# Modellgetriebene Entwicklung eines erweiterten UML-Editors zur Modellierung von ObjectTeams/Java-Programmen

Diplomarbeit

von

Andreas Werner

Matrikelnr. 192206

13. Dezember 2007

| | |
|---|---|
| Gutachter: | Prof. Dr.-Ing. Stefan Jähnichen<br>Dr.-Ing. Stephan Herrmann |
| Betreuer: | Dipl-Inf. Marco Mosconi |




Technische Universität Berlin

Institut für Softwaretechnik und Theoretische Informatik

Fachgebiet für Softwaretechnik

## Eidesstattliche Erklärung

Die selbständige und eigenhändige Anfertigung dieser Diplomarbeit versichere ich an Eides Statt.

Berlin, 13. Dezember 2007

# Inhaltsverzeichnis

# 1 Einleitung

Eines der größten Probleme langlebiger Softwaresysteme ist die Anpassbarkeit an sich verändernde Nutzungsanforderungen und Umgebungsbedingungen. Und die Wahrscheinlichkeit für eine notwendige Anpassung ist groß. So erscheinen in kurzen Abständen immer wieder neue Technologien, die z.B. bessere Integration oder höhere Performance versprechen. Auch Hardware, Betriebssysteme und nicht zuletzt die Kommunikation zwischen Maschinen und Anwendungen entwickeln sich weiter. Hinzu kommt noch, dass sich die funktionalen Anforderungen eines Softwaresystems ebenfalls ändern können. Aus diesen Gegebenheiten folgt, dass Software jederzeit und ohne großen Aufwand oder Kosten an neue Anforderungen anpassbar sein sollte. Leider ist diese Vorstellung heutzutage noch nicht zu verwirklichen.

Es gibt aber verschiedene Ansätze, die eine bessere Anpassbarkeit von Softwaresystemen versuchen umzusetzen. Die *Object Management Group* (*OMG*[25]), eine Vereinigung von mehreren IT-Firmen, schlägt zur Lösung dieser Problematik die *Model Driven Architecture* (*MDA*[24]) vor. Die Vision [23], welche die OMG in Zusammenhang mit der Umsetzung der MDA verbindet, zielt in erster Linie auf die Trennung des Entwurfs einer Software in einen rein funktionalen Entwurf und einen Entwurf, der ebenso Implementierungsdetails enthält, ab. Modelle spielen in der MDA eine strategische Rolle. Daher wird der erstgenannte Entwurf auch als *Platform Independent Model* (*PIM*) und der zweitgenannte als *Platform Specific Model* (*PSM*) bezeichnet. Ein PIM ist immer unabhängig von irgendwelchen Technologien definiert und spiegelt allein die funktionalen Anforderungen wider. Im PSM versucht man dagegen, das PIM auf eine konkrete Umgebung abzubilden. Der Vorteil der Aufteilung in PIM und PSM besteht darin, dass sich Änderungsanforderungen im besten Fall nur noch auf eine Ebene des Entwurfs beziehen. Zum Beispiel wäre im Fall des Austauschs der verwendeten Programmiersprache *nur* das PSM zu ändern, da die Funktionalität des Systems hier identisch bliebe. Aus der Annahme, dass ein PIM von den existierenden Technologien abstrahieren soll, folgt, dass die einem PIM zugrundeliegende Modellierungssprache selbst universell sein muss. Die Modellierungssprache eines PSM dagegen muss eine oder mehrere Technologien berücksichtigen, die im PSM ausgedrückt werden sollen. Da mehrere Technologien oder auch Plattformen existieren (Komponentenarchitekturen, Programmiersprachen, etc.), existieren zwangsläufig mehrere Modellierungssprachen für PSMs.

Damit die Abhängigkeiten zwischen allen Modellen nachvollziehbar und nachverfolgbar bleiben, werden Modelltransformationen durch sogenannte Mappings definiert. Model-Mappings sind in der MDA neben den eigentlichen Modellen ein weiteres Schlüsselkonzept. Die Transformationen von einem Modell in ein anderes werden heutzutage teilweise mit Generatoren durchgeführt. Diese be-

kommen als Eingabe ein oder mehrere Modelle sowie eine Abbildungsvorschrift (das Mapping) und erzeugen als Ausgabe ein neues Modell. Beispiele für solche Transformationen sind etwa PIM nach PSM, PSM nach PSM oder sogar PSM nach Code, bzw. PIM nach Code, da Code im Prinzip auch nichts anderes als ein Modell ist. Das oben genannte Beispiel zur Änderung der Programmiersprache für ein Softwaresystem könnte etwa mit Hilfe von Generatoren ganz automatisch und ohne größeren Aufwand durchgeführt werden. Dazu wird zuerst ein neues PSM aus dem vorher schon vorhanden PIM generiert und aus dem entstehenden PSM dann der Code. Diese Idealvorstellung der MDA ist in der Praxis aber zur Zeit nicht so einfach anwendbar. Es fehlt an Generatoren, Mappings und sogenannten Plattformmodellen, die die Eigenschaften von Zielplattformen beschreiben. Diese Bausteine müssen für die meisten Anwendungen erst erstellt werden. Aber es gibt auch schon eine Vielzahl an Code-Generatoren, die zumindest Teile von Modellen in ausführbaren Code transformieren können.[1]

Im Rahmen dieser Diplomarbeit soll nun ein Grundstein für die am betreuenden Fachgebiet entwickelte Sprache *ObjectTeams/Java* (*OT/J*[15]) im Kontext der MDA gelegt werden. Dafür wird eine Modellierungssprache entwickelt, welche ObjectTeams/Java-Konzepte unterstützt sowie ein grafischer Editor für die daraus erstellbaren PSMs. Die Modellierungssprache soll *ObjectTeams Modeling Language* (*OTML*) heißen, in Anlehnung an die zurzeit bekannteste Modellierungssprache *Unified Modeling Language* (*UML*[26]). Der Editor heißt *ObjectTeams Modeler* (kurz: *OTModeler*). Mit Hilfe des OTModelers und den mit ihm erzeugbaren Modellen könnten später direkt OT/J-Anwendungen generiert werden. Ebensogut eignen sich die Modelle auch einfach zu Dokumentationszwecken. Dabei gewährt die OTML im Rahmen dieser Diplomarbeit lediglich eine statische Sicht auf eine Anwendung, genauso wie die Klassendiagramme aus der UML.

## 1.1 Zielstellung

Durch die thematische Eingliederung der Diplomarbeit in den Kontext der MDA in Zusammenhang mit ObjectTeams/Java ergeben sich für die Zielstellung drei wichtige Randbedingungen.

**UML als Basis zur Definition der OTML.** Wie im vorhergehenden Abschnitt kurz beschrieben wurde, sind Modelle ein Schlüsselkonzept der MDA. Dabei ist zurzeit die bekannteste Modellierungssprache die Unified Modeling Language. Sie stellt gewissermaßen den Standard zur Modellierung von objektorientierter Software dar, ist dabei aber nicht auf eine bestimmte Programmiersprache festgelegt, weswegen sie als Sprache für PIMs gut geeignet ist. Daneben kann sie ebenfalls zur Erstellung von PSMs genutzt werden, wenn man die Einschränkungen der Zielplattform bei der Modellierung berücksichtigt. Ein Beispiel für eine Einschränkung ist die in vielen geläufigen Programmiersprachen vorkommende Einfachvererbung, wohingegen die UML Mehrfachvererbung unterstützt. Andererseits kann sie erwartungsgemäß nicht alle Konzepte unterstützen. Deshalb

[1] In der praktischen Umsetzung dieser Diplomarbeit wird ein solcher Generator verwendet.

existieren viele Erweiterungen der UML, zumeist als Profile, die für bestimmte Domänen wichtige Ergänzungen vornehmen. Beispiele für solche Erweiterungen sind u.a. das *UML Testing Profile* oder das *UML Profile for Systems Engineering* (*SysML*).[2]

Auch die mit ObjectTeams/Java ausdrückbaren Programme enthalten Konzepte, welche nicht unterstützt werden. Deswegen ist eine Erweiterung der UML um die von ObjectTeams/Java bereitgestellten Sprachkonstrukte notwendig. Aufgrund der hohen Verbreitung und Unterstützung der UML soll die OTML aber möglichst konform zu ihr bleiben. Dies kann durch eine rein additive Erweiterung ohne Veränderung der ursprünglichen UML-Anteile erreicht werden. Als Folge wäre es weiterhin möglich UML-Modelle mit der OTML zu erstellen oder vorhandene UML-Modelle mit OT/J-Konzepten anzureichern.

**Wiederverwendung von vorhandenen grafischen UML-Editoren.** Durch die Verwendung der UML als Grundlage der OTML ergibt sich die Notwendigkeit nicht nur OT/J-Konzepte im OTModeler zu beschreiben, sondern auch UML-Konzepte. Um UML-Klassendiagramme zu erstellen, existieren bereits viele grafische Editoren. Dieser Umstand soll für die Erstellung des OTModelers ausgenutzt werden. Eine Untersuchung vorhandener Editoren soll die Möglichkeiten zur Wiederverwendung überprüfen. Damit würde sich der Implementierungsaufwand erheblich reduzieren.

**OTModeler als Eclipse-Plugin.** Der OTModeler muss als Eclipse-Plugin implementiert werden. Diese Randbedingung resultiert daraus, dass für ObjectTeams/Java bereits eine komplette Entwicklungsumgebung (IDE) auf Grundlage der Eclipse-Plattform existiert. Diese wird als *ObjectTeams Development Tooling* (*OTDT*) bezeichnet. Weiterhin folgt, dass der Editor, der vom OTModeler wiederverwendet werden soll, ebenfalls als Eclipse-Plugin vorliegen muss.

