genommen. Dieser wird an allen Factories angemeldet und somit über jedes erstellte Objekt

informiert. An diesen wird anschließend der zuständige Listener vom Typ *FujabaEventListener* angemeldet.

Um Attributänderungen zwischen Modellen zu synchronisieren müssen die zueinander ge-




(a) Ausgangssituation (b) Setzen des Attributs „name“ (c) Attribut in Klasse einfügen

**Abbildung 4.11:** Beispielablauf für die Erzeugung eines Gegenelements zu dem Objekt attr

hörenden Objekte gefunden werden. Weil diese Zuordnung eindeutig sein muss, ist ein Mindestmaß an Informationen in den Objekten für eine Identifizierung nötig. In Fujaba kommen die ersten Events aber bereits ab der Erstellung, also bevor diese Grenze erreicht ist, an. Ein erläuterndes Beispiel ist in Abbildung 4.11 zu sehen. Abbildung 4.11(a) zeigt die Ausgangssituation. Für package und class gibt es in UML2 bereits passende Gegenstücke; attr ist hingegen noch ohne Gegenstück, weil dieses ohne existierendes Elternobjekt nicht erstellt wird. In Abbildung 4.11(b) ist der Name von attr geändert. Das zugehörige Event kann aber noch nicht verarbeitet werden, denn erst nachdem attr in Abbildung 4.11(c) zu dem Objekt class hinzugefügt wird, kann ein passendes, eindeutiges Objekt in UML2 erzeugt werden. Alle bis zu diesen Zeitpunkt erhaltenen Events zu dem Objekt attr müssen nachträglich bearbeitet werden, um keine Änderung zu verpassen. Hierzu werden die Events zwischengespeichert, falls sie nicht sofort verarbeitet werden können. Erfüllt zwischenzeitlich das Zielobjekt eines Events die Erstellungskriterien und sein Gegenobjekt wird erzeugt, erfolgt anschließend die Durchsuchung aller Zwischenspeicherungen und deren Neuauswertung, falls sie zum Ursprungsobjekt passen. Schließlich müssen alle Kindobjekte rekursiv durchlaufen und mit den zwischengespeicherten Events abgeglichen werden. Beispielsweise konnten alle Referenzen von diesen auf das Elternobjekt bisher nicht synchronisiert werden. Die zentrale Sammlung aller noch nicht auswertbaren Events stellt einen weiteren Vorteil in

der Nutzung eines einzelnen *PropertyChangeListener*s für alle Modell-Objekte dar.

### 4.4.2 Spezielle Verarbeitungsschritte für Events aus dem UML2-Metamodell

Im UML2-Metamodell gibt es keine Möglichkeit sich in die Objekterzeugung einzuhängen, um über neu erstellte Objekte benachrichtigt zu werden. Somit ist man darauf angewiesen über Events von bereits bekannten Objekten über neu hinzugekommene informiert zu werden. Dazu bietet EMF die Klasse *EContentAdapter* an, welche eingehende Events/Notifications untersucht, um sich an neuen bzw. nicht mehr existierenden Objekten als Listener an- oder abzumelden. Durch *UML2NotificationListener* wird die Klasse um die eigentliche Logik zur Vorverarbeitung von Events erweitert.

Nach der Überprüfung auf das zugeordnete Metamodell, werden alle Events ausgefiltert, die wegen ihres Typs bereits aussortiert werden können. Im UML2-Metamodell wird zwischen verschiedenen Typen unterschieden, beispielsweise SET, ADD, ADD_MANY, REMOVE, REMOVE_MANY, RESOLVE, REMOVING_ADAPTER, etc.. REMOVING_ADAPTER und RESOLVE enthalten keine für die Synchronisation relevanten Informationen, weswegen sie nicht weiter beachtet werden müssen.

Durch die automatische Verwaltung der An- und Abmeldung des Listeners ist dieser, im Gegensatz zu Fujaba, auch an Objekten angemeldet, die keine zu synchronisierenden Events erzeugen. Die Anzahl der zu überprüfenden Events ist daher deutlich größer und die Vorselektierung umso wichtiger. Aus diesem Grund wird in der Implementierung des Listeners sehr früh eine Überprüfung bezüglich der Klasse des eventerzeugenden Objekts durchgeführt. Verschiedene Klassen oder Interfaces dürfen niemals bzw. müssen immer in deren Vererbungshierarchie geführt werden. Beispielsweise sind Änderungen an Objekten, die nicht auf der Basisklasse *Element* basieren, nicht relevant. Fällt diese Überprüfung negativ aus, werden auch sie aussortiert, so dass viele unnötige Arbeitsschritte eingespart werden können.

Ist es nicht möglich ein Event auszuwerten, weil für das zu synchronisierende Objekt noch kein Gegenstück im anderen Modell erzeugt werden kann, ist eine Zwischenspeicherung nicht nötig. Dieser Mechanismus hat bei Fujaba den Grund, dass man garantiert alle Events erhält und dadurch im Nachhinein ausschließlich die bis dahin geänderten Attribute synchronisieren kann. In UML2 erfolgt die Anmeldung des Listeners erst bei einem vorgefertigten Objekt, d.h. mindestens das Elternelement, meistens aber sind bereits mehrere Attri-

bute gesetzt. Weil die veränderten Attribute somit nicht bekannt sind, muss auf jeden Fall eine vollständige Synchronisation des Objektes, wie in Abschnitt 4.5 beschrieben wird, erfolgen. Ein zweigleisiger Ansatz mit zusätzlicher Auswertung von bisher nicht verarbeiteten Events ist daher unnötig.

## 4.5 Synchronisierung vollständiger Objekte

Manchmal ist es notwendig Objekte vollständig, also inklusive aller Attribute, zu synchronisieren. Beispielsweise werden nach dem Laden von Modellen diese durch die Synchronisierung aller enthaltenen Objekte miteinander abgeglichen. Für das UML2-Metamodell ist dies auch bei der Erzeugung von neuen Objekten nötig, weil diese erst in die Eventverarbeitung integriert werden können, wenn sie bereits gesetzte Attribute besitzen. Die initiale Synchronisierung ist daher unumgänglich.

Der vollständige Abgleich eines Objektes erfolgt immer in eine Richtung, d.h. stammt das Synchronisierungsobjekt aus dem Fujaba-Metamodell, werden alle Attribute von diesem auf sein Gegenstück im UML2-Metamodell übertragen. Weil ungesetzte Werte nicht miteinbezogen werden, verbleiben in diesen Attributen im Zielobjekt möglicherweise unterschiedliche Werte. Eine einseitige Synchronisierung garantiert also nicht, dass anschließend beide Objekte in allen Attributen identisch sind. Stattdessen muss der Vorgang für beide zueinander gehörenden Objekte durchgeführt werden. Ausschließlich nach dem Laden von zwei Modellen ist der Abgleich aller Objekte in jedem Attribut nötig, so dass nur in diesem Fall eine automatische, beidseitige Synchronisation hilfreich ist. Eine Sonderbehandlung ist in diesem Fall aber unnötig, weil dieses Vorgehen keinen nennenswerten Vorteil gegenüber der Einzelsynchronisation der beiden Objekte bietet. Es ist beispielsweise nicht möglich die Synchronisierung der zueinander passenden Attribute in den Objekten zusammenzufassen. Der Grund liegt in der Implementierung für die Übertragung eines Attributwertes in das andere Modell. Dies wird in Abschnitt 4.6 vorgestellt und ist vor allem darauf ausgelegt die Kopplung der Modelle so gering wie möglich zu halten. Daher ist eine exakte Zuordnung von Attributen bzw. Attributsamen nicht möglich, was solche Einsparungen ausschließt. Als Beispiel sollen zwei Objekte vom Typ *FClass* bzw. *Class* miteinander abgeglichen werden. Wird das Attribut „abstract“ aus *FClass* synchronisiert, kann anschließend die Synchronisierung des Attributes „isAbstract“ aus *Class* nicht eingespart werden, weil die Zuordnung von „abstract“ auf „isAbstract“ als gleiches Attribut in den verschiedenen Metamodellen nicht bekannt ist. Solche Beziehungen zusätzlich zu speichern, verringert die Flexibilität

und verursacht zusätzliche Arbeit bei Anpassungen in der Synchronisierungslogik.

Ein wichtiger Punkt bei der Synchronisierung eines Objektes betrifft die Frage, ob nur das Objekt selbst oder auch alle assoziierten Objekte betroffen sein sollen. Zusätzlich muss festgelegt werden, wie weit sich die Synchronisierung rekursiv über assoziierte Objekte hinweg fortsetzen darf. Betrachtet man alle Fälle, in denen eine Synchronisation nötig ist, lassen sich diese in zwei Szenarien zusammenfassen:

- Die separate Synchronisierung eines Objektes
- Die Synchronisierung eines Objektes inklusive alle referenzierten Objekte wird soweit wie möglich rekursiv fortgeführt

Für das erste Szenario gilt, dass zu Beginn alle Attribute des Ursprungsobjektes synchronisiert werden. Anschließend werden alle bestehenden Links auf eines oder mehrere nicht primitive Objekte synchronisiert. Bei diesen gilt zum einen die Einschränkung, dass die Rekursion an dieser Stelle nicht fortgeführt werden darf, d.h. deren referenzierte Objekte werden nicht synchronisiert. Zum anderen werden nur Links bearbeitet deren Ziel das Ursprungsobjekt ist. Dieses Vorgehen liegt darin begründet, dass Kindobjekte nicht im Ganzen abgeglichen, sondern möglichst alle unidirektionale Assoziationen auf das Ursprungsobjekt erreicht werden sollen. Betrachtet man beispielsweise ein Szenario mit A als Ursprungsobjekt und B als referenziertes Kindobjekt; es gibt also eine unidirektionale Assoziation A $\rightarrow$ B. Gibt es auch einen unidirektionalen Rückweg B $\rightarrow$ A wird diese Assoziation ebenfalls mit dem anderen Modell synchronisiert, weil es einen Link zurück auf das Ursprungsobjekt darstellt. Existiert ausschließlich B $\rightarrow$ A, kann diese Assoziation nicht abgeglichen werden, weil B nicht über eine Assoziation von A ausgehend erreichbar, also kein Kindobjekt von A ist. Dies ist als Kompromiss zwischen einer möglichst vollständigen Synchronisierung eines Objektes und dem hierfür betriebenen Aufwand zu sehen.

Soll der Objektabgleich über alle synchronisierbaren Kindobjekte beliebig rekursiv fortgeführt werden, bleibt das grundsätzliche Vorgehen gleich; zuerst werden die Kindobjekte synchronisiert, anschließend die Attribute abgeglichen. Einschränkungen gibt es in keinem dieser Schritte. Es kommt allerdings eine neue Problematik zum Tragen. Wird ein Objekt A synchronisiert und es existiert eine bidirektionale Assoziation A $\leftrightarrow$ B, so resultiert hieraus die Synchronisierung von Objekt B. Durch dieselbe Assoziation wird bei der Abarbeitung der Attribute in B auch A wieder zum Abgleich freigegeben. Im Endeffekt resultiert dies in einer Endlosschleife. Eine einfach umzusetzende Gegenmaßnahme ist sich während eines Synchronisationsvorganges alle bereits besuchten Objekte lokal zu merken. So kann bei jedem neu übergebenen Objekt überprüft werden, ob es bereits bearbeitet wurde.

Insgesamt gesehen ergibt sich daraus folgender Ablauf bei der Synchronisierung:

1. Der zuständige Handler für das zu synchronisierende Objekt wird gesucht. An dieser Stelle ist die in Abschnitt 4.4 besprochene und verwendete „Chain-of-Responsibility“ relevant. Weil es nur einen zuständigen Handler geben kann, muss die Reihenfolge der Handler bei dieser Suche festgelegt und bei jedem Durchlauf identisch sein. Ansonsten kann durch die falsche Wahl ein Attribut im Anschluss falsch synchronisiert werden. Dies gilt beispielsweise für die Klasse *Property*. Diese kann im Fujaba-Metamodell entweder *FRole* oder *FAttr* zugeordnet werden. Daher gibt es für diesen Fall zwei verschiedene Handler, die unterschiedlich strenge Auswahlkriterien haben. Durch die feste Reihenfolge kann sichergestellt werden, dass die strengeren Kriterien stets zuerst überprüft werden.
2. Über das Mapping wird ein passendes Gegenstück gesucht
3. Ist dies nicht vorhanden, wird über die alle vollqualifizierenden Attribute nach diesem gesucht
4. Ist dies nicht möglich wird über den vorher gewählten Handler ein neues Objekt erstellt
5. Alle nicht primitiven Kindobjekte werden, wenn möglich, synchronisiert
6. Alle Attribute werden synchronisiert

Die Schritte 2-4 sind in Abschnitt 4.3 genauer ausgeführt. Die Reihenfolge der letzten beiden Schritte ist unerheblich und austauschbar.