Die bis hierher genannten Ziele und Randbedingungen sollen durch die folgenden Stichpunkte nochmal kurz zusammengefasst werden:

1. Erarbeitung einer Modellierungssprache für OT/J-Anwendungen
2. Größtmögliche Ausdruckskraft der OTML-Modelle hinsichtlich OT/J
3. Einfache Bearbeitung der OTML-Modelle durch grafischen Editor (OTModeler)
4. Wiederverwendung vorhandener Editoren zur Aufwandsreduzierung
5. Hohe Konformität zur UML speziell zu den UML-Klassendiagrammen
6. Austauschbarkeit von UML/OTML-Modellen mit anderen Modellierungswerkzeugen

[2] Die aufgeführten Profile sind der UML Homepage[26] entnommen.

## 1.2 Aufbau der Diplomarbeit

Im nächsten Kapitel wird auf die allgemeinen Grundlagen genauer eingegangen. Dazu zählen ObjectTeams/Java sowie die Eclipse-Plattform. Weitere wichtige Grundlagen, die in engem Zusammenhang mit der Umsetzung der Arbeit stehen, werden dann in Abschnitten des Hauptteils erläutert, bspw. die UML und einige Eclipse-Plugins.

Der Hauptteil besteht aus zwei Kapiteln, die sich mit der Bearbeitung der beschriebenen Aufgaben beschäftigen. Das erste Kapitel geht dabei auf den Entwurf und die Implementierung der OTML ein. Im zweiten Kapitel wird dann die Erstellung des OTModelers besprochen.

Das letzte Kapitel bildet den Schluss dieser Diplomarbeit und fasst nochmal alle wichtigen Ergebnisse zusammen. Außerdem wird noch eine Bewertung der Vorgehensweise und ein Ausblick auf weiterführende Aufgaben gegeben.

# 2 Allgemeine Grundlagen

## 2.1 ObjectTeams/Java

Die aspektorientierte und Rollen-basierte Sprache ObjectTeams/Java bildet die Basis für diese Diplomarbeit. Deswegen soll an dieser Stelle kurz auf ihre Konzepte und deren Bedeutung eingegangen werden. Die Beschreibung dieser Konzepte ist nicht nur wichtig für die Entwicklung der Modellierungssprache OTML, sondern kann auch als deren semantischer Teil begriffen werden. Die hier wiedergegebenen Erläuterungen stammen aus verschiedenen Quellen von der Projektseite ([15]), insbesondere aus [16] und [17].

Objektorientierte Sprachen bieten in erster Linie das Konzept der Klasse zur Strukturierung von Programmen an. Sie sind Module, die sowohl Daten als auch Verhalten von Objekten klassifizieren. Daten und Verhalten können dabei über die Generalisierungsbeziehung wiederverwendet werden. Das Problem, das bei dieser nur eindimensionalen Strukturierung von Programmen entstehen kann, besteht darin, dass nur ein Teil des Verhaltens so strukturiert werden kann. Andere Teile werden hingegen über mehrere Klassen verteilt und sind dadurch nicht mehr modularisiert. Dies wird in der Aspektorientierung auch als *Crosscutting Concerns* bezeichnet.

**Teams und Rollen.** ObjectTeams/Java versucht die Crosscutting Concerns durch die Einführung eines neuen Modul-Begriffs zu strukturieren, den sogenannten *Object Teams* oder einfach nur *Teams*. Dabei bilden sie ein Modul für die sogenannten *Rollen*, welche die Concerns in den Klassen referenzieren und damit diese im Kontext eines Teams repräsentieren. Die Referenz zwischen Rolle und Klasse, in diesem Zusammenhang auch als Basis-Klasse oder Basis bezeichnet, wird mit aspektorientierten Konzepten umgesetzt. In OT/J sind Klassen technisch gesehen normale Java-Klassen, deren interne Klassen die Rollen sind. Rollen müssen dabei immer in einem Team deklariert werden.

**Rollen-Bindung.** Die Rollen-Bindung wird über die sogenannte PlayedBy-Relation umgesetzt. Durch diese wird es möglich, dass Referenzen (Bindungen) zwischen den Methoden einer Rolle und den Attributen und Methoden ihrer Basis-Klasse definiert werden können. Diese Bindungen heißen entweder Callout- oder Callin-Bindung, je nachdem in welcher Richtung der Kontrollfluss von der Rolle zur Basis verläuft. Bspw. können Methoden der Basis-Klasse über eine Rollenmethode aufgerufen werden. Die entsprechende Bindung wird dann als sogenanntes *Callout Method Binding* bezeichnet. Weiterhin existieren noch das *Callin Method Binding* und das *Callout Field Binding*, das den lesenden und schreibenden Zugriff auf ein Basis-Klassen-Attribut über eine Rollenmethode definiert.

Das Callin Binding kann dazu benutzt werden das Verhalten der Basis-Klasse durch die Rolle zu verändern. Das geschieht nicht-invasiv. Die Basis-Klasse „weiß“ also nichts über ihre Adaptierung. Wenn in diesem Fall eine Methode der Basis-Klasse aufgerufen wird, so wird über das Callin Method Binding der Kontrollfluss zu der gebundenen Rollenmethode umgeleitet. Diese kann das ursprüngliche Verhalten ergänzen oder komplett ersetzen. Außerdem wird für das Callin Binding noch das Konzept der Team-Aktivierung eingeführt (Team Activation). Da eine Rolle stets nur im Kontext einer Team-Instanz ausgeführt werden kann, muss beim Callin Binding eine solche automatisch erstellt werden. Ob die Aktivierung eines Teams erfolgt, kann durch sogenannte Guards auf verschiedenen Ebenen gesteuert werden. Guards sind dabei boolesche Ausdrücke und enthalten Bedingungen für die Aktivierung des Teams.

## 2.2 Die Eclipse-Plattform und OT/Equinox

Eclipse (siehe [6]) ist in der Regel als Entwicklungsumgebung für die Sprache Java bekannt. Darüberhinaus ist sie aber eine wiederverwendbare Plattform zur Entwicklung von Anwendungen. Sie basiert auf einer Implementierung des Komponentenframeworks OSGi, die Equinox genannt wird. Beispiele für bekannte Komponenten von Eclipse, die hier Plugins heißen, sind etwa die *Java Development Tools* (JDT) oder das weiter unten beschriebene *Eclipse Modeling Framework*. in denen gleichartige Funktionalität gebündelt wird. Plugins können Beziehungen zu anderen Plugins besitzen und somit deren bereitgestellte Funktionalität benutzen. Dies wird in dem *Manifest* eines Plugins deklariert. Speziell können Plugins Packages exportieren, also für andere Plugins sichtbar machen, welche wiederum die darin enthaltenen Klassen benutzen können. Zuvor muss ein Plugin eine Import-Beziehung zu dem referenzierten Plugin in dem Manifest deklarieren. Diese Referenz wird auch als *Plugin Dependency* bezeichnet.

Mit dem *Extension Point*-Mechanismus stellt Eclipse eine andere Art der Beziehung zwischen Plugins zur Verfügung. So können vorhandene Plugins über Extension-Points Stellen angeben, an denen sie in ihrer Funktionalität erweitert werden können. Dazu muss ein anderes Plugin, das eine Dependency zum ersten besitzt, für einen konkreten Extension-Point eine zugehörige Extension definieren. Das geschieht in der Datei *plugin.xml* in Form von XML-Einträgen. Die Einträge verweisen meist auf Klassen, die eine bestimmte Schnittstelle implementieren und so die eigentliche Erweiterung darstellen. Die deklarierten Klassen werden dann zur Laufzeit über Registry-Klassen vom ursprünglichen Plugin mitberücksichtigt.

Im Rahmen von ObjectTeams/Java existiert eine Erweiterung des Eclipse-Plugin-Konzeptes. Diese wird als OT/Equinox bezeichnet und ermöglicht eine Beziehung zwischen Rolle und Basis über Plugin-Grenzen hinweg. Auf diese Weise können dann vorhandene Eclipse-Plugins mit Hilfe von OT/J adaptiert werden. Ein Beispiel für die Anwendung stellt die Adaptierung des Eclipse-Kerns und der JDT dar, welche die Erstellung und Ausführung von ObjectTeams/Java-Programmen mit Hilfe von Eclipse ermöglicht (siehe [18]).

Die Adaptierung eines Plugins, in diesem Kontext *Base Plugin* genannt, durch ein *Aspect Plugin* wird über ein *Aspect Binding* in einem Extension-Point in der *plugin.xml* definiert. Dort wird genaugenommen jedem Team des Aspect-Plugins ein Base-Plugin zugeordnet. Die Rollen eines Teams dürfen dann nur noch an Klassen dieses Plugins gebunden werden. OT/Equinox erlaubt dabei sogar die Adaptierung von Klassen in Packages, die nicht exportiert werden und so normalerweise nicht sichtbar wären.

Durch OT/Equinox ist eine umfassende Änderung oder Erweiterung von Plugins möglich, auch ohne das Vorhandensein von Extension-Points. Bei der Entwicklung eines Plugins können nicht immer die Stellen antizipiert werden, die andere Plugins ändern müssen. So wird mit Hilfe von OT/Equinox und OT/J ein hoher Grad an Wiederverwendung erreicht, der mit dem herkömmlichen Extension- Point-Konzept nicht erreicht werden kann.

Der Extension-Point-Mechanismus von Eclipse sowie die Möglichkeiten der Wiederverwendung und Adaptierung mit Hilfe von OT/Equinox wird bei der Umsetzung des OTModelers noch eine Rolle spielen.

# 3 Die Modellierungssprache OTML

Für den Entwurf der OTML müssen einige Entscheidungen getroffen werden, die auch Auswirkungen auf anschließende Arbeiten haben können. Die OTML sollte möglichst alle statischen Sprachkonzepte von ObjectTeams/Java beinhalten, aber dennoch einfach erweiterbar und veränderbar sein, da OT/J selbst sich auch weiterentwickelt. Schwierig daran ist, den Fokus bei der Auswahl nicht nur auf die hier existierenden Probleme zu begrenzen, sondern andere Bereiche miteinzubeziehen, um eine nachträgliche Erweiterung zu erleichtern. Ein Beispiel wäre die Ergänzung um dynamische Sprachkonstrukte zur Modellierung von Verhalten. Dazu kommt, dass nicht nur die statischen Konzepte modelliert, sondern auch Einschränkungen auf ihnen formuliert werden müssen. Dies geschieht innerhalb von UML-Modellen mit Hilfe der *Object Constraint Language* (*OCL*), die eine textuelle Notation besitzt, im Gegensatz zur größtenteils grafischen Notation der UML. Die Einschränkungen für die OTML ergeben sich zum überwiegenden Teil aus der *ObjectTeams/Java Language Definition* (*OTJLD*[17]) und zum restlichen Teil aus dem Entwurf selbst.

Die OTML beschreibt weiterhin nicht nur welche Sprachelemente von OT/J modellierbar sein sollen und welche Einschränkungen bei der Modellierung von OT/J-Anwendungen gelten, sondern führt weiterhin auch eine grafische Notation für diese Sprachelemente ein. Diese Notation soll dann im OTModeler verwendet werden. Für eine Erweiterung der UML stehen verschiedenene vordefinierte Mechanismen zur Verfügung. Die Auswirkungen der Erweiterungsvarianten müssen untersucht und bewertet werden und letztlich soll eine Variante ausgewählt und mit dieser der tatsächliche Entwurf durchgeführt werden.

Auf den Entwurf folgend muss das OTML-Metamodell noch als Java-Code zur Verfügung gestellt werden, damit der OTModeler dieses benutzen. Im Rahmen der Eclipse-Plattform existieren einige Projekte (siehe Abschnitte 3.1.3 und 3.1.4), die sich speziell mit Modellierung und daruf basierender Quellcode-Generierung beschäftigen.

## 3.1 Grundlagen

### 3.1.1 Modellierung

Modelle dienen im Allgemeinen zur grafischen Darstellung komplexer Sachverhalte, da sie leichter verständlich sind als z.B. Programmcode oder lange textuelle Beschreibungen. Ein Grund dafür ist, dass grafische Modelle üblicherweise in einem Kontext betrachtet werden bzw. eine eingegrenzte Sicht auf eine Problemstellung bieten und dadurch von unnötigen Details abstrahieren. Als Konsequenz sind sie meist auch weniger präzise als textuelle oder formale Beschreibungen.Man benötigt daher mehrere Sichten und mehrere Modelle, wenn

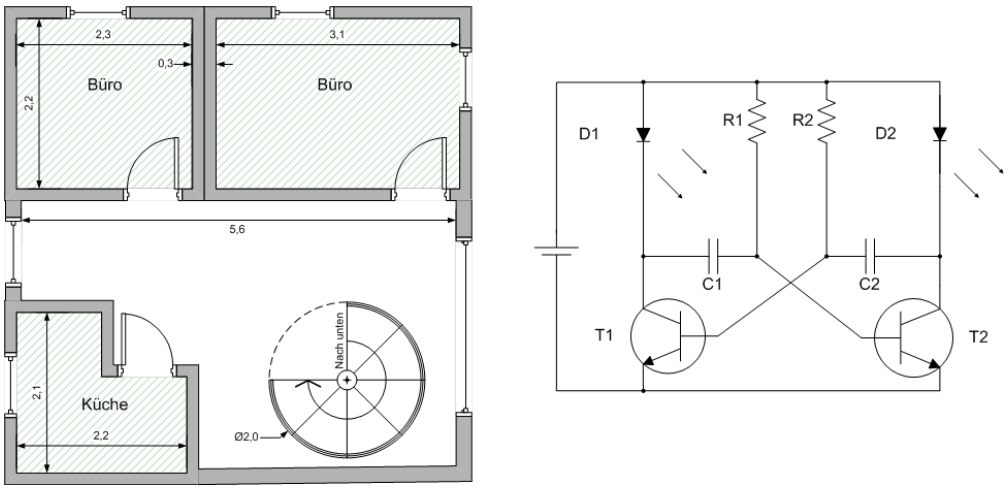


Abbildung 3.1: Bsp. für Modelle: Gebäudegrundriss (li.), elektronische Schaltung (re.)

es darum geht konkrete Instanzen zu erstellen.

Zusätzlich haben die Art der grafischen Darstellung und das Layout einen wesentlichen Einfluss auf die Rezeption. Beispiele für Modelle sind etwa Grundrisszeichnungen in der Architektur, Streckennetzdiagramme im öffentlichen Personennahverkehr oder Schaltungsdiagramme in der Elektrotechnik (siehe Abbildung 3.1). Sie können aber nicht nur Konzepte visualisieren und veranschaulichen, sondern helfen auch dabei, diese in der Praxis umzusetzen. Auf die Beispiele bezogen, bildet der Gebäudegrundriss die Vorlage für ein oder mehrere tatsächlich zu bauende Häuser. Oder der Schaltplan kann für den Aufbau einer realen Schaltung dienen. Im Kontext der hier beschriebenen Modellierung werden Schaltungen und Gebäude auch als *Instanzen* ihrer Modelle bezeichnet. Dabei unterliegen sie immer den *Einschränkungen* des Modells, z.B. müssen die Raummaße eingehalten werden, die Türen müssen in die entsprechende Richtung öffnen, oder die Treppe muss eine Wendeltreppe sein. Die letzte Bedingung lässt sich darauf zurückführen, dass in Modellen durch die Verwendung bestimmter Symbole direkt der Typ derjenigen Instanzen festgelegt ist. Dies wird auch als grafische Notation oder konkrete Syntax bezeichnet. Welche Elemente (Fenster, Wand, etc.) in einem Modell überhaupt vorkommen dürfen und in welcher Beziehung sie stehen (Fenster befindet sich in Wand), ist meistens in einer Modellspezifikation festgehalten, genauer gesagt in deren sogenannter abstrakter Syntax. Außerdem haben Modelle auch immer eine bedeutungsvolle Aussage, die durch die Anordnung und Kombination von Modellelementen abgeleitet werden kann. Diese wiederum haben jeweils eine eigene Bedeutung, die im Kontext einer Modellisierungssprache festgelegt sein sollte, so dass keine unterschiedlichen Interpretationen über ein Modell entstehen können. Diese Festlegung geschieht ebenfall in der Modellspezifikation und wird Semantik genannt.

Heutzutage werden Modelle nicht mehr nur per Hand erstellt, sondern immer häufiger mit Hilfe des Computers. Als Vorteile der computergestützten Model-

lierung gelten die automatische Überprüfung der Korrektheit eines Modells, eine einfache Vervielfältigung und Verteilung, sowie die Ableitung weiterer Modelle. Dazu ist aber eine formale Spezifikation der in Modellen enthaltenen Konzepte erforderlich. Diese Spezifikation wird auch als Metamodell bezeichnet, was wörtlich übersetzt soviel wie Modell über ein Modell bedeutet.

### 3.1.2 Die UML

Die UML ist in dieser Arbeit von grundlegender Bedeutung. So besteht ein Ziel unter anderem darin, dass ein Metamodell für die Sprache ObjectTeams/Java entwickelt werden soll. Da ObjectTeams/Java auf der objektorientierten Sprache Java basiert, bzw. eine Ergänzung dieser darstellt, und die UML ein Metamodell zur Beschreibung von objektorientierten Modellen ist, würde die Wiederverwendung der UML erheblich zur Minimierung des Erstellungsaufwands beitragen. So wie Java durch ObjectTeams/Java um neue Konzepte ergänzt wurde, ließe sich auch die UML um diese neuen Konzepte erweitern. Aus diesem Grund werden in den folgenden Unterabschnitten nähere Details zum Aufbau und den verfügbaren Erweiterungsstrategien der UML erläutert.

Die UML, aktuell ist die Version 2.0, stellt generell eine grafische Sprache zur Visualisierung, Spezifikation und Dokumentation von Softwaresystemen dar. Sie abstrahiert dabei von konkreten Programmiersprachen und kann als universelle Modellierungssprache für objektorientierte Systeme genutzt werden. Dadurch bildet sie ein einheitliches Vokabular für Entwickler und Architekten. Neben Sprachkonzepten wie Klassen, Interfaces, Assoziationen usw., die in den meisten objektorientierten Sprachen vorkommen, werden auch verschiedenen Sichten auf ein Softwaresystem definiert. Diese Sichten werden Diagramme genannt, wobei außer dem Klassendiagramm als dem wohl populärsten weitere Diagrammarten vorhanden sind, etwa das Aktivitätsdiagramm, das Sequenzdiagramm oder das Zustandsdiagramm.

Als Modellierungssprache besteht die UML aus mehreren Teilen. Dazu gehören die abstrakte Syntax, die konkrete Syntax sowie die Semantik. Die abstrakte Syntax spezifiziert, welche Elemente in einem UML-Modell enthalten sein können, was für Eigenschaften diese besitzen und welche Beziehungen zwischen ihnen bestehen können. Wie die Elemente in den Diagrammen repräsentiert werden, ist in der konkreten Syntax festgelegt. Sie wird auch Notation genannt. Die Semantik beschreibt die Bedeutung der Modellelemente sowie Regeln, die zu einer Verletzung der Semantik führen und damit zum Verlust der Aussage eines Modells. In der UML ist die Bedeutung oder Verwendung der Modellelemente natürlichsprachlich angegeben. Dagegen sind die Verletzungsregeln größtenteils formal mit Hilfe der *Object Constraint Language* (*OCL*) definiert. Die OCL wird in erster Linie zu eben diesem Zweck, dem Formulieren von Constraints (Einschränkungen), genutzt. Laut [30] ist sie aber unter anderem auch zur Definition von Modellabfragen (Queries) und Geschäftsbedingungen geeignet sowie zur Beschreibung von Transformationsvorschriften zwischen Modellen im Kontext der MDA.