Im Folgenden wird auf die technischen Unterschiede eingegangen, die sich aus der Synchronisierung eines Objektes basierend auf dem Fujaba- bzw. dem UML2-Metamodell ergeben. Bis einschließlich Schritt 4 ist das Vorgehen in beiden Fällen gleich. Daher wird ausschließlich auf die Feinheiten der letzten beiden Schritte eingegangen.

### 4.5.1 Spezielle Schritte bei Objekten aus dem Fujaba-Metamodell

Für die Synchronisierung eines Objektes aus dem Fujaba-Metamodell sind die Klassen *FujabaSynchronizer* und *FujabaSingleSynchronizer* zuständig. Letztere erweitert *FujabaSynchronizer* und überschreibt einige Methoden, um die nötigen Einschränkungen für die Synchronisierung eines Einzelobjektes festzulegen.

Wenn ein passendes Gegenstück für das abzugleichende Objekt gefunden ist, müssen alle

zu synchronisierenden referenzierten Objekte extrahiert werden. Die in Abschnitt 2.4 vorgestellte „Java Feature Abstraction“-Bibliothek bietet hierzu nützliche Hilfsklassen bzw. -methoden. Über sie bekommt man den Zugriff auf den sogenannten *ClassHandler* für ein Objekt, der unter anderem den reflektiven Zugriff auf alle existierenden Felder einer Klasse ermöglicht. Neben den Feldern der Klasse selbst, gilt dies auch für alle, die über ihre Oberklassen zugänglich sind. Dies liefert wiederum für jedes Feld einen *FieldHandler*, der Informationen über den Namen, die Sichtbarkeit etc. enthält. Jeder auf diesem Weg gefundene aktive Link zu einem anderen Objekt wird ebenfalls in den Synchronisierungsvorgang eingefügt und bearbeitet. Handelt es sich um eine Sammlung von Objekten, wird diese durchlaufen und jedes einzeln behandelt.

Für den Abgleich eines Feldes wird der Name und gesetzte Wert jedes einzelnen an den in Schritt eins gewählten Handler (die genaue Funktionsweise wird in Abschnitt 4.6 erläutert) übergeben und dort verarbeitet.

### 4.5.2 Spezielle Schritte bei Objekten aus dem UML2-Metamodell

Soll ein Objekt aus dem Metamodell von UML2 synchronisiert werden, kommen die Klassen *UML2Synchronizer* und *UML2SingleSynchronizer* zum Einsatz. Letztere Klasse erweitert *UML2Synchronizer* und überschreibt, vergleichbar mit dem Vorgehen in Fujaba, Methoden, um die Synchronisierung eines einzelnen Objektes zu ermöglichen.

Um die Felder des übergebenen Objektes auf zu synchronisierende Referenzen auszuwerten, wird die aus EMF stammende Methode `eContents()` genutzt. Diese liefert die nicht primitiven Werte der Felder als Liste zurück. Auf manche dieser Felder gibt es mehrere Zugriffsmethoden, welche jeweils ein eigenes Event erzeugen, wenn der neu hinzugefügte bzw. entfernte Wert deren Kriterien entspricht. Soll eine bestimmte Zugriffsmethode verwendet werden, kann `synchronizeAdditionalContent(NamedElement element)` in einem Handler überschrieben werden. Diese Methode ist im Normalfall ohne Funktion und kommt daher nur bei Bedarf zum Einsatz. Nötig ist dies beispielsweise bei der Klasse *Association*. Die beiden Enden einer Assoziation werden nicht automatisch über `eContents()` hinzugefügt, so dass dies manuell nachgeholt werden muss, um eine Assoziation korrekt synchronisieren zu können. Ist die Liste vollständig wird über sie iteriert, um jedes Element, wenn möglich, einzeln abzugleichen.

Um die Felder eines Objektes synchronisieren zu können, wird für alle Attribute und Referenzen aus dessen Vererbungshierarche ihr Bezeichner und ihr Wert extrahiert und in dem

in Schritt eins gewählten Handler (genaue Funktionsweise in Abschnitt 4.6) ausgewertet.

## 4.6 Attributwerte synchronisieren

Unabhängig davon, ob ein Attributwert auf Grund eines einzelnen Events (siehe Abschnitt 4.4) oder wegen der Synchronisierung eines vollständigen Objektes (siehe Abschnitt 4.5) zwischen Modellen abgeglichen werden soll, erfolgt beides in denselben Handlern. Diese bieten, neben der Attributsynchronisierung, Zugriffsmethoden für die Erzeugung oder Suche eines passenden Objektes im anderen Metamodell, d.h. in den Handlern sind alle Informationen enthalten, die sich auf die Zuordnungen zwischen zwei Objekten bzw. Klassen im Fujaba- und UML2-Metamodell beziehen.

Im Folgenden werden Bereiche dieses Vorgangs separat beschrieben, bevor zum Ende der Ablauf im Zusammenhang betrachtet wird. Alle Beispiele in diesem allgemeinen Abschnitt werden aus Sicht des Fujaba-Metamodells betrachtet. Die Unterschiede bei der Verarbeitung im Bezug auf das UML2-Metamodell sind sehr gering und daher größtenteils vernachlässigbar. Erwähnenswerte Differenzen werden im Abschnitt 4.6.6 aufgeführt.

### 4.6.1 Strukturierung der Synchronisationslogik

Betrachtet man das Fujaba- und UML2-Metamodell bzw. den Bereich, der in dieser Arbeit relevant ist, zeigen sich zahlreiche Attribute bzw. Links zwischen den Elementen, die abgeglichen werden müssen. Die Synchronisierung von diesen unterscheidet sich teilweise stark hinsichtlich der Komplexität oder der Spezialisierung. Beispielsweise kann das Attribut „name“ zwischen *Class* und *FClass* auf dieselbe Weise abgeglichen werden wie zwischen *Operation* und *FMethod*, *Parameter* und *FParam* oder allen weiteren Elementen mit diesem Attribut. Im Gegensatz dazu gibt es spezielle Attribute wie „cardinality“, welches ausschließlich bei der Synchronisierung eines Objekttyps, in diesem Fall bei den Typen *FRole* bzw. *Property*, vorhanden ist. Die Komplexität nimmt stark zu, wenn sich ein Attribut nicht exakt auf das andere Metamodell abbilden lässt, d.h. wenn die gleichen Informationen strukturell unterschiedlich verwaltet werden. Dazu gehört unter anderem die Generalisierung von Klassen. Während diese in Fujaba ausschließlich über Objekte vom Typ *FGeneralization* abgebildet werden, bietet das UML2-Metamodell verschiedene Ansätze, die vom Typ der Ober- bzw. Unterklasse abhängig sind. Somit müssen diese Fälle mit

zusätzlichen Unterscheidungsmechanismen bedacht werden. Basierend auf diesen Erkenntnissen wird die Struktur für die Synchronisierungslogik festgelegt.

Als grundsätzlicher Ansatz wird für jede Klasse aus einem der Metamodelle, die zu synchronisierende Attribute enthält, ein eigener Handler erstellt. In den Abbildungen 4.12 und 4.13 sind die Hierarchien der Handler (Diese werden in den speziellen Abschnitten 4.6.5 und 4.6.6 genauer beleuchtet) in Klassendiagrammen abgebildet. Die Synchronisierungslogik wird dort für jedes synchronisierbare Attribut in einer separaten Methode gekapselt. Das bedeutet beispielsweise für eine Klasse mit drei Attributen a, b und c, dass der für diese Klasse zuständige Handler drei Methoden besitzt, die jeweils für den Abgleich eines der Attribute zuständig ist.

```
1    public void synchronizationMethod(NamedElement targetObject, Object
         newValue, Object oldValue, FElement sourceObject)
2    {
3        //Synchronisationslogik
4    }
```

**Listing 4.1:** Beispiel für den Kopf einer Synchronisationsmethode

Um den Zugriff auf diese Methode möglichst einheitlich zu gestalten, werden die Methodenköpfe nach festen Regeln aufgebaut. Weil jedes Attribut bei der Synchronisierung unterschiedliche Ansprüche an die Menge der benötigten Informationen hat, können die Methodenköpfe den Rahmenbedingungen entsprechend individualisiert werden. Dabei müssen sie Einschränkungen bezüglich der Reihenfolge, der Parametertypen und der Anzahl der Parameter beachten. In Listing 4.1 ist ein Beispiel für einen solchen Methodenkopf zu sehen. Pro Methode kann aus den bis zu fünf verschiedenen Parametern gewählt werden:

**EventType:** Dieser Parameter kann nur bei einer Methode für das UML2-Metamodell genutzt werden. Er kann Werte wie SET, ADD oder REMOVE annehmen und lässt Rückschlüsse darauf zu, was für Auswirkungen die Synchronisierung auf das Attribut haben soll: handelt es sich um einen einzelnen Wert bzw. eine Objektsammlung und wird ein Objekt entfernt oder hinzugefügt.

**Target:** Dabei handelt es sich um das Ziel der Synchronisierung, also das Objekt im anderen Metamodell, in welchem das abzugleichende Attribut liegt. Dieser Parameter sollte immer gesetzt werden, um den Abgleich durchführen zu können. Sollte er als Wert `null` haben, also nicht gesetzt sein, wird die Synchronisierung normalerweise abgebrochen.

**NewValue:** Dies ist der neue Attributwert, der auf das Zielobjekt übertragen werden soll. Er wird für die Methoden bereits in einem mit dem Zielmetamodell kompatiblen Typ erwartet.

**OldValue:** Hierbei handelt es sich um den Attributwert, der vor der Änderung gesetzt war. Dieser ist nur bei Events verfügbar und wird beispielsweise genutzt, wenn ein Objekt aus einer Objektsammlung gelöscht wird. In diesem Fall enthält **OldValue** das gelöschte Objekt.

**Source:** Dieser Parameter enthält das Objekt, aus dem das zu synchronisierende Attribut des Quell-Metamodells stammt. Es wird nur in wenigen Ausnahmefällen benötigt, wenn neben dem Abgleich des Attributs noch zusätzliche Anpassungen im Zielobjekt vorgenommen werden müssen, so dass weitere Attribute aus dem Zielobjekt ausgelesen werden müssen. Dies ist unter anderem bei Änderungen an Assoziationsenden der Fall, weil dies Auswirkungen auf die Art der Assoziation haben kann und damit zusätzliche Anpassungen nach sich zieht.

Die Reihenfolge der Parameter ist der Auflistung entsprechend vorgeschrieben. Die Parametertypen werden speziell auf die Anforderungen der Synchronisierungsmethode angepasst. Wird beispielsweise der Name einer Methode geändert, besitzt der Parameter „targetObject“ den speziellen Typ *Operation*. Mit demselben Beispiel kann auch die Varianz in der Parameteranzahl gezeigt werden. So ist der bisherige Name für den Abgleich der Änderung nicht nötig, so dass in diesem Fall der alte Attributwert im Methodenkopf nicht auftaucht.

### 4.6.2 Zuordnung zwischen Synchronisierungsmethoden und geänderten Attributen

Nachdem eine Synchronisierungsmethode angelegt ist, muss diese als nächstes dem Attribut zugewiesen werden, welches sie abgleicht. Für den Handler muss schließlich entscheidbar sein, welche Attributänderung er an welche Methode delegiert. Für diesen Schritt gibt es beispielsweise als einfachen Ansatz die Möglichkeit alle Methoden separat in Abhängigkeit von dem Identifizierungsmerkmals ihres Attributs aufzulisten, um jeweils die Zuständigkeit festzustellen. Neben der nicht zu vernachlässigenden Laufzeit, die bei einer großen Anzahl an Alternativen auftritt, ist vor allem die Verwaltung umständlich. Für jedes neu hinzukommende Attribut müssen Änderungen an zwei Stellen vorgenommen werden. Erstens wird eine neue Methode in einem Handler eingefügt, die das Attribut synchronisiert.

Zweitens muss sie in die Auflistung der zu testenden Alternativen aufgenommen werden. Somit können Inkonsistenzen leicht zu einem fehlerhaften Verhalten, wie nicht synchronisierten Attributänderungen, führen. Dem wirkt eine Vorhaltung aller nötigen Informationen direkt an der Methode entgegen. Dazu kann in Java eine Annotation[2] genutzt werden, die an die individuellen Ansprüche angepasst wird. Weil das Fujaba- und das UML2-Metamodell unterschiedliche Rahmenbedingungen bieten, ist jeweils eine eigene Annotation vorhanden. Im Folgenden werden alle Parameter besprochen, die in beiden Annotationen enthalten sind. Die speziellen Erweiterungen dieser werden in den Abschnitten 4.6.5 bzw. 4.6.6 beschrieben.

1. **propertyName:** Ein statischer Bezeichner für das Attribut. Diese Bezeichner werden ebenfalls in den Events genutzt und ermöglichen die Identifizierung der richtigen Synchronisierungsmethode, damit ein korrekter Abgleich erfolgen kann. Weil der Inhalt in jeder Annotation individuell ist, wird kein Standardwert gesetzt. Der Inhalt des Parameters sollte allerdings auch nicht leer gelassen werden, weil sonst keine Zuordnung erfolgen kann.

2. **type:** Legt den Objekttyp des neuen (und alten) Attributwertes fest. Dieser kann den Wert ComplexElement, Type, Visibility, Generalization oder None annehmen. **type** entscheidet auf welche Weise ein Attributwert in einen zu dem Ziel-Metamodell kompatiblen Wert umgewandelt wird. Die Bedeutung der Werte wird im späteren Verlauf dieses Abschnittes genauer erläutert. Als Standardwert gilt None.

3. **addNewValue:** Ein boolescher Wert, der festlegt, ob die Methode den neuen Attributwert als Parameter beim Aufruf erwartet. Weil dies normalerweise zutrifft, ist der Standardwert true.

4. **addOldValue:** Ein boolescher Wert, der festlegt, ob die Methode den alten Attributwert als Parameter beim Aufruf erwartet. Als Standardwert ist false gewählt.

5. **addSource:** Ein boolescher Wert, der festlegt, ob die Methode das Quellobjekt als Parameter beim Aufruf erwartet. Dieses Objekt enthält das zu synchronisierende Attribut aus dem Ursprungs-Metamodell. Der Standardwert ist auf false gesetzt.

Die Parameter der Annotation stehen also in direkter Abhängigkeit zu den Parametern der Synchronisierungsmethode. Daher muss manuell dafür Sorge getragen werden, dass keine

[2] Annotationen dienen in Java zur Anreicherung des Quelltextes um strukturierte Metadaten. Sie können beispielsweise an Klassen, Methoden oder Attribute angefügt werden. Eine Annotation kann um eine beliebige Anzahl an Parametern erweitert werden, die zur Laufzeit auslesbar sein können.

Inkonsistenzen bestehen. Wird beispielsweise **addOldValue** auf true gesetzt, muss dieser Parameter auch im zugehörigen Methodenkopf aufgeführt sein, um ein fehlerhaftes Verhalten zu verhindern. Die parallele Pflege ist allerdings sehr einfach einzuhalten, weil Methode und Annotation direkt nebeneinander liegen und leicht abzugleichen sind. Auch die nachträgliche Erweiterung oder Einschränkung der Methodensignatur vereinfacht sich, weil sich die Anpassungen auf Methodenkopf und Annotation beschränken.

Durch die Annotationen wird zusätzlich die Möglichkeit geboten das Verhalten bei der Synchronisierung eines Attributs in abgeleiteten Klassen zu überschreiben. Beispielsweise existiert in einer Klasse *A* eine annotierte Methode, die für den Abgleich von Attribut „name“ zuständig ist. Wird nun in einer Klasse *B*, welche die Klasse *A* erweitert, ebenfalls eine annotierte Methode für die Synchronisierung von Attribut „name“ angelegt, wird ausschließlich die speziellere Methode aus Klasse *B* aufgerufen.

Zusammengefasst bedeuten die bisherigen Schritte: Soll ein zusätzliches Attribut behandelt werden, muss zuerst eine Methode für den Abgleich im passenden Handler angelegt und anschließend annotiert werden. Die Verarbeitungsroutine erkennt bei ihrer Initialisierung diese annotierten Methoden und merkt sie sich für zukünftige Synchronisierungen vor.

### 4.6.3 Aufruf der Synchronisierungsmethoden

Ist eine Synchronisierungsmethode angelegt und mit einer passenden Annotation versehen, kann sie schließlich aufgerufen werden. Weil die Methodenköpfe stark variieren und von den Annotationen abhängen, erfolgt kein direkter Aufruf. Stattdessen wird die zu übergebene Parameterliste dynamisch erzeugt und der reflektiv aufgerufenen Methoden übergeben. Aus diesem Vorgehen resultieren verschiedene Vor- und Nachteile. Weil diesen Methoden niemals direkt aufgerufen werden, kann bei der Kompilierung keine Überprüfung auf Typsicherheit vorgenommen werden, d.h. in einem Methodenkopf können Parametertypen genutzt werden, die möglicherweise nicht denen entsprechend, die in der übergebenen dynamischen Parameterliste enthalten sind. Um die Auswirkungen bei einem auftretenden Fehler zu minimieren, wird dieser unmittelbar nach dem versuchten Methodenaufruf abgefangen und in eine Fehlerausgabe umgewandelt.

### 4.6.4 Ablauf der Synchronisierung eines Attributes

Es gibt bei jedem Handler zwei Einstiegspunkte. Den einen, wenn über die Eventvorverarbeitung aus Abschnitt 4.4 auf ihn zugegriffen wird, den anderen für den Einstieg über einen *Synchronizer* aus Abschnitt 4.5. Diese Differenzierung ist notwendig, weil die für die Synchronisierung benötigten Informationen in beiden Fällen teilweise in verschiedenen Objekten bzw. in unterschiedlichem Umfang vorliegen und aus diesen extrahiert werden müssen. Es ist zwar denkbar diese Vereinheitlichung bereits im vorherigen Schritt vorzunehmen. Um allerdings die Möglichkeit zu bieten auf diese Einstiegspunkte auch von zusätzlichen Quellen aus zuzugreifen, ist diese Vorgehensweise vorteilhafter. Beide Einstiegspunkte werden im Folgenden in Bezug auf die Fujaba-Handler genauer beleuchtet. Weil die für das UML2-Metamodell zuständigen Handler in den relevanten Punkten identisch vorgehen, muss keine separate Betrachtung dieser erfolgen.

#### Ablauf der Synchronisierung von einem *PropertyChangeEvent*

Bei der Synchronisierung eines Attributes ausgehend von einem *PropertyChangeEvent* zeigt Listing 4.2 alle durchgeführten Verarbeitungsschritte. In Zeile 3 wird für das geänderte Attribut nach einer zugehörigen Methode gesucht, die den Abgleich den Attributwertes durchführt. Für die Identifizierung wird der Bezeichner aus dem Event extrahiert. Dieser muss mit dem Inhalt des Parameters **propertyName** in einer der Annotationen übereinstimmen, um die Suche erfolgreich abzuschließen. Sollte keine Methode gefunden werden, ist das *PropertyChangeEvent* nicht relevant und die Verarbeitung wird in Zeile 6 erfolgreich beendet. In den Zeilen 10 und 11 werden der neue bzw. alte Attributwert „umgewandelt“, d.h. die aus dem *PropertyChangeEvent* extrahierten und aus dem Fujaba-Metamodell stammenden Werte werden durch äquivalente Objekte ersetzt, die auf dem UML2-Metamodell basieren.

```
1    protected boolean handlePCE(final PropertyChangeEvent evt, final
         Element uml2Target, final Package model)
2    {
3        final Method method =
             getAdditionalMapping().get(evt.getPropertyName());
4        if (method == null)
5        {
6            return true;
7        }
8
```

```
9           final FujabaProperty fujabaProperty =
                method.getAnnotation(FujabaProperty.class);
10          final Object newUML2Value = get(evt.getPropertyName(),
                fujabaProperty.type(), evt.getNewValue());
11          final Object oldUML2Value = get(evt.getPropertyName(),
                fujabaProperty.type(), evt.getOldValue());
12
13          if (oldUML2Value != null || newUML2Value != null)
14          {
15              return this.handle(method, evt.getSource(), newUML2Value,
                    oldUML2Value, uml2Target);
16          }
17          return false;
18      }
```

**Listing 4.2:** Bereitstellung aller für die Synchronisierung benötigten Informationen aus *PropertyChangeEvent*s

Soll beispielsweise das primitive Attribut „name“ eines Objektes vom Typ *FClass* synchronisiert werden, muss irgendwann während der Verarbeitung die Identifizierung des Gegenobjektes vom Typ *Class* erfolgen, um den geänderten Attributwert auf dieses übertragen zu können. Dieser Schritt kann sehr früh während der Verarbeitung oder erst sehr spät, beispielsweise in den synchronisierenden Methoden der Handler, erfolgen. Zwei Argumente sprechen allerdings gegen das späte Vorgehen. Zum Ersten bietet ein möglichst früher Austausch ein vereinheitlichtes Vorgehen an, d.h. sollte Änderungsbedarf an diesem Austausch herrschen, muss nur an einer zentralen Stelle und nicht in vielen Methoden eine Anpassung vorgenommen werden. Zum Zweiten können hierdurch früh inkompatible Werte ausgefiltert werden, d.h. gibt es kein passendes Gegenstück in dem anderen Metamodell, kann die Bearbeitung bereits beendet werden. Zusätzlich können in den Methodenköpfen der Synchronisierungsmethoden bereits die speziellen Klassen aus dem anderen Metamodell genutzt werden, so dass die nachträgliche Typkonvertierung ausgespart werden kann. Als Konsequenz dieses Vorgehens muss der Austausch sehr generisch angelegt sein. Dazu werden Attribute in eine von fünf Gruppen eingeteilt, die in der Annotation durch den Parameter **type** festgelegt wird:

**ComplexElement:** In diese Gruppe fallen alle Objekte, welche zum einen auf Klassen aus einem der Metamodelle basieren und zum anderen in dem zentralen Mapping, beschrieben in Abschnitt 4.3.3, enthalten sind. Über dieses wird auch das passende Gegenstück

gesucht.

**Type:** Diese Gruppe ist die umfassendste, weil sie zusätzlich zu allen Elementen der Gruppe **ComplexElement** auch alle primitiven Objekte enthält. Der Zugriff erfolgt über die in Abschnitt 4.3 beschriebenen Methoden der Klasse *Mapping*. Genutzt wird sie beispielsweise bei der Synchronisierung des Typs eines Parameters oder wenn der Typ des Rückgabewertes einer Methode abgeglichen wird.

**Visibility:** Diese Gruppe wird ausschließlich bei der Synchronisierung der Sichtbarkeit von Klassen, Methoden, Attributen, etc. genutzt. Nötig ist sie, weil Fujaba und UML2 zwei sehr unterschiedliche Konzepte bei der Verwaltung der Sichtbarkeit verwenden. Während diese in Fujaba über eine statische *Integer*-Variable ausgedrückt wird, kommt in UML2 ein komplexes Objekt vom Typ *VisibilityKind* zum Einsatz.

**Generalization:** Auch diese Gruppe hat einen sehr speziellen Einsatzzweck. Sie wird bei der Synchronisierung einer Vererbungshierarchie zwischen Klassen bzw. Interfaces verwendet. Der Grund liegt in der ebenfalls sehr unterschiedlichen Herangehensweise in beiden Metamodellen. Während in Fujaba die Vererbung über ein Objekt vom Typ *FGeneralization* modelliert wird, nutzt UML2 verschiedene Ansätze, beispielsweise Objekte vom Typ *InterfaceRealization* oder *Generalization*.

**None:** Diese Gruppe bildet die Ausnahme. Wird sie gewählt bedeutet dies, dass kein Austausch erfolgen soll, das übergebene Objekt also schlicht zurückgegeben wird. Ein naheliegendes Einsatzgebiet sind primitive Attributwerte. Wird in beiden Metamodellen für zwei zueinander gehörende Attribute beispielsweise ein *boolean* als Typ verwendet ist keine Umwandlung des Wertes nötig. Diese Gruppe dient auch als Standardeinstellung, d.h. sollte keine andere Gruppe für ein Attribut vermerkt sein, erfolgt kein Austausch des Objektes.

Wie sich zeigt, können manche Objekte durchaus in mehreren der Gruppen vorkommen, aber jede Gruppe umfasst für sich eine separate Objektmenge, die auf den jeweiligen Einsatzzweck abgestimmt ist.

Sollte für den neuen oder alten Attributwert ein passendes Objekt im jeweils anderen Metamodell gefunden werden, wird die Synchronisierung in Zeile 15 der Methode (siehe Listing 4.2) an den letzten Verarbeitungsschritt übergeben. Dort werden die Parameter der für den Abgleich zuständigen Methode gesammelt, um diese schließlich aufzurufen und den Vorgang abzuschließen.