#### 3.1.2.1 Aufbau der UML

Die UML ist eine sich selbst beschreibende Modellierungssprache. Dies ist durch die Definition des sogenannten UML-Metamodells möglich, welches dafür wiederum Kernkonzepte der UML verwendet. Sie werden in der *UML Infrastructure* (siehe [28]) beschrieben, die zusammen mit der *UML Superstructure* (siehe [29]) die Elemente des UML-Metamodells definiert. Unter anderem wird dort auch das Konzept der Klasse spezifiziert. Sie enthält Attribute sowie Operationen und kann von weiteren Klassen abgeleitet werden. Außerdem können Klassen durch Assoziationen miteinander verbunden werden. Elemente wie Klassen sind dabei in *Packages* gruppiert. Meist sind die Kernkonzepte aus der Infrastructure ebenfalls in der Superstructure enthalten, wobei sie in letzterer um weitere Eigenschaften ergänzt wurden. Prinzipiell bildet die Infrastructure eine Bibliothek von Elementen, die anderweitig wiederverwendet werden. Das geschieht mit Hilfe des sogenannten *Package Merge*-Mechanismus. Der Package-Merge funktioniert ähnlich, wie die Spezialisierung in der Objektorientierung. Ein Package mit dessen Elementen wird kopiert und inhaltlich mit einem anderen Package vereinigt. Als Resultat entsteht ein Package mit den Elementen ursprünglichen Packages. Kommen in diesen Elemente mehrfach vor, so werden diese zu einem Element zusammengefasst, welche alle Eigenschaften der ursprünglichen Elemente vereinigt.

Die Konzepte, die in der UML Infrastructure definiert werden, bilden ebenfalls die Grundlage der *Meta Object Facility* (*MOF* [27]). Dieses Modell ist das Metamodell des UML-Metamodells und dient dazu Metamodellierungssprachen zu modellieren, eben wie das UML-Metamodell oder das *Common Warehouse Metamodel* (*CWM*). So sind die Elemente des UML-Metamodells Instanzen von MOF-Elementen. Anders ausgedrückt sind in der MOF alle Konzepte beinhaltet, die zur Modellierung von Metamodellen notwendig sind. Zum Beispiel sind die UML-Konzepte Klasse, Assoziation, Attribut alles Instanzen des MOF-Typs Klasse. Beziehungen im UML-Metamodell zwischen den Konzepten, also bspw. „Klasse enthält Attribute“, sind demnach Instanzen des MOF-Elements Assoziation.

Abbildung 3.2 zeigt einen Überblick über die Einordnung der UML. Dort sieht man nicht nur, dass Modelle auf verschiedenen Meta-Ebenen Instanzen von ihren Metamodellen sind, sondern auch, dass Modelle auf derselben Ebene um zusätzliche Elemente erweitert werden können (siehe nächster Abschnitt). Um die semantisch unterschiedlichen Ebenen abzugrenzen, wurde dazu eine 4-Schichten-Architektur definiert. In dieser sind auf der untersten Ebene (M0) die Objekte zu finden, die zur Laufzeit in einem Programm auftreten können. Darüber (M1) befindet sich das Modell, mit dem die Objekte spezifiziert werden. Üblicherweise kann das durch den Quellcode einer Programmiersprache oder einem Klassendiagramm geschehen. Wenn man versucht alle möglichen Modelle (Quellcode, Klassendiagramme, Zustandsmaschinen etc.) zu beschreiben, so braucht man dazu ein Vokabular, um die betreffenden Elemente und ihre Beziehungen auszudrücken. Dieses ist in einer Modellierungssprache (M2) wie der UML definiert, genauer gesagt in deren Metamodell. Damit das Vokabular nicht zu groß wird, beschränkt sich eine Modellierungssprache meist auf

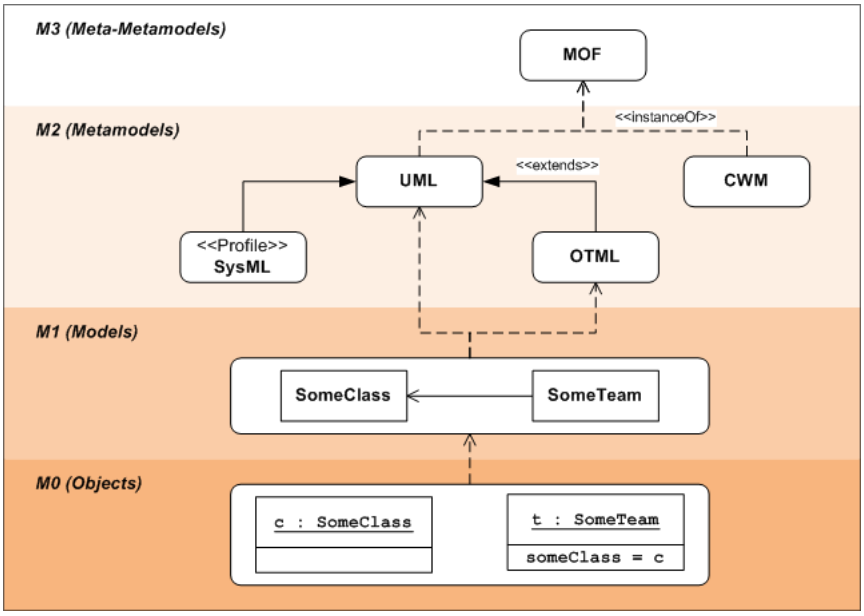


Abbildung 3.2: Überblick über die Metamodellierungsebenen

eine bestimmte Domäne. Bei der UML sind das die objektorientierten Sprachen. Die ebenfalls in der Abbildung dargestellte Modellierungssprache SysML hat demgegenüber den Fokus auf der Anwendungsentwicklung, wo z.B. Elemente wie Requirements, Prozesse und Methoden eine Rolle spielen. Metamodelle, die Gemeinsamkeiten aufweisen, können auch eine Wiederverwendungsbeziehung eingehen, wie das z.B. mit dem Package Merge möglich ist. Solche Beziehungen zwischen Metamodellen und deren Vokabular wird letztendlich im Meta-Metamodell (M3) festgelegt.

Ein Mechanismus, der bei der Erstellung der inneren Struktur der UML vorkommt, wurde bereits erwähnt. Außer dem Package Merge gibt es aber noch weitere Konzepte, die bei der Entwicklung der UML eine Rolle spielen (siehe [2]). Diese Konzepte heißen *Subsetting*, *Redefinition* sowie *Derived Union*. Subsetting bedeutet, dass ein Attribut eine Untermenge eines anderen Attributs darstellt. Das Attribut, das die Untermenge definiert, kann dabei eine geringere Multiplizität besitzen als in der Obermenge und es kann einen spezielleren Typ besitzen. Die Bildung eines Subset kann dabei durch eine Operation bzw. ein OCL-Ausdruck erfolgen oder per Hand. Im ersten Fall spricht man auch von einem Derived-Subset. Der Vorteil besteht darin, dass der Aufwand für den Zugriff auf Elemente über ein Untermengen-Attribut für den Klienten geringer ist, um an dieselben Informationen zu gelangen. Bei nicht-abgeleiteten Untermengen, denen man auch manuell Elemente hinzufügen kann, werden diese automatisch auch dem Attribut, welches die Obermenge darstellt, hinzugefügt. Meistens ist in diesem Fall die Obermenge als Derived Union ausgewiesen. Ein Attribut, das eine Derived Union darstellt, kann nur durch Untermengen-Attribute (Subsetting Properties) Werte enthalten. Dabei bildet sie die Vereinigungsmenge aller Werte in den Untermengen. Die Untermengen-Attribute werden dabei meist in

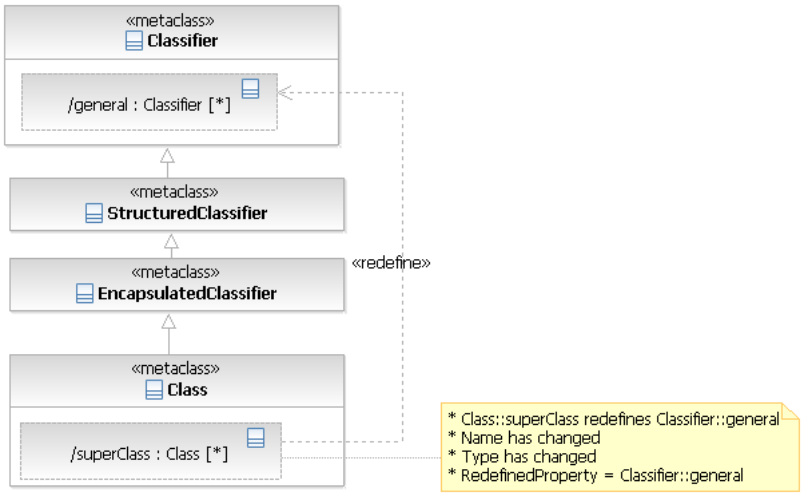


Abbildung 3.3: Anwendung der Redefinition (Quelle: [2])

abgeleiteten Klassen definiert. Das Konzept ähnelt den abstrakten Methoden in Java.