### Ablauf der Synchronisierung eines Attributes eingeleitet durch den vollständigen Abgleich eines Objektes

Wird der Abgleich eines Attributes während der vollständigen Synchronisierung eines Objektes initiiert, so kommt der Einstiegspunkt aus Listing 4.3 zum Einsatz. In Zeile 4 wird die zuständige Synchronisierungsmethode gesucht. Ist diese Suche erfolglos, soll dieses Attribut nicht abgeglichen werden und der Vorgang wird in Zeile 7 beendet. In Zeile 11 wird der aktuelle Attributwert „umgewandelt“. An dieser Stelle ist keine Unterscheidung zwischen einem alten und neuen Wert möglich. Weil der Wert direkt aus dem zu synchronisierenden Objekt extrahiert wird, ist ausschließlich der aktuelle Attributwert bekannt.

```
1     public boolean handle(final String propertyName, final FElement
          fElement, final Object fValue,
2               final Element uml2Target)
3     {
4           final Method method = getAdditionalMapping().get(propertyName);
5           if (method == null)
6           {
7               return true;
8           }
9           final FujabaProperty fujabaProperty =
              method.getAnnotation(FujabaProperty.class);
10
11          Object newUML2Value = get(propertyName, fujabaProperty.type(),
              fValue);
12          return this.handle(method, fElement, newUML2Value, null,
              uml2Target);
13    }
```

**Listing 4.3:** Bereitstellung aller für die Synchronisierung benötigten Informationen aufbauend auf dem *FujabaSynchronizer*

In Zeile 12 erfolgt schließlich der Übergang in den letzten Verarbeitungsschritt. Dort werden die Parameter für die Synchronisierungsmethode ausgewählt, bevor diese aufgerufen wird um den Abgleich abzuschließen.

Im Folgenden wird auf technische Unterschiede und Feinheiten bei der Synchronisierung eines Attributwertes basierend auf dem Fujaba- oder dem UML2-Metamodell eingegangen.

### 4.6.5 Details der Handler für das Fujaba-Metamodell

Abbildung 4.12 zeigt das Klassendiagramm aller Handler für das Fujaba-Metamodell. Zu Gunsten der Übersichtlichkeit sind die enthaltenen Synchronisierungsmethoden ausgeblendet. In den abstrakten Handlern *AbstractFujabaUMLHandler* und *AbstractUMLClassifierHandler* sind alle Verarbeitungsschritte enthalten, für die eine Bündelung möglich ist.

Als Beispiel für die Synchronisierung eines Attributwertes aus dem Fujaba-Metamodell

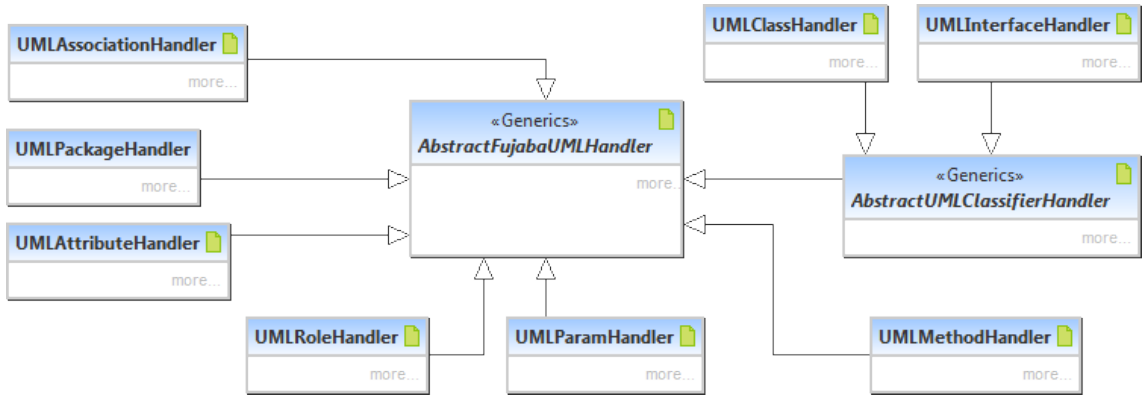



**Abbildung 4.12:** Klassendiagram der Fujaba-Handler

dient Listing 4.4. Zeile 1 zeigt die Annotation der Methode. *FujabaProperty* kann die fünf in Abschnitt 4.6.2 bereits vorgestellten Parameter **parameterName**, **type**, **addNewValue**, **addOldValue** und **addSource** besitzen.

```
1   @FujabaProperty(propertyName = FDeclaration.VISIBILITY_PROPERTY,
        type = SynchronizationType.VisibilityKind)
2   public void visibility(final NamedElement targetElement, final
        VisibilityKind visibility)
3   {
4       if (targetElement != null)
5       {
6           FujabaUtils.synchronize(new
                UMLLabRunnable<NamedElement>(targetElement)
7           {
8               public void update(NamedElement toSync)
9               {
10                  toSync.setVisibility(visibility);
11              }
12          });
13      }
```

```
14      }
```

**Listing 4.4:** Beispielmethode eines Handlers zuständig für das Fujaba-Metamodell

Betrachtet man unter diesen Gesichtspunkten die Annotation in diesem Listing, wird das Attribut mit dem Bezeichner „VISIBILITY_PROPERTY“ aus der Objektgruppe „VisibilityKind“ synchronisiert, wobei nur der neue Attributwert übergeben werden muss, während der alte Wert und das Quellobjekt nicht relevant sind. Dementsprechend ist auch der Methodenkopf in Zeile 2 aufgebaut. Der erste Parameter enthält das Zielobjekt, der zweite den neuen Attributwert, der bereits zum UML2-Metamodel kompatibel ist. Zwischen den Zeilen 6 und 12 ist die Synchronisierung des Attributwertes zu sehen. Bei UML Lab dürfen Änderungen im UML2-Modell von außen nicht direkt durchgeführt werden, sondern müssen in einem *Runnable*-Objekt gekapselt sein, welches der API von UML Lab übergeben wird. Dabei handelt es sich um Instanzen der Klasse *UML2Runnable*, einer Implementation des *Runnable*-Interfaces. Dort wird die in Abschnitt 4.2.1 erläuterte Sperrung für Metamodell-Objekte, die sich in Synchronisation befinden, verwaltet.

Durch die Nutzung von Objekten des Typs *FStereotype* kann ein Element aus dem Fujaba-Metamodell um zusätzliche Informationen angereichert werden. Beispielsweise wird über den Stereotyp „interface“ ein Objekt vom Typ *FClass* statt als Klasse als Interface interpretiert. Weil diese Einflussnahme sich auch auf den für die Synchronisierung relevanten Bereich des Metamodells bezieht und Stereotypen unterschiedliche Auswirkungen auf das Zielobjekt haben, müssen sie in den Handlern berücksichtigt werden. Um nicht in jedem relevanten Handler eine Methode inklusive passender Annotation anlegen zu müssen, gibt es eine Standardimplementierung, die bei Bedarf erweitert werden kann. Dazu muss die Methode `handleStereotypeSpecific(...)` überschrieben werden. Diese erwartet ein *boolean* als Rückgabewert, um behandelte Stereotypen zu erkennen und für die übrigen eine Standardbehandlung vorzunehmen, die eine String-Repräsentation des Stereotyps als Keyword (ein für vergleichbares nutzbares Attribut aus dem UML2-Metamodell) an das Zielobjekt überträgt.

### 4.6.6 Details der Handler für das UML2-Metamodell

Die Abbildung 4.13 zeigt das Klassendiagramm aller Handler für das UML2-Metamodell. Um eine ausreichende Übersichtlichkeit zu gewähren sind alle Synchronisierungsmethoden ausgeblendet. In den abstrakten Handlern sind alle Methoden, die Attribute bezüglich mehrerer Klassen synchronisieren können, gebündelt.

Der genaue Ablauf bei der Synchronisierung eines Objektes aus dem UML2-Metamodell

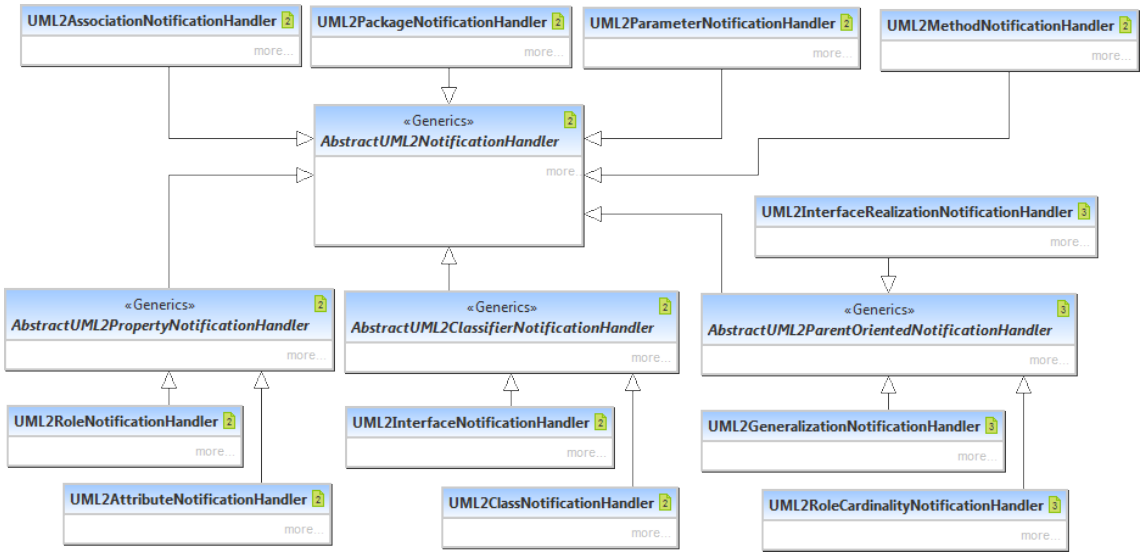



**Abbildung 4.13:** Klassendiagram der UML2-Handler

ist in Listing 4.5 an einem Beispiel zu sehen. Zeile 1 zeigt die Annotation (*UML2Property*) zur Markierung der Methode. Diese kann aus bis zu sieben Parametern bestehen:

1. **property:** entspricht **propertyName** aus Abschnitt 4.6.2
2. **type:** siehe Abschnitt 4.6.2
3. **addEventType:** Ein boolescher Wert, der festlegt, ob die Methode den Eventtyp (bspw. ADD, SET, REMOVE) als Parameter beim Aufruf erwartet. Weil dieser sehr selten benötigt wird, ist der Standardwert false.
4. **addNewValue:** siehe Abschnitt 4.6.2
5. **addOldValue:** siehe Abschnitt 4.6.2
6. **addNotifier:** entspricht **addSource** aus Abschnitt 4.6.2
7. **targetMayBeNull:** Dieser Parameter ist für wenige Sonderfälle nötig und daher im Normalfall auf false gesetzt. Andernfalls verhindert er, dass die Synchronisierung abgerochen wird, wenn für das Quellobjekt kein passendes Zielobjekt im Fujaba-Modell gefunden wird. Genutzt wird dies beispielsweise bei der Änderung des Rückgabetyps einer Methode. Dieser ist im Fujaba-Modell ein Attribut von *FMethod*, im UML2-Modell hingegen ein als spezieller Parameter in die Liste einer *Operation* eingehängt.

Weil dieser Parameter nicht in beiden Modellen existiert, kann kein Gegenstück gefunden werden.

```
1  @UMLProperty(property = UMLPackage.CLASS__OWNED_OPERATION, type =
       SynchronizationType.ComplexElement, addOldValue = true,
       addEventType = true)
2  public void operation(final int eventType, final FClass
       fujabaClass, final FMethod operation,
3          final FMethod oldOperation)
4  {
5      if (eventType == Notification.ADD && operation != null)
6      {
7          operation.setParent(fujabaClass);
8      } else if (eventType == Notification.REMOVE && oldOperation !=
           null)
9      {
10         fujabaClass.removeFromMethods(oldOperation);
11     }
12 }
```

**Listing 4.5:** Beispielmethode eines Handlers zuständig für das UML2-Metamodell

Überträgt man diese Erklärungen auf die Annotation im Beispiellisting, behandelt die Methode das Attribut „OWNED_OPERATION“ aus der Objektgruppe „ComplexElement“ und erwartet in den Parametern neben dem Eventtyp den neuen und alten Attributwert. Dafür ist das Quellobjekt nicht notwendig, das Zielobjekt muss jedoch gesetzt sein. Im Methodenkopf in Zeile 2 ist zu sehen, dass die Parameterklassen so konkret wie nötig gewählt werden (FClass, FMethod). Im Methodenrumpf zwischen Zeile 5 und 11 wird das Hinzufügen und das Löschen einer Methode aus einer Klasse behandelt. Die Änderungen am Zielobjekt können unmittelbar ausgeführt werden.

Während sich die meisten Synchronisierungen auf ein Attribut von zwei zueinander gehörenden Objekten in den beiden Metamodellen beziehen, wird bei manchen aus diesem Muster ausgebrochen. Betrachtet man beispielsweise den Abgleich der Kardinalitäten von beiden Enden einer Assoziation, offenbaren sich deutliche Unterschiede zwischen dem Fujaba- und UML2-Metamodell. In Fujaba gibt es die Klasse *FCardinality*, welche die obere und untere Schranke eines Assoziationsendes festgelegt. Ändert sich die Kardinalität nun beispielsweise von „0..n“ auf „1..n“ wird nicht das *FCardinality*-Objekt angepasst, sondern es wird durch ein anderes ausgetauscht. Im Gegensatz dazu gibt es in UML2 für die obere und untere Schranke jeweils ein eigenes Objekt vom Typ *ValueSpecification*. Ändert sich hier die Kardinalität von „0..n“ auf „1..n“ wird der Wert der unteren Schranke von 0 auf 1 geändert.

In Fujaba wird also ein Event auf der Elternebene, an *FRole*, erzeugt. In UML2 erfolgt dies hingegen nicht an dem Äquivalent *Property*, sondern auf der Kindebene, an *ValueSpecification*. Von Fujaba aus ist diese Zuordnung leicht zu realisieren, weil der Handler für *FRole* dieses Event erhält und entsprechend verarbeiten kann. Für UML2 wird eine Erweiterung der bisherigen Handler benötigt. *AbstractUML2ParentOrientedNotificationHandler* (siehe Abbildung 4.12) bildet die Basis für diese Erweiterung. Sie sorgt für das richtige Mapping von Kindelement (*ValueSpecification*) aus UML2 auf Elternelement (*FRole*) aus Fujaba. Auf diesem Weg kann der Methode eines Handlers das Zielobjekt übergeben werden, welches von der Synchronisierung betroffen ist. Dies bedeutet, dass Zwischenebenen, die durch die Nutzung eines zusätzlichen Objektes wie eine *ValueSpecification* auftreten, ausgelassen werden können.

## 4.7 Verwaltung von ungebundenen Story-Diagrammen

Durch die Synchronisierung bewirkt das Löschen einer Methode in einem UML2-Modell die Entfernung seines Gegenstücks im Fujaba-Modell. Weil dies durchaus unbedacht passieren kann, wird möglicherweise vergessen, dass diese Methode bereits durch ein Story Diagramm in Fujaba erweitert wurde. Dieses geht bei der Löschung normalerweise verloren. Um zu verhindern, dass solche Diagramme ungewollt entfernt werden, ist es nötig, sie separat aufzubewahren, damit sie durch den Anwender an eine neue Methode zugewiesen werden können.

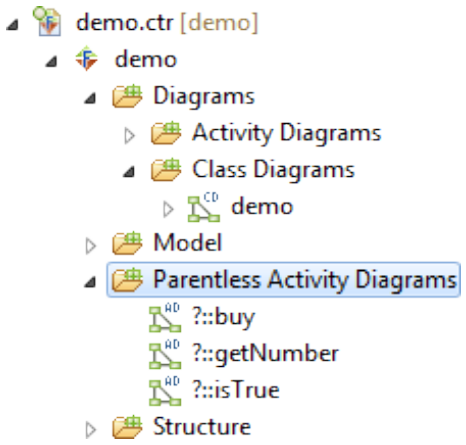



**Abbildung 4.14:** Übersicht über alle elternlose Story-Diagramme

Eine Verwaltung für die Zwischenspeicherung der Story-Diagramme ist leicht umzusetzen. Dazu genügt eine leicht erreichbare Liste für solche Diagramme. Diese wird gefüllt, wenn eine Methode aus einem UML2-Modell entfernt wird. Weil durch die Löschung eines Elementes aus einem Fujaba-Modell alle Inhalte dessen ebenfalls aus dem Modell entfernt werden, muss das Story-Diagramm vor der Synchronisierung der Löschung aus der Methode entfernt und separat gespeichert werden. Sobald eines der Diagramme wieder einer neuen Methode zugewiesen wird, wird es aus der Liste wieder herausgenommen.

Neben der internen Verwaltung sollen alle betroffenen Story-Diagramme auch für den Anwender sichtbar sein. Zudem muss es eine Möglichkeit geben, wie er diese an neue Methoden zuweist. Dazu wird auf eine Erweiterung des Fujaba4Eclipse-Plugins zurückgegriffen, die es ermöglicht, den sogenannten „Project Explorer“ zu erweitern. Zum einen können auf diesem Weg Elemente des Modells nach eigenen Kriterien angezeigt werden. Das bedeutet in diesem Fall, dass alle als elternlos markierten *UMLActivityDiagram*-Objekte separat zu erreichen sind. In Abbildung 4.14 ist der hinzugefügte Eintrag inklusive drei elternloser Story-Diagramme zu sehen. Um diese wieder neuen Methoden zuzuweisen, kann der Drag&Drop-Support genutzt werden, der von der Eclipse-Platform bereitgestellt und durch das Fujaba4Eclipse-Plugin erweitert wird. Dazu kann ein eigener Handler eingesetzt werden, der auf die Kombination *FMethod* - *FDiagram* reagiert. Wenn das Diagramm als elternlos erkannt wird, erfolgt seine Zuweisung an die übergebene Methode und schließlich die Entfernung aus der Verwaltungsliste.

## 4.8 Verknüpfen/Laden von zwei zu synchronisierenden Projekten

Für das Laden und Verknüpfen von Projekten zur Synchronisierung sind zwei Punkte relevant. Die Information, welche Projekte miteinander verknüpft sind, soll über die Laufzeit der einmaligen Bearbeitung hinaus existieren. Zusätzlich soll das Mapping zueinander gehörender Objekte nicht beim Schließen eines Projektes verloren gehen. Auf diese Weise wird sichergestellt, dass außerhalb des Synchronisationsprozesses vorgenommene Veränderungen bei der nächsten Synchronisierung dem anderen Modell korrekt zugeordnet werden können.

Alle verknüpften Projekte (aktiv oder inaktiv) werden über eine *ConnectedProjectsCollection* verwaltet. Diese enthält pro Paar einen Eintrag, in dem die Mappings aller enthaltenen Projekte und zusätzlich genauere Informationen zu den Projekten verwaltet werden. Dazu gehören neben dem Projekt-Objekt selbst dessen aktuelle Lage im Dateisystem und ein Bezeichner zur Identifizierung. Um diese Verknüpfungen von Projekten zu persistieren wird, wegen der einfachen Festlegung der Datenstruktur und der unkomplizierten Erweiterbarkeit, XML gewählt. Alle zur Wiederherstellung nötigen Informationen werden bei einer Änderung in die XML-Struktur überführt und anschließend im aktuellen Eclipse-Workspace als Datei abgelegt. Somit kann diese Konfigurationsdatei während der Initialisierung des

Plugins eingelesen und alle bekannten Verknüpfungen wiederhergestellt werden. Wird ein Projekt geladen, welches sich einem der Einträge zuweisen lässt, erfolgt die automatische Ergänzung dessen um die Instanz des Modells. Solange nicht beide zueinander gehörenden Projekte geladen sind, hat dies keine weiteren Auswirkungen. Erst nach dem Laden des zweiten Projektes wird die Verknüpfung als aktiv erachtet und die Synchronisierung kann beginnen. Erfolgt das Schließen eines Projektes, wird die Verknüpfung automatisch deaktiviert. Auf die Persistierung hat dies alles keinen Einfluss. Erst wenn eine Verknüpfung vollständig entfernt wird oder das Identifizierungsmerkmal angepasst wird, muss die Änderung im Dateisystem abgelegt werden.

Um ein Mapping zwischen zwei Objekten dauerhaft zu erhalten gibt es zwei Möglichkeiten. Zum einen kann es separat gespeichert werden, zum anderen zu einem der beiden Projekte hinzugefügt werden. Die letztere Lösung ist vorzuziehen, weil keine zusätzliche Datei verwaltet werden oder diese bei der Übertragung eines Modellpaares in eine andere Entwicklungsumgebung beachtet werden muss. Das Fujaba-Metamodell bietet eine genau zu diesen Anforderungen passende Struktur. So können an jedes für das Mapping erlaubte Element *FTag*-Objekte hinzugefügt und persistiert werden. Ein *FTag* ist eine Kombination aus einem Schlüsselwert und einem Inhaltswert. Mit Hilfe einer Konstante als Schlüsselwert, kann ein *FTag* für ein Objekt eindeutig zur Speicherung eines Mappings gekennzeichnet werden. Für den Inhaltswert muss ein Bezeichner gefunden werden, über den das zugehörige Objekt im UML2-Modell spezifisch identifiziert werden kann. Hierfür wird im Metamodell bereits eine ausreichende Lösung angeboten, indem über die *Resource* des Modells für jedes enthaltene Objekt eine eindeutige String-Repräsentation abgerufen werden kann. Diese wird dem *FTag* als Inhalt hinzugefügt und ermöglicht die Identifizierung einzelner Objekte im UML2-Modell. Diese Vorgehensweise schreibt bei der Synchronisierung von Projekten eine feste Reihenfolge vor. Das Fujaba-Modell muss stets zuerst synchronisiert werden, um die enthaltenen *FTag*s auswerten zu können. Sollte eines vorhanden sein, kann mit dem Inhalt das zugehörige Objekt direkt im UML2-Modell gesucht werden. Bei der entgegen gesetzten Reihenfolge müsste pro Objekt das gesamte Fujaba-Modell durchlaufen werden und jedes möglicherweise passende Objekt nach einem *FTag* untersucht werden. Das resultiert in einem deutlich höheren Arbeitsaufwand und somit in einer deutlich langsameren Laufzeit.

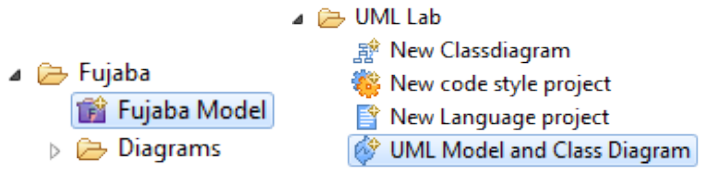
Wird die Verknüpfung eines Projektpaares vervollständigt, muss direkt im Anschluss eine Synchronisierung erfolgen, um beide Modelle auf denselben Stand zu bringen. Dabei ist es irrelevant, ob die Verknüpfung neu angelegt oder als bereits existierende durch Laden von Projekten vervollständigt wird. Auf die Synchronisierung von neu angelegten Model-

len muss nicht genauer eingegangen werden, weil in diesen noch leeren Modellen außer dem Wurzelpaket keine weiteren Elemente zu finden sind. Handelt es sich um bereits gefüllte Modelle, muss die Reihenfolge und das Vorgehen bei der Synchronisierung festgelegt werden. Die Reihenfolge resultiert bereits durch die Einschränkung, dass auf Grund der Nutzung von *FTag*s mit dem Fujaba-Modell begonnen werden muss. Für die Synchronisierung dieses Modells kommen zwei unterschiedliche Herangehensweisen in Frage. Bei der ersten wird von Beginn an jeder eingehende Event normal verarbeitet. Weil bei Fujaba jeder Event von der Erstellung eines Objektes an empfangen wird, kann somit jeder Schritt während der Erstellung nachvollzogen und abgeglichen werden. Daraus resultieren allerdings zwei gravierende Nachteile. Weil die Persistierungsbibliothek von Fujaba auch die Versionierung der Projekte unterstützt, wird während der Wiederherstellung nicht nur der aktuelle Zustand rekonstruiert, sondern zusätzlich auch alle Zwischenschritte nachvollzogen. Dazu gehört beispielsweise die Erstellung und spätere Löschung von nicht weiter benötigten Objekten oder die wiederholte Änderung desselben Attributs mit verschiedenen Werten. Es werden daher zu Lasten der Laufzeit deutlich mehr Events ausgewertet, als für den Abgleich des Endzustandes nötig wären. Als zweiter Nachteil können Seiteneffekte zur unerwünschten Duplizierung eines Objekts führen. Wenn die Events bereits während dem Ladevorgang des Fujaba-Modells synchronisiert werden, erfolgt parallel auch die Erstellung der Mappings zusammen gehörender Objekte in den beiden Modellen. Wird nun im UML2-Modell wegen eines zu synchronisierenden Links ein Gegenobjekt gesucht, was im Fujaba-Modell noch nicht existiert, aber in der Folge noch geladen werden würde, erfolgt unzulässigerweise die Erstellung des Objektes. Aus diesem Grund werden beide Modelle erst komplett geladen und anschließend im Ganzen synchronisiert.