Eine Redefinition erfolgt immer im Kontext einer Generalisierungsbeziehung. Dabei können geerbte Attribute und Operationen redefiniert werden. Das bedeutet, dass etwa der Typ eingeschränkt oder zusätzliche OCL-Constraints hinzugefügt werden können. Weiterhin gilt für Attribute, dass der Name und die Sichtbarkeit geändert werden dürfen. Für Operationen gilt, dass auch die Parameter im Typ eingeschränkt werden können. Im Unterschied zum Subsetting, wird hier das redefinierte Attribut bzw. die redefinierte Operation durch die neue Version ersetzt und nicht ergänzt. In Abbildung 3.3 ist ein Beispiel für die Redefinition eines Attributs angegeben.

#### 3.1.2.2 Erweiterung der UML

Die Erweiterung der UML dient im Allgemeinen der Anpassung an eine bestimmte Domäne. In einer solchen kommen oft spezielle Entitäten mit einer im Kontext besonderen Bedeutung vor. Da die UML universell ist, kann diese Bedeutung nicht schon im UML-Metamodell vorhanden sein. Ein Beispiel ist etwa die Modellierung von Klassen in einem Komponentenframework wie EJB[1]. Bei EJB besitzen bspw. die sogenannten *EntityBeans* und *SessionBeans* eine essentielle Bedeutung. Sie haben bestimmte Eigenschaften, was in den Modellen zum Ausdruck gebracht werden soll. In diesem Fall wäre eine Ergänzung der UML um die EJB-Konzepte sinnvoll. Zu diesem Zweck ist der Erweiterungsmechanismus der UML-Profile vorgesehen.

**UML-Profile.** Das grundlegende Konzept für die Erweiterung eines Metamodells mit Profilen sind die sogenannten *Stereotypes*. Sie können dazu genutzt werden, vorhandene Metaklassen um Constraints und neue Eigenschaften, die

[1]Enterprise JavaBeans, Homepage: http://java.sun.com/products/ejb/

*Tagged Values*, zu ergänzen. Tagged-Values entsprechen dabei den Attributen eines Stereotyps. Die Stereotypen können kein eigenes Verhalten definieren, also keine Operationen enthalten. Desweiteren besitzen sie keinen Zugriff auf die Features der erweiterten Klasse in dem Sinne, dass diese in irgendeiner Weise geändert werden können, sei es durch OCL-Constraints oder über die oben erwähnten Mechanismen wie Subsetting oder Redefinition. Die Einzelheiten zur Definition von Profilen sind in [29] aufgeführt.

Zusammenfassend kann man sagen, dass Profile mit den Konzepten Stereotyp und Tagged-Value einfache Hilfsmittel für die Ergänzung von Metamodellen besitzen. Wenn die Domäne mit diesen Mitteln ausreichend genug beschrieben werden kann, so stellt der Profil-Mechanismus den besten Weg für eine Erweiterung dar. Nicht zuletzt werden Profile von den meisten UML-Werkzeugen unterstützt.

**UML-Metamodel-Extension.** Eine weitere Möglichkeit das UML-Metamodell zu erweitern besteht darin, ein neues Metamodell zu definieren, welches das komplette oder nur Teile des UML-Metamodells via Package Merge wiederverwendet. Anschließend kann man neue Metaklassen hinzufügen oder in der UML vorhandene spezialisieren. Dabei stehen sämtliche Hilfsmittel zur Verfügung, die auch bei der Entwicklung der UML zum Einsatz kommen. Außer Package Merge sind das die oben genannten Konzepte Subsetting, Derived Union und Redefinition.

Durch Redefinition lässt sich bei dieser Erweiterungsmethode, die in [2] als Middleweight oder Heavyweight Extension bezeichnet wird, auch das Verhalten von spezialisierten Metaklassen ändern, was bei den UML-Profilen nicht möglich ist. Zusätzlich kann neues Verhalten spezifiziert werden.

Die Nachteile dieses Erweiterungsansatzes sind dagegen:

- Aufwendigere Erstellung, da komplizierte Konzepte erlernt werden müssen.
- Abhängigkeit von einer bestimmten UML-Version macht Anpassung an neue Version schwieriger.
- Modelle können nicht zwischen beliebigen Editoren ausgetauscht werden.

Grundsätzlich hängt die Entscheidung für die Wahl eines Erweiterungsmechanismus davon ab, was mit dem erweiterten Metamodell geschehen soll. Stehen einfacher Austausch der Modelle zwischen Editoren und einfache Umsetzung im Vordergrund, so würde die Wahl auf eine Erweiterung mittels Profil fallen. Soll jedoch das Metamodell möglichst präzise Aussagen über seine Domäne machen oder aus den Modellen andere Modelle abgeleitet werden, dann spräche einiges für die zweite Variante.

### 3.1.3 Das Eclipse Modeling Framework (EMF)

EMF (siehe [7]) ist ein Projekt mit dem Ziel, durch die Bereitstellung eines Meta-Metamodells und verschiedener Generatoren den Grundstein für eine mo-

dellgetriebene Softwareentwicklung zu legen. Dazu basiert EMF auf der sogenannten *Essential Meta Object Facility* (EMOF), die einen Teil der MOF darstellt. Das EMF-Projekt stellt für die EMOF eine Implementierung bereit, so dass man mittels dieser Implementierungen von Metamodellen erstellen kann. Die Implementierung der EMOF heisst in EMF *Ecore*, weswegen die mit ihr erstellten Modelle auch Ecore-Modelle genannt werden. Essential MOF beinhaltet eine eingeschränkte Anzahl an Konzepten, die aus der UML Infrastructure wiederverwendet werden. Eine Konsequenz davon ist, dass die Erstellung von Ecore-Modellen relativ einfach und benutzerfreundlich ist und dadurch auch von weniger geübten Modellierern durchgeführt werden kann.

Hauptsächlich beinhaltet Ecore Konzepte wie Packages, Klassen, Attribute, Operationen sowie einfache Beziehungen zwischen Klassen, wie gerichtete Assoziationen (Referenzen genannt) und die Generalisierung. Außerdem wird durch Ecore lediglich eine statische Sicht auf ein Programm ermöglicht. Diese ähnelt den Klassendiagrammen aus der UML. Die Persistierung von Ecore-Modellen geschieht über das von der OMG spezifizierte *XMI*-Format, ein XML-Derivat, das speziell zu dem Zweck des Austausches von Metadaten konzipiert wurde. Auch alle auf dem EMF-Projekt aufbauenden Projekte, wie das nachfolgend beschriebene UML2-Projekt, nutzen XMI zur Speicherung von Modellen.

Um ein Modell zu erzeugen, gibt es mehrere Möglichkeiten. Als Ausgangspunkte zur Erstellung von Ecore-Modellen können die Beschreibung eines Web-Services (durch die WSDL [2]), eine XML-Schema-Datei, annotierte Java-Interfaces aber auch UML-Modelle verwendet werden. Zu diesem Zweck sind im EMF diverse Importer verfügbar. Abgesehen von den Transformationen vorhandener Modellbeschreibungen in Ecore-Modelle gibt es aber natürlich auch die Möglichkeit ein solches Modell manuell zu erstellen. Dies kann mit Hilfe des bereitgestellten Ecore-Editors oder mit dem aus dem GMF-Projekt stammenden Ecore-Diagram-Editor bearbeitet werden.

Das Eclipse Modeling Framework ist sehr populär, was es hauptsächlich auch dem bereitgestelleten Quellcode-Generator zu verdanken hat. Dieser bietet die Möglichkeit an, aus den Ecore-Modellen Java-Code zu erzeugen, so dass sie direkt in Anwendungen genutzt werden können. Neben der eigentlichen Implementierung in Java besteht darüberhinaus auch die Möglichkeit zur Generierung von unterschiedlichen Eclipse-Plugins. Unter anderem kann ein Editor-Plugin, ein Unit-Test-Plugin zur Validierung des Modell-Codes sowie ein Plugin mit dessen Hilfe Änderungen an Modellinstanzen über die Eclipse-Benutzeroberfläche durchgeführt werden können, generiert werden. Der erstellbare Editor ist sehr nützlich, wenn auf schnelle und einfache Weise Modelle des generierten Metamodells angelegt und bearbeitet werden sollen. Er besitzt allerdings keine grafische Oberfläche, stattdessen werden die Modellelemente in einer Baum- oder Tabellenansicht dargestellt. Ein Nachteil dieser Art der Visualisierung besteht darin, dass Beziehungen zwischen Modellelementen nicht übersichtlich dargestellt werden.

[2] Web Service Definition Language

### 3.1.4 Das UML2-Projekt

Das UML2-Projekt ist ein Eclipse-Projekt mit dessen Hilfe neben klassischen UML-Modellen auch die Erstellung von UML-Profilen und Metamodellen erfolgen kann. Dafür stellt es eine Implementierung des UML2-Metamodells der OMG bereit. Die Implementierung beruht dabei auf dem Ecore-Metamodell des EMF-Projekts, das wie erwähnt eine Implementierung der (E)MOF darstellt.

Vom EMF „erbt“ das UML2-Projekt auch das XMI-Datenformat, in dem die mit ihm definierten Modelle gespeichert werden können. Durch die Verwendung von XMI ist es auch möglich Modelle mit anderen UML-Werkzeugen auszutauschen, die ebenfalls diesen Standard zur Serialisierung von Metadaten unterstützen.