# 5 Bedienung

Weil sowohl Fujaba als auch UML Lab über Plugins in Eclipse eingebettet sind, werden auch die Bedienelemente zur Nutzung und für Anpassungen an der Projektsynchronisierung als Eclipse-Plugin umgesetzt. Im Folgenden werden anhand eines Beispielszenarios alle nützlichen Bedienelemente gezeigt und beschrieben. Dabei dient das Anlegen neuer Projekte und befüllen dieser mit Hilfe des Reverse-Engineering von UML Lab als roter Faden. Zudem werden einige Elemente manuell angepasst, ein Story-Diagramm angelegt und schließlich für die modellierten Klassen Quellcode generiert.

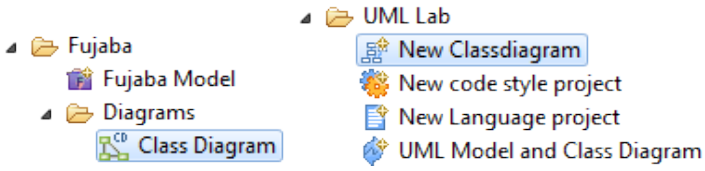
Um neue Projekte anzulegen sind für Fujaba und UML Lab jeweils Menüpunkte im ent-




(a) Fujaba (b) UML Lab

**Abbildung 5.1:** Anlegen eines neuen Projektes in Fujaba und UML Lab

sprechedenen Eclipse-Wizard vorhanden. In Abbildung 5.1 sind beide in Ausschnitten zu sehen. Legt man ein neues Projekt an, kann daraufhin das Gegenstück automatisch erzeugt werden, d.h. erstellt man ein neues Projekt für Fujaba, wird selbständig eines für UML Lab erzeugt und beide miteinander für die Synchronisierung verknüpft. Zur weiteren Bearbei-




(a) Fujaba (b) UML Lab

**Abbildung 5.2:** Anlegen eines neuen Klassendiagramms in Fujaba und UML Lab

tung müssen als nächstes Klassendiagramme angelegt werden. Dies erfolgt über denselben Dialog wie soeben und ist in der Abbildung 5.2 zu sehen. Bei UML Lab kann dieses bei der Erzeugung eines Modells automatisch erstellt werden. Im Falle einer externen .uml-Datei oder bei einem gelöschten bzw. zusätzlichen Klassendiagramm ist die manuelle Erzeugung allerdings nötig. Im nächsten Schritt sollen existierende Klassen importiert werden.

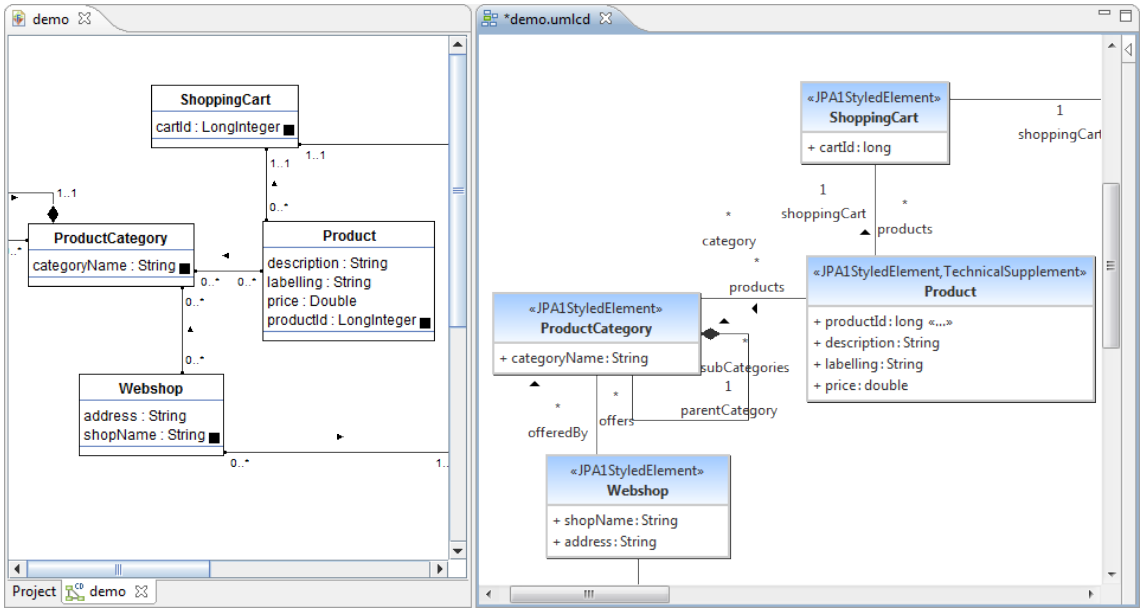
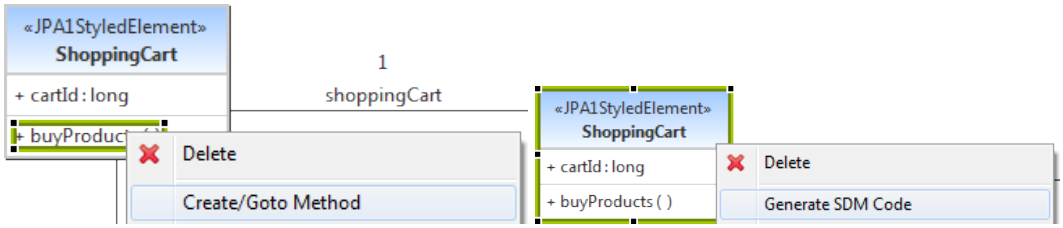



**Abbildung 5.3:** Ausschnitt der synchronisierten Modelle nach dem ReverseEngineering

Dazu müssen alle gewünschten Klassen per Drag&Drop in das Klasseneditorfenster von UML Lab gezogen werden. Durch die Synchronisierung bewirkt das Reverse-Engineering in UML Lab den parallelen Aufbau des gleichen Modells in Fujaba. Das Ergebnis für einige Beispielklassen in Abbildung 5.3 zu sehen. Die zueinander gehörenden Klassen sind an vergleichbaren Positionen platziert, um die Ähnlichkeit zu verdeutlichen. Nun soll eine Methode angelegt und für diese ein Story-Diagramm modelliert werden. Das kann entweder auf normalem Wege über Fujaba oder aus UML Lab heraus geschehen. Dazu ist bei Aufruf auf einer Methode das Kontextmenü um den Eintrag „Create/Goto Method“ erweitert. Dieser leitet den Anwender in das geöffnete Story-Diagramm in Fujaba weiter. Dort kann anschließend, wie gewohnt, der gewünschte Inhalt modelliert werden.

Ist dieser Schritt vollendet, bleibt zum Abschluss die Codegenerierung. Um den Quellcode inklusive der Story-Diagramme aus UML Lab heraus zu generieren, müssen zuerst alle nötigen Arbeiten von Fujaba ausgeführt werden, d.h. für alle Diagramme wird Code generiert[5] und in das Modell aus UML Lab eingefügt. Dieser Vorgang muss manuell ausgelöst werden. Hierzu ist das Kontextmenü in UML Lab, wie in Abbildung 5.4(b) zu sehen, um den




(a) Menüeintrag um ein Story Diagramm aus UML Lab heraus aufzurufen

(b) Menüeintrag um den Quellcode für Story Diagramme zu generieren

Eintrag „Generate SDM Code“ erweitert. Wählt man ihn aus, unterscheidet sich das Resultat je nach selektiertem Objekt. Ist beispielsweise zurzeit eine Methode im Fokus, wird nur für diese Methode Quellcode erzeugt; ist eine Klasse ausgewählt, begrenzt sich die Generierung auf alle in ihr enthaltenen Methoden. Schließlich wird dieser Abschnitt durch die Codegenerierung von UML Lab vollendet. Diese ist ebenfalls über das Kontextmenü unter „Generate Code“ erreichbar und kombiniert den Quellcode der Klassendiagramme mit dem der Story-Diagramme. Ein einfaches Beispiel für die Kooperation bei der Codegenerierung für eine Methode ist unter Listing 5.1 zu sehen.

```
1   public void buyProducts() {
2       boolean fujaba__Success = false;
3       Iterator fujaba__IterThisToProduct = null;
4       Product product = null;
5
6       // story pattern storypatternwiththis
7       try {
8           fujaba__Success = false;
9
10          // iterate to-many link from this to product
11          fujaba__Success = false;
12          fujaba__IterThisToProduct = this.iteratorOfProducts();
13
14          while (fujaba__IterThisToProduct.hasNext()) {
15              try {
16                  product = (Product)
                        fujaba__IterThisToProduct.next();
17
18                  // check object product is really bound
19                  JavaSDM.ensure(product != null);
20                  System.out.println(product.getProductId());
21
22                  fujaba__Success = true;
23              } catch (JavaSDMException fujaba__InternalException) {
```

```
24                      fujaba__Success = false;
25                  }
26              }
27              JavaSDM.ensure(fujaba__Success);
28              fujaba__Success = true;
29          } catch (JavaSDMException fujaba__InternalException) {
30              fujaba__Success = false;
31          }
32
33          return;
34      }
```

**Listing 5.1:** Ausschnitt aus einer generierten Java-Datei

## 5.1 Einstellungen

Weil einige Verhaltensweisen vom Benutzer beeinflussbar sein sollen, ist dies über eine Integration in das Eclipse-Einstellungs-Menü möglich. Diese umfasst die Festlegung des Ablaufs beim Laden eines neuen Modells und die Darstellung aller zurzeit miteinander verbundenen Modelle.

Wie in Abbildung 5.4 zu sehen, ist das Vorgehen beim Laden eines noch nicht mit einem

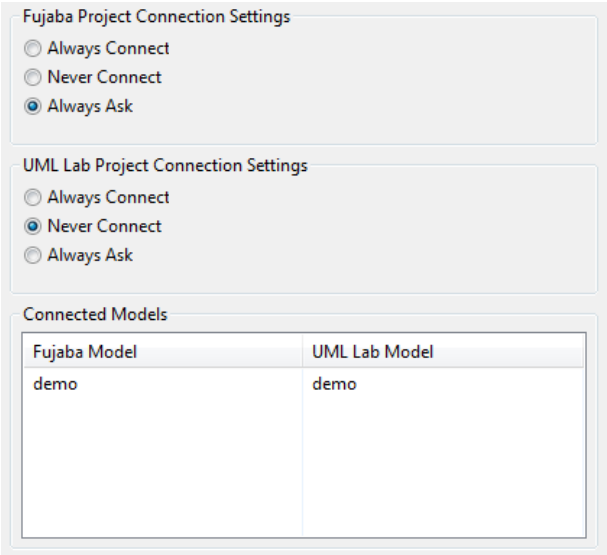



**Abbildung 5.4:** Übersicht der Einstellungsmöglichkeiten

zweiten Modell verknüpften Projektes festzulegen. Dieses kann für Fujaba-Projekte bzw.

die über UML Lab geladenen Modelle separat bestimmt werden. Durch die jeweils oberste Auswahl wird die Verknüpfung, das bedeutet gegebenenfalls auch das zweite Modell, automatisch erzeugt. Durch den mittleren Punkt wird grundsätzlich kein Versuch unternommen ein zweites Modell für die Synchronisierung zu finden. Ist der unterste Punkt ausgewählt, wird der Anwender für jeden Fall einzeln nach der gewünschten Reaktion gefragt.

Der untere Bereich der Einstellungen ist zur Information angedacht und zeigt alle aktiven und verknüpften Modellpaare.

# 6 Fazit

Diese Arbeit befasst sich mit der Echtzeit-Synchronisierung von zwei Modellen basierend auf der UML-Spezifikation in verschiedenen Versionen. Die Aufgabenstellung beschränkt den Umfang auf Bereiche der Metamodelle, die Klassendiagramme betreffen.

Besonders der Tatsache, dass die Synchronisierung zur Laufzeit vorgenommen werden soll, ist eine große Bedeutung in der Implementierung zuzuordnen. Die Reihenfolge, in der Änderungen an einem Modell auftreten, hängt hierbei im Allgemeinen von dem Anwender und seinen Aktionen (wann legt er welches Objekt an bzw. ändert welches Attribut) bzw. im Speziellen von der Realisierung des grundlegenden Metamodells (wie werden zusammenhängende Strukturen, z.B. Assoziation erzeugt) ab. Daraus resultiert der Bedarf nach einer sehr flexiblen Struktur des Synchronisierungsmechanismus. Eine der Herausforderungen dieser Arbeit lag somit darin, sich nicht durch tiefergehendes Wissen über die Abläufe der implementierten Metamodelle beeinflussen zu lassen, um versteckte, potentielle Fehlerquellen zu vermeiden. Stattdessen muss jede Änderung einzeln und unbeeinflusst von seinen vorhergehenden und nachfolgenden Änderungen bearbeitet werden.