Durch die Unterstützung aller im Abschnitt zur UML erwähnten Erweiterungsmechanismen, etwa Redefinition, Package Merge, usw., kann das UML2-Projekt auch zur Erweiterung der von ihr bereitgestellten UML-Implementierung genutzt werden. Ein auf diesem Weg erstelltes Metamodell könnte dann mit Hilfe von EMF automatisch implementiert werden. Zur Umwandlung eines UML-Modells in ein Ecore-Modell steht ein entsprechender Import-Mechanismus in EMF zur Verfügung. Sowohl das UML2-Projekt als auch EMF unterstützen die Definition und Interpretation von OCL-Ausdrücken zur Modellierungs- sowie zur Laufzeit. Dies ist durch die Implementierung der OCL in einem weiteren Eclipse-Projekt möglich, welches in die beiden genannten Projekte eingebunden werden kann.

## 3.2 Durchführung

Bevor das OTML-Metamodell entwickelt werden kann, muss noch die Entscheidung getroffen werden, welches Vorgehen zur Erweiterung des UML-Metamodells am sinnvollsten ist. Dabei stehen zwei Möglichkeiten zur Auswahl. Einerseits kann das Metamodell durch ein UML-Profil definiert werden und andererseits mittels einer schwergewichtigen Erweiterung. Wie bereits in Abschnitt 3.1.2.2 erörtert wurde, liegen die Vorteile von Profilen in der einfachen Anwendung sowie einer guten Unterstützung durch vorhandene UML-Editoren. Der Vorteil der schwergewichtigen Auswahl besteht darin, dass das Metamodell sehr viel detaillierter definiert werden kann als mit einem Profil. Der Grund dafür ist, dass hier alle Konzepte zur Verfügung stehen, die auch bei der Entwicklung der UML selbst genutzt wurden. Also Subsettings, Unions und Redefinitions.

Aufgrund der höheren erreichbaren Genauigkeit wird für die Entwicklung des OTML-Metamodells an dieser Stelle die schwergewichtige Vorgehensweise benutzt. Später soll jedoch auch ein Profil verfügbar gemacht werden, damit OTML-Modelle auf beliebigen Editoren bearbeitet werden können, die die Verwendung von Profilen unterstützen. Die Entwicklung eines Profils muss dabei nicht von Grund auf neu erfolgen. Mittlerweise gibt es Werkzeuge wie das *Atlas Model Weaver* Projekt (siehe [5]), mit dessen Hilfe Transformationen zwischen erweiterten UML-Metamodellen und Profilen weitgehend automatisiert durchgeführt werden können.

### 3.2.1 Das OTML-Metamodell

In diesem Unterkapitel geht es darum, das in dieser Arbeit entwickelte Metamodell für die OTML vorzustellen. Dazu werden in den nächsten Abschnitten die meisten Modellelemente sowie deren Beziehungen zueinander erläutert. Auf die Semantik der Elemente wird allerdings nicht eingegangen. Dazu wird entweder auf Abschnitt 2.1 verwiesen bzw. auf die ObjectTeams/Java Language Definition ([17]). Außerdem werden nur eine kleine Auswahl an Constraints als Beispiele präsentiert. Das vollständige Metamodell wird zukünftig als Teil der OTML-Spezifikation in einem separaten Dokument verfügbar sein.

In den nachfolgenden Klassendiagrammen sind teilweise Klassen grün eingefärbt. Damit soll kenntlich gemacht werden, dass diese aus dem UML-Metamodell stammen. Alle anderen (weißen) Klassen sind Teil des OTML-Metamodells.

#### 3.2.1.1 Teams und Rollen

Dieser Abschnitt geht auf die beiden grundlegenden Konzepte in ObjectTeams/-Java ein: Teams und Rollen. Ein Team (`TeamClass`) ist, wie schon im Abschnitt 2.1 beschrieben, nichts anderes als eine Klasse, weshalb es von `Class` aus der UML Superstructure abgeleitet wird.

Wie man in Abbildung 3.4 sieht, gibt es neben den aus der OTJLD bekannten Konzepten Rolle (`RoleClass`) und Rollen-Interface (`RoleInterface`) noch eine zusätzliche, abstrakte Klasse `Role`. Ein Grund für die Einführung dieser Klasse ist Wiederverwendung. Gemeinsame Anteile von `RoleClass` und `RoleInterface` brauchen so nur einmal definiert werden, wie Attribute (`isConfined`, `isOpaque`, usw.), die Assoziation zu `PlayedByRelation` und Constraints. Getrennte Anteile verbleiben in den jeweiligen Klassen. Wenn man sich die OTML genauer anschaut, wird man allerdings feststellen, dass in `RoleInterface` zur Zeit keine neuen Features definiert werden. Was für einen Sinn hat dann eine Spezialisierung von `Role` zu `RoleInterface`? In früheren Entwürfen gab es in der Tat nur die Klasse `Role` und keine Spezialisierungen von ihr. Da sie aber von `Classifier` erben muss, damit sowohl Rollen-Klassen als auch Rollen-Interfaces unterstützt werden, ergeben sich einige Ungenauigkeiten. `Classifier` sind nicht nur Klassen und Interfaces, sondern auch Assoziationen (`Association`), Daten-Typen (`DataType`), etc. Da ein `Classifier` aber von beliebigen `Classifiern` erben kann, wäre es z.B. später möglich gewesen, eine Rollen-Klasse von einer Assoziation erben zu lassen. Erlaubt ist aber laut OTJLD nur, dass eine Rollen-Klasse von einer anderen Klasse erben kann. Man müsste somit eine neue Constraint einführen, die solche Fälle unterbindet. Ähnliches gilt auch für die Definition von enthaltenen Operationen. Ein weiterer Punkt ist, dass sich in Zukunft die speziellen Anteile von Rollen-Klasse und Rollen-Interface durchaus ändern können. Ein Entwurf, der für beide eine eigene Klasse vorsieht, würde Änderungen in dieser Hinsicht deutlich begünstigen. Auch die Formulierung von Constraints für nur eines der beiden Konzepte vereinfacht sich dadurch, da immer ein klarer Kontext – entweder Rollen-Klasse oder Rollen-Interface – existiert. Und für Rollen-Klassen sind durchaus Constraints vorhanden, die nicht für `RoleInterface` zutreffen (s. OTJLD §1.2.1.(d) und §1.5.(b)).

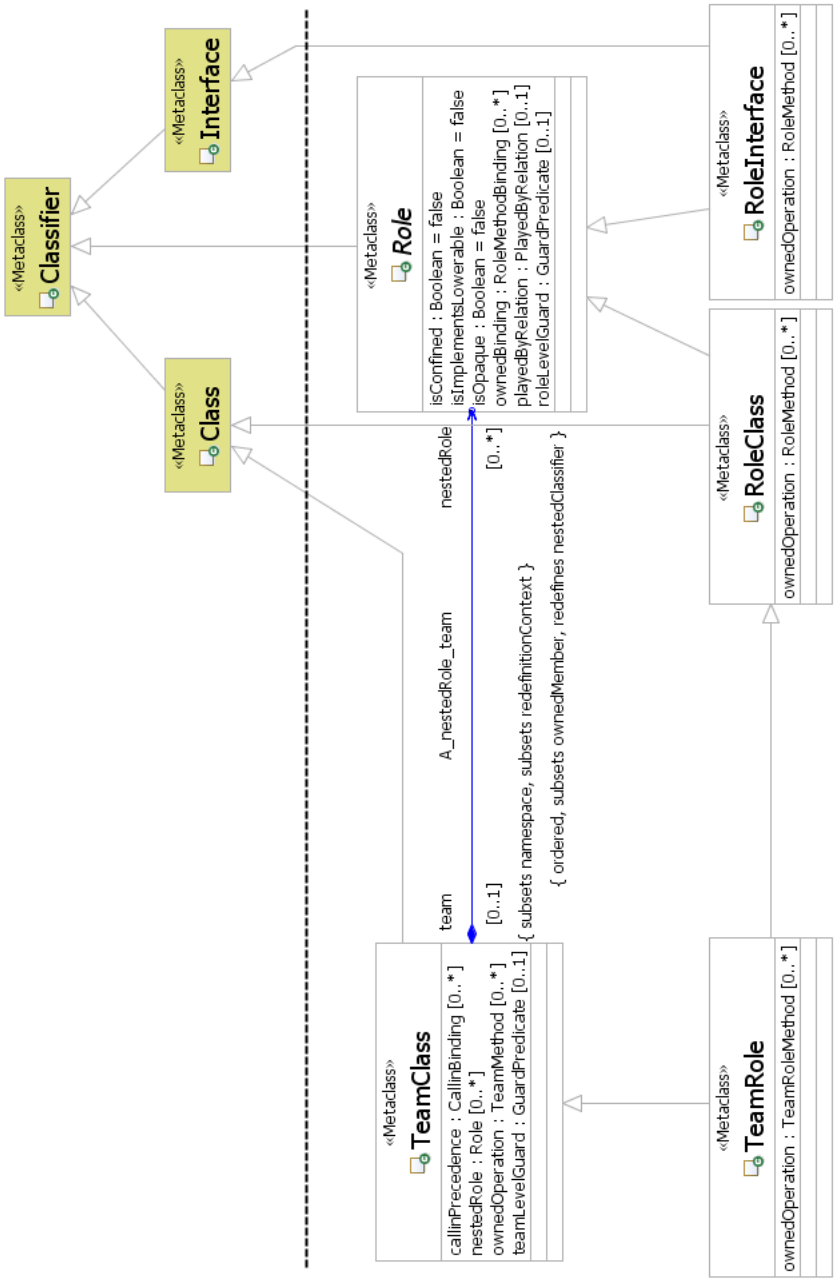


Abbildung 3.4: Modellierung von Teams und Rollen

In der Modellierung der Rollen-Hierarchie erkennt man auch, dass `RoleClass` und `RoleInterface` nicht nur von `Role` erben, sondern auch von den Klassen `Class` bzw. `Interface`, so dass die Spezialisierung von `Classifier` durch `Role` nun unnötig erscheint. Da `Classifier` die Superklasse von `Class` und `Interface` ist, braucht man aber für `TeamClass` nur eine Komposition von Rollen (`nestedRoles`) einzuführen statt derer zwei. Die Beziehung `nestedRoles` ist analog zu der Beziehung `nestedClassifier` zwischen `Class` und `Classifier`. Da Teams nur Rollen enthalten dürfen und keine Klassen oder Interfaces, wird die Assoziation `nestedClassifier` durch `nestedRoles` redefiniert und der Wertebereich auf die Untermenge `Role` eingeschränkt. Würde man jetzt etwa versuchen einer `TeamClass` eine andere Klasse, die keine `Role` ist, hinzuzufügen, so wäre das nicht möglich. Dies gilt auch für den Fall, wenn diese Klasse direkt über die Assoziation `nestedClassifier` von `Class` hinzugefügt werden würde. Diese ist zwar noch sichtbar, allerdings wurde ihre Spezifikation durch die Redefinition ersetzt. Im Zusammenhang mit der Redefinition von geerbten Features wäre es auch durchaus denkbar, die Generalisierungsbeziehung `superClass` auf die Multiplizität 1 und den Wertebereich `TeamClass` einzuschränken. Es ist einem Team, da es durch eine Java-Klasse implementiert wird, nur erlaubt, genau ein anderes Team zu spezialisieren, statt wie es für `Class` zulässig ist, beliebig viele andere `Classes` zu spezialisieren. Der Grund dafür, dass eine Redefinition von `superClass` hier nicht erfolgt, liegt in der weiter unten beschriebenen Implementation dieses OTML-Entwurfs. Redefinitionen werden durch EMF teilweise nicht generiert, so dass dafür selbst Code geschrieben werden muss. Im Falle der Beziehung `superClass`, gelang es im Rahmen dieser Arbeit nicht, eine fehlerfrei arbeitende Implementation durchzuführen.

Interessant für diesen Abschnitt ist weiterhin die Modellierung von Team-Rollen (`TeamRole`). Diese besitzen sowohl die Möglichkeit, ebenso wie Teams, Rollen zu enthalten, als auch die Eigenschaft von Rollen `playedBy`-Beziehungen mit anderen Klassen einzugehen. Daher erbt `TeamRole` hier von `TeamClass` und `RoleClass`. Es wird also an dieser Stelle von Mehrfachvererbung Gebrauch gemacht, was die UML erlaubt. In der Tat wurde das Mehrfacherben auch schon für die Klassen `RoleClass` und `RoleInterface` angewendet. Allerdings tritt für die Klasse `TeamRole` nun ein Problem auf, das gelegentlich bei der gleichzeitigen Spezialisierung mehrerer Klassen vorkommen kann. Dieses wird *Diamond Inheritance* genannt. Das Problem besteht darin, dass `TeamRole` dasselbe Feature von verschiedenen Eltern erbt, wobei dieses in einem Elternteil redefiniert wurde: `nestedClassifier` aus der Klasse `Class`. In `TeamClass` wurde `nestedClassifier` auf Werte vom Typ `Role` eingeschränkt. `Role` und `RoleClass` besitzen ebenfalls dieses Feature, redefinieren es aber nicht, so dass sie beliebige Objekte vom Typ `Classifier` enthalten könnten. Es kommt nun die Frage auf, ob die Redefinition aus `TeamClass` überhaupt Bestand hat oder ob sie vielleicht von der geerbten Originalversion überschrieben wird. In diesem Fall wäre es wünschenswert, wenn die redefinierte Version erhalten bliebe. Die Behandlung des Problems der Diamond Inheritance scheint in der UML aber nicht behandelt zu werden. Auch in Newsgroups erhält man auf Anfrage dazu keine eindeutige Antwort (siehe [11]). Es zeigt sich aber, wenn man das OTML-Metamodell durch EMF in Java-Code übersetzen lässt (siehe Abschnitt 3.2.2),

dass in `TeamRole` tatsächlich die redefinierte Version von `nestedClassifier` namens `nestedRoles` implementiert wird.

Die grafische Notation für Teams, Rollen und Team-Rollen wird im nächsten Kapitel in Abschnitt 4.2.2.2 über den grafischen Editor beschrieben.

#### 3.2.1.2 Rollen-Bindung

Eine Rolle bindet man in OT/J mit Hilfe der PlayedBy-Beziehung entweder an eine Klasse oder ein Interface. Diese werden dann als Basis-Klasse bzw. Basis-Interface bezeichnet. Die PlayedBy-Beziehung kann man sich dabei als gerichtete Beziehung von Rolle zu Basis vorstellen, da die Basis von ihrer Rolle per Definition nichts wissen soll. In Abbildung 3.5 kann man erkennen, dass in der OTML für die PlayedBy-Beziehung eine separate Klasse (`PlayedByRelation`) eingeführt wurde, die `DirectedRelationship` erweitert. `DirectedRelationship` besitzt zwei Assoziationen `source` und `target`, welche die Elemente enthalten, die eine Beziehung miteinander eingehen, ähnlich zu einer `AssociationClass`. Dabei „kennt“ aber nur das `source`-Element die Tatsache, dass es Teil einer Beziehung ist. Weggelassen wurde in der Abbildung, dass in `PlayedByRelation` die geerbten Assoziationen in ihrem Wertebereich eingeschränkt und in `role` und `base` umbenannt wurden. Konnten `source` und `target` ursprünglich in `DirectedRelationship` noch jegliche Objekte vom Typ `Element` aufnehmen, so ist nun der Typ von `source` auf die Klasse `Role` beschränkt und der Typ von `target` auf `Classifier`. In der Abbildung sind die entsprechenden Assoziation für `role` und `base` eingezeichnet. Die Assoziation zwischen `Role` und `PlayedByRelation` ist eine Komposition, so dass keine `PlayedByRelation` ohne zugehörige `Role`-Instanz existieren kann. Die `target`- bzw. `base`-Assoziation ist eine gerichtete Assoziation und ist nur von `PlayedByRelation` in Richtung `Classifier` navigierbar. `Classifier` wurde deswegen gewählt, da diese Klasse die nächste gemeinsame Superklasse von `Class` und `Interface` bildet. Damit nicht andere Objekte vom Typ `Classifier` in `base` gespeichert werden können, die nicht auch vom Typ `Class` oder `Interface` sind, muss zusätzlich in `PlayedByRelation` eine entsprechende Constraint hinzugefügt werden:

```
1  context PlayedByRelation
2  inv binding_interfaces:
3    base.oclIsKindOf(uml::Class)
4      or base.oclIsKindOf(uml::Interface)
```

Darüberhinaus sind in `PlayedByRelation` weitere Constraints vorhanden. Die Constraints sind auch der Grund dafür, dass diese Modellierung der Rollen-Bindung gegenüber der Variante, dass eine Basis direkt mit einer Rolle assoziiert wird, der Vorzug gegeben wurde. Sie können dadurch einfacher formuliert werden. Als Beispiel für eine etwas komplexere Constraint soll die folgende dienen:

```
1  context TeamClass
2  inv no_role_of_the_same_team:
3    let boundRoles : Set(Role) = nestedRole->
4      select(r | not r.playedByRelation.oclIsUndefined())
5    in nestedRole.oclAsType(uml::Classifier)->
6      excludesAll(boundRoles.playedByRelation.base)
```

Die Aussage der Constraint ist, dass für jede Rollen-Klasse gilt, dass die zugehörige Basis-Klasse keine Rollen-Klasse desselben Teams sein darf. Die Be-

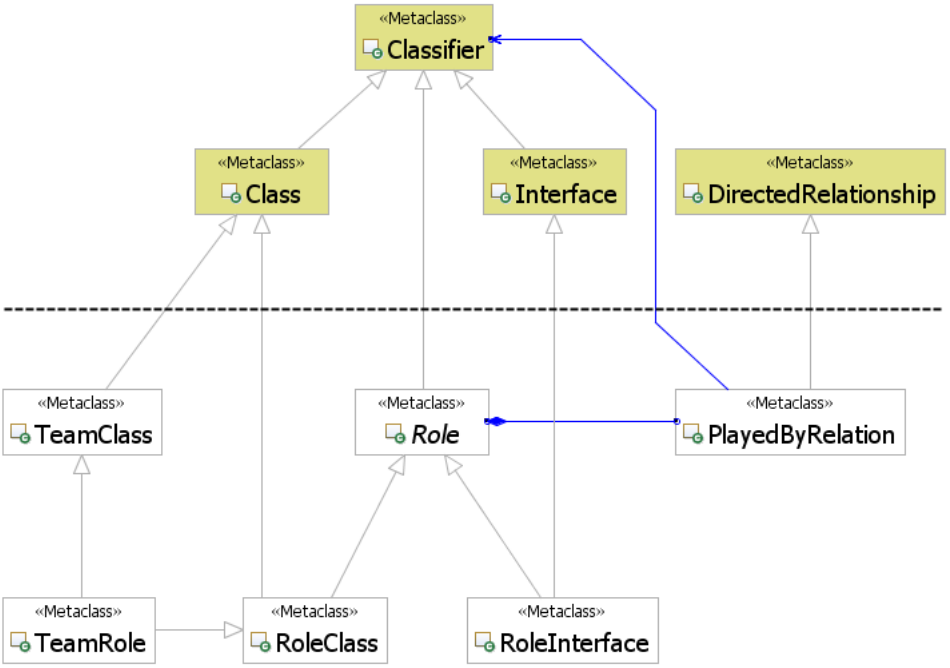


Abbildung 3.5: Modellierung der PlayedBy-Beziehung

schränkung resultiert aus §2.1.2.(a) in der OTJLD. Dabei wird zunächst die Variable `boundRoles` definiert als die Untermenge an Rollen eines Teams, welche an eine Basis-Klasse gebunden sind. Anschließend wird getestet, ob die Basis-Klassen der Rollen in `boundRoles` *keine* Schnittmenge mit den im Team befindlichen Rollen-Klassen bilden.