Ein weiteres Augenmerk dieser Arbeit liegt in der einfachen Erweiterbarkeit der Umsetzung, um die Synchronisierung in Zukunft auf weitere Bereiche der Metamodelle auszudehnen zu können. Hierzu sind viele der Verarbeitungsschritte so allgemein wie möglich gehalten und es wird erst sehr spät auf speziell angepasste Klassen zurückgegriffen.

Neben der Implementierung der Synchronisierungslogik liegt ein weiterer Schwerpunkt auf umfangreichen Unit-Tests. Indem eine möglichst breit angelegte Varianz in getesteten Szenarien gewährleistet wird, konnten Fehler bei Feinheiten der Umsetzung schnell erkannt und zeitsparend behoben werden. Durch die Bereitstellung der meisten Tests bereits sehr früh in der Entwicklungsarbeit, wurde die Korrektheit beim Abgleich einzelner Attribute von Beginn an verifiziert. Somit machten nur nicht beachtete Sonderfälle spätere Anpassungen nötig.

Schließlich soll sich auch die Integration in ein Eclipse-Plugin möglichst nahtlos in die Fujaba und UML Lab-Umgebung einbinden. Dazu boten sowohl die Eclipse-Platform selbst,

als auch die beiden Umgebungen im Speziellen die benötigte Infrastruktur an, damit der Anwender auf entwickelten Funktionalitäten innerhalb der Fujaba bzw. UML Lab-Bereiche zurückgreifen kann.

## 6.1 Limitierungen der Umsetzung

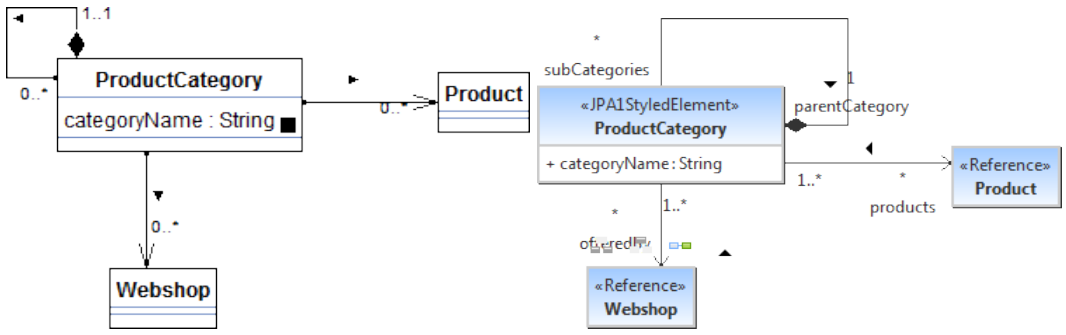
Trotz der Bemühungen bei der Umsetzung der Synchronisierung möglichst alle auftretenden Fälle abzudecken, sind einige Limitierungen geblieben, die bei der Benutzung bedacht werden müssen. Im Folgenden werden sie zum einen ausführlich beschrieben und zum anderen wird erläutert, wie mit diesen umzugehen ist.

Eine Beschränkung, die sich ohne manuellen Eingriff nicht zufriedenstellend lösen lässt, ist eine separate, konkurrierende Änderung in zwei zueinander gehörenden Modellen. Damit dieses Problem auftritt, muss jedes Modell einzeln, also ohne aktive Synchronisierung, bearbeitet werden. Als zusätzliches Kriterium muss die Änderung an den gleichen Objekten durchgeführt werden, d.h. es wird beispielsweise in beiden Modellen die gleiche Klasse umbenannt. Schließlich muss die Änderung zudem unterschiedlich sein. Im Fujaba-Modell nennt man die Klasse „A“ beispielsweise in „B“ um, in UML Lab trägt sie nach der Umbenennung den Namen „C“. Werden beide Modelle anschließend wieder in Verbindung mit dem Synchronisierungs-Plugin eingeladen, kommt ein weiteres Kriterium hinzu. Sollte das Mapping der geänderten Objekte nicht, wie in Abschnitt 4.8 beschrieben, über ein *FTag* persistiert worden sein, werden die beiden Objekte nicht als zueinander passend erkannt. Weil somit kein Mapping besteht und sie durch die unterschiedlichen Klassennamen bei der Suche innerhalb der Modelle ebenfalls nicht als zueinander passend erkannt werden, erfolgt die jeweilige Erstellung eines neuen Partnerobjekts im anderen Modell. Auch wenn das vom Anwender nicht unbedingt gewollt wird, ist dies als naheliegende Lösung anzusehen. Sollte im Gegensatz dazu ein *FTag*-Objekt existieren, werden die beiden automatisch als aktives Mapping erkannt und eine Synchronisierung durchgeführt. Wenn diese das Attribut „name“ erreicht, kann nicht objektiv bestimmt werden, welcher der beiden Werte der bevorzugte sein soll. Aus diesem Grund wird der Wert des zuerst synchronisierten Modells gesetzt. Eine Möglichkeit dieses Problem durch einen manuellen Eingriff anzugehen, wird in Abschnitt 7 angesprochen.

Durch das Round-Trip-Engineering in UML Lab können vereinzelt Szenarien entstehen, die das Fujaba-Modell durch die Synchronisierung in einen inkonsistenten Zustand versetzen. Im Folgenden wird das Zustandekommen anhand eines Beispiels erklärt. Weil diese

Beschränkung unter anderem bei Assoziationen auftritt, genügen zur Darlegung zwei im Quellcode vorliegende Klassen, die mit einer bidirektionalen Assoziation verbunden sind. In diesem Beispiel handelt es sich um *ProductCategory* und *Webshop*.

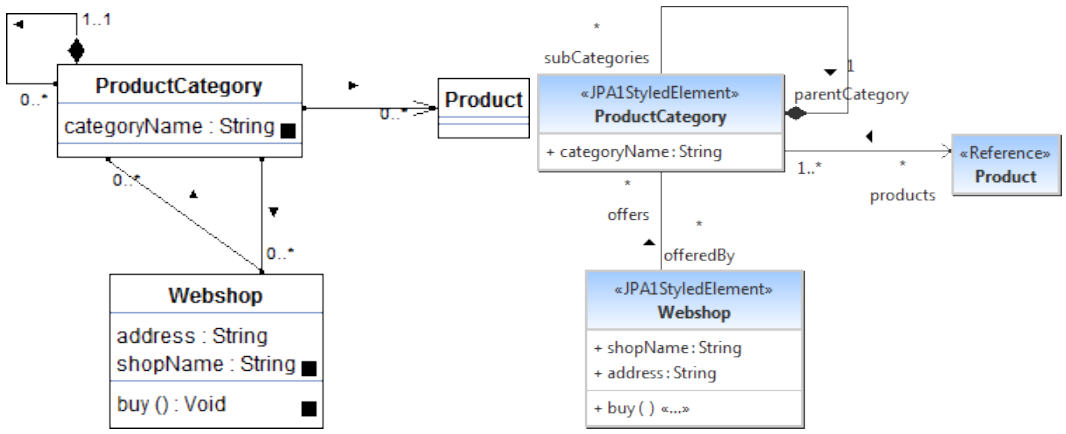
Wird *ProductCategory* durch das Reverse-Engineering in UML Lab eingelesen und mit




(a) Fujaba (b) UML Lab

**Abbildung 6.1:** Zustand der Modelle nach Reverse-Engineering von *ProductCategory*

dem Fujaba-Modell synchronisiert, resultieren die beiden in Abbildung 6.1 gezeigten Zustände. Weil nur eine Klasse eingelesen wird, ist die zweite Klasse nur als Referenz und die Assoziation unidirektional aufgeführt. Wird *Webshop* auf dem gleichen Weg zu den Model-




(a) Fujaba (b) UML Lab

**Abbildung 6.2:** Zustand der Modelle nach Reverse-Engineering von *Webshop*

len hinzugefügt, ergibt sich das in Abbildung 6.2 dargestellte Resultat. Die zweite Klasse ist vollständig analysiert und in UML Lab ist aus der unidirektionalen Referenz eine bidirektionale Assoziation geworden. In Fujaba zeigen sich hingegen zwei Assoziationen, die in *WebShop* zusammentreffen. Der Grund für diese außergewöhnliche Struktur liegt in der

Schwierigkeit aus dem Modell gelöschte Objekte zu identifizieren. Während dem Reverse-Engineering von *WebShop* werden einige Objekte der Assoziation ausgetauscht. Diese entfernten Elemente werden aber nicht vollständig gelöscht, sondern für einen Undo/Redo-Mechanismus intern vorgehalten. Auf Fujaba-Seite ist mit den bisher zur Verfügung stehenden Möglichkeiten ein sicheres Entdecken solcher entfernten Objekte nicht effizient möglich, solange sie nicht vollständig gelöscht werden. Daher verbleiben in diesem Modell die alten und neuen Elemente der Assoziation und resultieren in der Struktur aus Abbildung 6.2(a).

Wie diese Beschränkung zukünftig aufgelöst werden kann, wird in Abschnitt 7 beschrieben. Ansonsten sollte, soweit wie möglich, auf ein partielles Einlesen von Klassenverbänden verzichtet werden.

## 6.2 Auswertung/Bewertung der Umsetzung

Um die Korrektheit der Synchronisierungen zu evaluieren können sie entweder manuell oder automatisch überprüft werden. Dazu gehört sowohl die Synchronisierung eines einzelnen Attributs, als auch eines vollständigen Objektes oder einer ganzen Objektwelt. Durch die zahllosen, verschiedenen Szenarien, für die eine Überprüfung denkbar ist, kann keine einhundertprozentige Abdeckung erreicht werden. Für eine jederzeit fehlerfreie Synchronisation kann also nicht garantiert werden. Dennoch war es das Ziel möglichst viele Szenarien auf die Korrektheit hin zu testen.

Unter die manuelle Überprüfung fällt beispielsweise der Vergleich der Modelle während der normalen Benutzung oder die Erzeugung bestimmter Modellstrukturen, die möglichweise anfällig sein können. Für eine sinnvolle Evaluierung ist dieses Vorgehen also nicht geeignet, sondern dient eher als Möglichkeit falsche oder unvollständige Synchronisierungen zu erkennen.

Eine automatische Überprüfung der Synchronisierungsergebnisse ist bei einer Aufgabe, die soviele potentielle Fehlerquellen bietet, unumgänglich. Selbst kleine Änderungen in der Verarbeitungslogik können große Auswirkungen haben und an vielen Stellen zu falschen Ergebnissen führen. Diese fallen bei einer manuellen Prüfung möglicherweise nicht sofort auf und erschweren im späteren Verlauf die Identifizierung der Fehlerquelle. Daher ist eine größtmögliche Abdeckung durch automatisierte Überprüfungen als Ziel gesetzt worden. Für dessen Erreichung wurden zwei verschiedene Strategien verfolgt: Unit-Tests und die Möglichkeit zum Vergleich vollständiger Modelle.

Alle Unit-Tests sind auf zwei Bereiche ausgerichtet. Erstens dienen sie der Verifizierung der Synchronisierungslogik bei einer schrittweisen Änderung von Attributen. Zum Zweiten stellen sie sicher, dass die Suche nach und anschließende Synchronisierung mit dem Partnerobjekt im anderen Modell fehlerfrei verläuft. Alle Tests sind stets vor oder in wenigen Fällen zumindest parallel zur Implementierung der getesteten Logik entstanden. Somit war von Beginn an eine Qualitätssicherung möglich und bei Änderungen hervorgerufene Fehler konnten sofort erkannt und zeitnah behoben werden. Obwohl mit der Entwicklung der Tests vor der Implementierung der getesteten Logik begonnen wurde, ist dies keine testgetriebene Entwicklung nach [2], weil die Tests nicht dem Ziel bzw. der Struktur der Implementation einen festen Rahmen vorgeben, sondern lediglich der Verifizierung einzelner Bereiche dienen. Ungeachtet dessen sind die Tests essentiell wichtig, um bei den unzähligen Fehlerquellen, die bei einer Modellsynchronisation auftreten können, die Übersicht zu behalten. Dazu wird mit insgesamt über 200 separaten Tests beigetragen. Für die Entwicklung der Unit-Tests wurde auf das Test-Framework junit[9] zurückgegriffen.

Durch Tests im kleinen Maßstab können bei weitem nicht alle denkbaren Szenarien abgedeckt werden. Daher wurde zusätzlich eine Vergleichsmöglichkeit für Modelle im Ganzen eingeführt. Somit kann mit beliebigen, möglicherweise bereits existierenden Modellen eine Synchronisation durchgeführt und das Ergebnis auf dessen Korrektheit überprüft werden. Auf diese Weise ist zum einen keine zusätzliche Arbeit für Erstellung weiterer Testfälle nötig und zum anderen können viele weitere Objektstrukturen sehr einfach zur Überprüfung des Synchronisationsmechanismus herangezogen werden. Die Vergleichsoperationen sind sehr gradlinig aufgebaut. Die beiden zu testenden Modelle werden an sie übergeben und in ihren einzelnen Elementen verglichen. Dazu werden beispielsweise alle *FPackage*-Objekte aus dem Fujaba-Modell nacheinander verifiziert, indem für jedes nach einem passenden Gegenstück im UML2-Modell geschaut wird, bevor anschließend die Attribute der beiden Objekte verglichen werden. Für jeden entdeckten Unterschied wird ein Objekt mit einer Fehlerbeschreibung gesammelt. Diese enthält Informationen darüber, welche Objekte betroffen sind und welcher Vergleich aus welchem Grund fehlgeschlagen ist. Zum Abschluss können durch die zusammengetragenen Unterschiede möglicherweise bisher übersehene, aber problematische Objektstrukturen offenbart werden. Das größte für diese Überprüfung genutzte Projekt ist ein Modell aus der Automobilindustrie mit 247 Klassen.

# 7 Ausblick

Für die zukünftige Weiterentwicklung gibt es mehrere denkbare Ansatzpunkte, die sich allerdings in ihrem Nutzen und somit in der Priorisierung unterscheiden. Teilweise sind sie zusätzlich von der Erweiterung der API von UML Lab abhängig.

In Abschnitt 6.1 wird auf die Problematik bei der Synchronisierung neu geladener Projekte in Bezug auf die Priorisierung bei der Änderung gleicher Attribute eingegangen. Die automatisierte Lösung ist als Kompromiss zu sehen und kann nur durch eine manuelle Entscheidung verbessert werden. Dazu müssen alle konfliktbehafteten Attribute erkannt und gesammelt werden, bevor sie dem Anwender vorgelegt werden können. Dieser muss entscheiden, ob der Attributwert des Objektes aus dem Fujaba-Modell oder des Objektes aus dem UML Lab-Model übernommen werden soll. Für eine qualifizierte Entscheidung ist eine übersichtliche und leicht verständliche Darstellung aller nötigen Informationen unumgänglich. Beispielsweise können die Unterschiede zwischen den zwei fraglichen Elementen gezeigt werden, um dem Anwender in der Entscheidung zu unterstützen. Dabei ist für die Darstellung eine rein textuelle oder die grafische Abbildung aller betroffenen Objekte denkbar. Um zu entscheiden, welche Priorität eine solche Erweiterung hat, muss das generelle Anwendungsgebiet betrachtet werden. Im normalen Gebrauch ist davon auszugehen, dass stets mit aktiver Synchronisierung gearbeitet wird, so dass dies als Funktionalität angesehen werden kann, um Sonderfälle abzudecken.

In Abschnitt 6.1 wurde die Limitierung betreffend das partielle Reverse-Engineering von Klassenverbänden ausführlich erläutert. Ohne grundlegende Änderung in der Eventverarbeitung in Bezug auf das UML2-Metamodel ist diese Problematik nicht effizient aufzulösen. Statt sich mittels eines Listeners an jedem Objekt im Modell einzeln anzumelden und auf diesem Weg Zugriff auf alle erzeugten Events zu erhalten, gibt es die Möglichkeit einen *ResourceSetListener* einzusetzen. Dieser kann beispielsweise auf ein *ResourceSet*, also den Container aller zusammengehörigen Modelle (das Modell des Anwenders, die Modelle für die primitiven Typen, etc.), angewendet werden. Über diesen werden keine einzelnen, sondern mehrere in einer Transaktion gebündelte Events verschickt. Diese können anschließend

separat voneinander die Verarbeitungsschritte durchlaufen. Dadurch, dass eine Transaktion in sich abgeschlossen ist, können sogar zusätzliche Informationen aus den Events herausgelesen werden. Befindet sich beispielsweise am Ende einer Transaktion ein Objekt nicht in einem Elternobjekt, wird es nicht mehr als dem Modell zugehörig interpretiert. Somit ist es nicht nötig auf die endgültige Löschung zu warten, sondern es kann bereits zu diesem Zeitpunkt aus dem Fujaba-Modell entfernt werden. Durch die Bündelung der Events entstehen automatisch weitere Vorteile. Sind mehrere auf ein einziges Objekt bezogene Änderungen vorhanden, wurden diese bereits auf das Objekt angewendet, d.h. alle Attribute besitzen bereits den neuen Wert. Somit wird eine bessere Möglichkeit zur Identifizierung des zuständigen Handlers für die Weiterverarbeitung geboten. Beispielsweise gelangt die Zuordnung von *Property* auf *FAttr* bzw. *FRole* häufiger ohne Umwege über *FAttr* direkt zu einer *FRole*, weil die strengeren Kriterien für diesen Handler früher erfüllt werden. Schließlich werden durch diese Änderung auch die *WeakReference*s aus UML2-Sicht in der *BidiWeakMap* aus Abschnitt 4.3.3 obsolet und vereinfachen das Mapping zusätzlich. Sie können sogar komplett entfernt werden, weil das Fujaba-Metamodell ein „RemoveYou“-Event bereitstellt, welches ausschließlich bei der Löschung eines Objektes gesendet wird und somit für die nachfolgende Entfernung des Partnerobjektes in UML2 genutzt werden kann. Wegen den resultierenden Verbesserungen in mehreren Bereichen der Synchronisierung, ist diese Anpassung mit einer hohen Priorität zu versehen.

Auf Grund einer solchen Anpassung sind zukünftig noch weitere Erweiterungen denkbar. Weil die in Fujaba für die Persistenz, und somit die Protokollierung aller Änderungen in den Objekten, zuständige Bibliothek CoObRa[15] ebenfalls mit Transaktionen arbeitet, können diese mit den Transaktionen aus EMF abgeglichen werden. Dadurch ist beispielsweise ein Undo/Redo über eine Transaktion ebenfalls synchronisierbar.

Durch eine Ausweitung der API in UML Lab kann die Einbindung des Plugins schrittweise verbessert werden. So ist beispielsweise die Kombination der Codegenerierungen aus Fujaba und UML Lab noch umständlich und erfordert ein einzelnes Generieren für jedes der beiden Modelle. Wenn es in Zukunft möglich ist sich in die Codegenerierung von UML Lab einzuhängen, kann die zusätzliche Generierung für Story-Diagramme automatisch in den Prozess eingebunden werden, was eine Erhöhung Bedienkomforts bedeutet.

Zudem ist es denkbar den Umfang des synchronisierten Modells in Zukunft zu erweitern. Zurzeit werden nur Teile der beiden Metamodelle synchronisiert, die sich auf Klassendiagramme beziehen, weil es sich dabei um den aktuellen Anwendungsbereich von UML Lab handelt. Wenn durch zusätzliche Funktionalitäten der verwendete Bereich der UML2-Spezifikation ausgeweitet wird, kann die Synchronisierung ebenfalls entsprechend erwei-

tert werden. Grundvoraussetzung ist allerdings, dass eine vergleichbare Struktur im Fujaba-Metamodell zu finden ist. Solange keine solche Erweiterung stattfindet, sind aber keine Anpassungen in diesem Bereich nötig.

Des Weiteren kann die Synchronisierung auch von der Metamodell-Ebene auf die GUI-Ebene erweitert werden, um beispielsweise Diagramme in ihrer Darstellung abzugleichen. So könnte die Positionierung zueinander gehörender Elemente in beiden Darstellung aufeinander abgestimmt werden. Alternativ kann auch der Inhalt des angezeigten Modellausschnittes in den Klassendiagrammen synchronisiert werden. Durch diese Maßnahmen wird die Übersichtlichkeit bei der parallelen Bearbeitung der beiden Modelle erhöht und das manuelle Anpassen oder Suchen und Umpositionieren von Klassen zur besseren Vergleichbarkeit kann eingespart werden. Die parallele Bearbeitung inklusive paralleler Darstellung der beiden Klassendiagramme ist allerdings bei der normalen Anwendung eher ungewöhnlich, so dass dies als kosmetische und somit niedrig zu priorisierende Verbesserung anzusehen ist.

# Abbildungsverzeichnis

# Literaturverzeichnis


[1] ATL: *ATL Eclipse plugin project*. http://www.eclipse.org/atl/. Version: 2010

[2] BECK, Kent: *Test Driven Development: By Example*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0321146530

[3] BUNEMAN, Peter ; FERNANDEZ, Mary ; SUCIU, Dan: UnQL: a query language and algebra for semistructured data based on structural recursion. In: *The VLDB Journal* 9 (2000), Nr. 1, S. 76–110. http://dx.doi.org/http://dx.doi.org/10.1007/s007780050084. – DOI http://dx.doi.org/10.1007/s007780050084. – ISSN 1066–8888

[4] E. GAMMA, R. J. R. Helm H. R. Helm ; VLISSIDES, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. – ISBN 0201633612

[5] GEIGER, Leif ; SCHNEIDER, Christian ; RECKORD, Carsten: Template- and modelbased code generation for MDA-Tools. In: *3rd International Fujaba Days*. Paderborn, Germany, September 2005

[6] GIESE, Holger ; HILDEBRANDT, Stephan: Efficient Model Synchronization of Large-Scale Models / Hasso Plattner Institute at the University of Potsdam. 2009 (28). – Forschungsbericht. – ISBN 978–3–940793–84–3

[7] GREAT: *Homepage des GReAT-Projektes*. http://www.escherinstitute.org/Plone/tools/suites/mic/great. Version: 2010

[8] HIDAKA, Soichiro ; HU, Zhenjiang ; INABA, Kazuhiro ; KATO, Hiroyuki ; MATSUDA, Kazutaka ; NAKANO, Keisuke: Bidirectional Graph Trasformations / Center for global research in advanced software science and engineering. 2010. – Forschungsbericht

[9] JUNIT: *Homepage von JUnit*. http://www.junit.org. Version: 2010

[10] KINDLER, Ekkart ; WAGNER, Robert: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios / University of Paderborn. 2007. – Forschungsbericht

[11] OMG: OMG Unified Modeling Language (OMG UML) Infrastructure Version 1.4. Version: 2001. http://www.omg.org/spec/UML/1.4/Infrastructure/PDF. 2001 (formal/2001-09-67). – Forschungsbericht

[12] OMG: *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Object Modeling Group, April 2008. (OMG document formal/2008-04-03) . http://fparreiras/papers/mof_qvt_final.pdf

[13] OMG: OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.2. Version: 2009. http://www.omg.org/spec/UML/2.2/Infrastructure/PDF. 2009 (formal/2009-02-04). – Forschungsbericht

[14] REENSKAUG, Trygve: *MVC*. http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[15] SCHNEIDER, Christian: *CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten*, Diss., 2007. http://kobra.bibliothek.uni-kassel.de/handle/urn:nbn:de:hebis:34-2007121319874

[16] SCHÜRR, Andy: Specification of Graph Translators with Triple Graph Grammars. In: *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*. London, UK : Springer-Verlag, 1995. – ISBN 3–540–59071–4, S. 151–163

[17] VIATRA: *VIATRA Eclipse plugin project*. http://eclipse.org/gmt/VIATRA2/. Version: 2010

[18] WAGNER, Robert: *Inkrementelle Modellsynchronisation*, Diss., 2009. http://ubdok.uni-paderborn.de/servlets/DerivateServlet/Derivate-11439/dissertation.pdf

[19] XIONG, Yingfei ; LIU, Dongxi ; HU, Zhenjiang ; ZHAO, Haiyan ; TAKEICHI, Masato ; MEI, Hong: Towards automatic model synchronization from model transformations. In: *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–882–4, S. 164–173

[20] ZÜNDORF, A.: *Rigorous Object Oriented Software Development*. Habilitation Thesis, University of Paderborn, 2001

**Ehrenwörtliche Erklärung**

Ich versichere, dass ich die beiliegende Masterarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher Form noch keiner Prüfungsbehörde vorgelegen. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

**Ort, Datum** **Unterschrift**