#### 3.2.1.3 Callout- und Callin-Bindung

Neben Rollen-Bindungen sind in OT/J noch sogenannte Callout- und Callin-Bindungen definiert, die an Methoden in Rollen geknüpft sind. Sie beschreiben, auf welche Weise Attribute oder Methoden der Basis-Klasse in der Rolle verfügbar sind. Für die Modellierung von Callout- und Callin-Bindungen sind mehrere Varianten denkbar. In der OTML wird die in Abbildung 3.6 dargestellte Lösung als günstig angesehen. Erkennbar ist dort eine Vererbungshierarchie mit der abstrakten Klasse `RoleMethodBinding` an der Spitze. Instanzen von `RoleMethodBinding` sind immer in einer Rolle enthalten (Komposition) und referenzieren genau eine Methode aus dieser Rolle. Die referenzierte Rollen-Methode repräsentiert dabei zur Laufzeit einen Delegaten für das zugehörige Basis-Feature. Die Art des Features, also Attribut oder Methode, wird durch die `RoleMethodBinding` implementierenden Unterklassen `CalloutFieldBinding`, `CalloutMethodBinding` und `CallinMethodBinding` bestimmt. Sie enthalten dafür jeweils ein Attribut mit dem Namensschema `base*`, in `CalloutFieldBinding` etwa `baseField`, welches das konkrete Basis-Feature referenziert. Weiterhin ist in `CallinMethodBinding` noch ein Attribut zur Spezifizierung des Callins enthal-

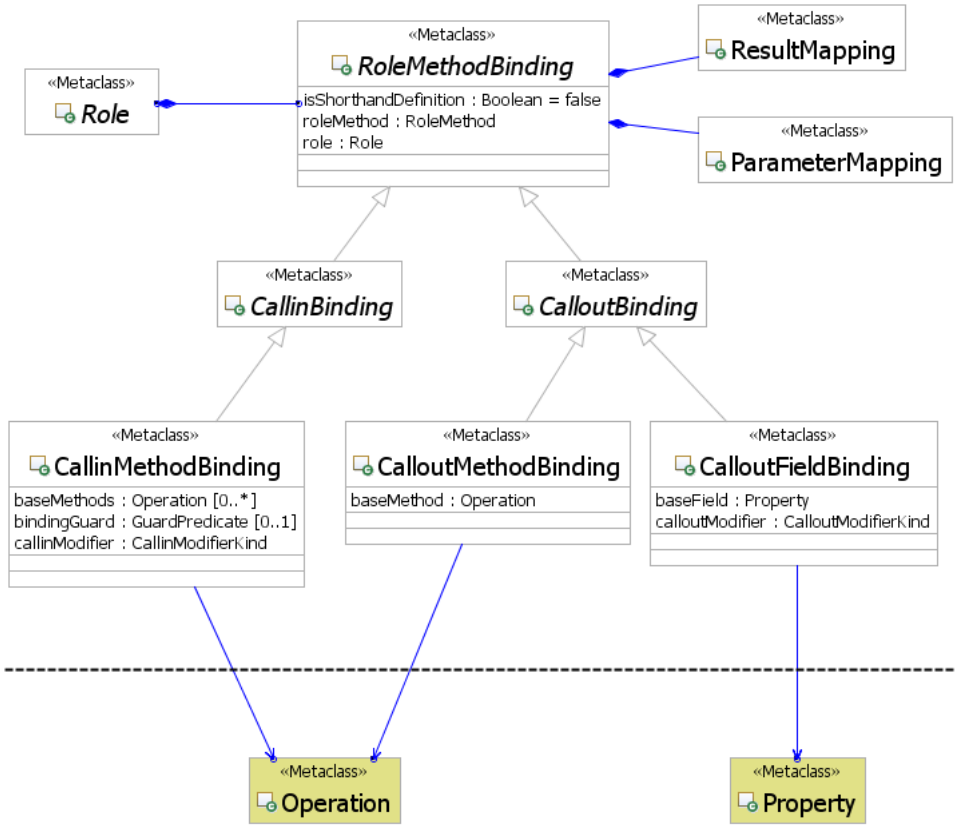


Abbildung 3.6: Modellierung der Callout- und Callin-Bindungen

ten, welches die Werte `after`, `before` und `replace` annehmen kann, die in einer nicht dargestellten Aufzählungsklasse `CallinModifierKind` definiert werden. Analog dazu existiert noch die Aufzählungsklasse `CalloutModifierKind` mit den Werten `get` und `set` für das in `CalloutFieldBinding` enthaltene Attribut `calloutModifier`.

Zwischen der abstrakten Klasse `RoleMethodBinding` und den implementierenden Klassen befindet sich in der Vererbungshierarchie eine weitere Ebene mit abstrakten Klassen, die gemeinsame Attribute der Subklassen zwecks Wiederverwendung enthalten. Im Falle von `CallinBinding` mag das überflüssig erscheinen, da nur die Klasse `CallinMethodBinding` existiert. Allerdings ist für Version 2.0 der OTJLD eine Callin-Bindung für Klassenattribute geplant.

`CallinBinding` enthält weiterhin ein Attribut `label`, mit dem man Callin-Bindungen Namen geben kann für den Fall, dass eine Rolle oder ein Team (indirekt) mehrere Callin-Bindungen für dasselbe Basis-Feature mit identischem `callinModifier`-Wert besitzt. Dann muss man in dem umschließenden Team eine Reihenfolge festlegen, in der die Callin-Bindungen zur Laufzeit aktiviert werden. In TeamClass ist für diesen Fall das Attribut `callinPrecedence` vorgesehen, welches eine geordnete Liste von `CallinBinding`s enthält. Das Attribut `label` besitzt zur Modellierungszeit keine Bedeutung für die Festlegung der Reihenfolge gleicher Callin-Bindungen. Es erlangt erst Bedeutung, wenn aus einem

OTML-Modell Quellcode abgeleitet werden soll. Dann muss laut OTJLD zu Callin-Bindungen immer ein `label` vorhanden sein, das in einem entsprechenden Callin-Precedence-Statement referenziert wird.

Die Reihenfolge von Callin-Bindungen kann außer über das Attribut `callinPrecedence` noch über das Attribut `rolePrecedence` festgelegt werden. Dieses ist ebenfalls in der Klasse `TeamClass` enthalten. Mit Hilfe von `rolePrecedence` wird die Reihenfolge von Callin-Bindungen über die jeweils zugehörigen Rollen definiert. Zu diesem Zweck ist `rolePrecedence` als eine geordnete Liste über Klassen vom abstrakten Typ `Role` definiert.

Die beiden Klassen `ParameterMapping` und `ResultMapping` sind momentan nur Platzhalter und rein der Vollständigkeit halber in der OTML enthalten. Eine genaue Spezifizierung steht in diesem Zusammenhang noch aus. Ein einfacher Ansatz bestünde darin, lediglich Strings für die Mappings bereitzustellen, wobei diese nicht auf Korrektheit überprüft werden könnten. Interessant wäre, ob man mit Hilfe einer detaillierteren Modellierung und OCL-Constraints nicht auch die Korrektheit zur Modellierungszeit garantieren könnte.

#### 3.2.1.4 Team- und Rollenmethoden

Im letzten Abschnitt zur Metamodellierung der OTML werden die Team- und Rollenmethoden erörtert. Sie sind als eigenes Konzept nicht explizit in der OTJLD aufgeführt. Dennoch sind sie im OTML-Metamodell notwendig, da die Konzepte Guard und Declared-Lifting an sie gebunden sind. Daher wurde eine Modellierung gewählt, in der die entsprechenden Klassen `TeamMethod` und `RoleMethod` von `Operation` erben und um die angesprochenen Konzepte erweitert werden. In Abbildung 3.7 sieht man, dass in `RoleMethod` ein Attribut namens `roleMethodGuard` enthalten ist. Dieses Attribut vom Typ `GuardPredicate`, ist ein boolescher Ausdruck, der zur Laufzeit darüber entscheidet, ob eine Callin-Methode, welche mit einer Callin-Bindung assoziiert sein muss, aktiviert werden soll. Damit zur Modellierungszeit schon sicher gestellt ist, dass eine Callin-Bindung definiert ist, wird `RoleMethod` um eine OCL-Constraint ergänzt. Diese verhindert im Falle eines vorhandenden Guards, dass entweder keine Bindung gesetzt ist bzw. keine andere Bindung als solche vom Typ CallinBinding:

```
1  context RoleMethod
2  inv binding_fits_guard:
3    not roleMethodGuard.oclIsUndefined()
4      implies roleMethodBinding.oclIsKindOf(otml::CallinBinding)
```

Guards können auch an anderen Stellen vorkommen. In den Klassen `CallinMethodBinding`, `Role` und in `TeamClass` ist ebenfalls ein entsprechendes Attribut enthalten. Die Bedeutung dieser Guards ist analog zu der oben beschriebenen. Allerdings besitzen sie unterschiedliche Tragweiten. So bestimmt der Ausdruck in `teamLevelGuard` aus der Klasse `TeamClass` zur Laufzeit über die Aktivierung sämtlicher Callin-Bindungen, die in den Rollen einer `TeamClass`-Instanz vorkommen. Guard-Ausdrücke selbst werden im derzeitigen OTML-Metamodell lediglich als Strings in der Klasse `GuardPredicate` gespeichert. In Zukunft könnte aber darüber nachgedacht werden, inwieweit schon während der

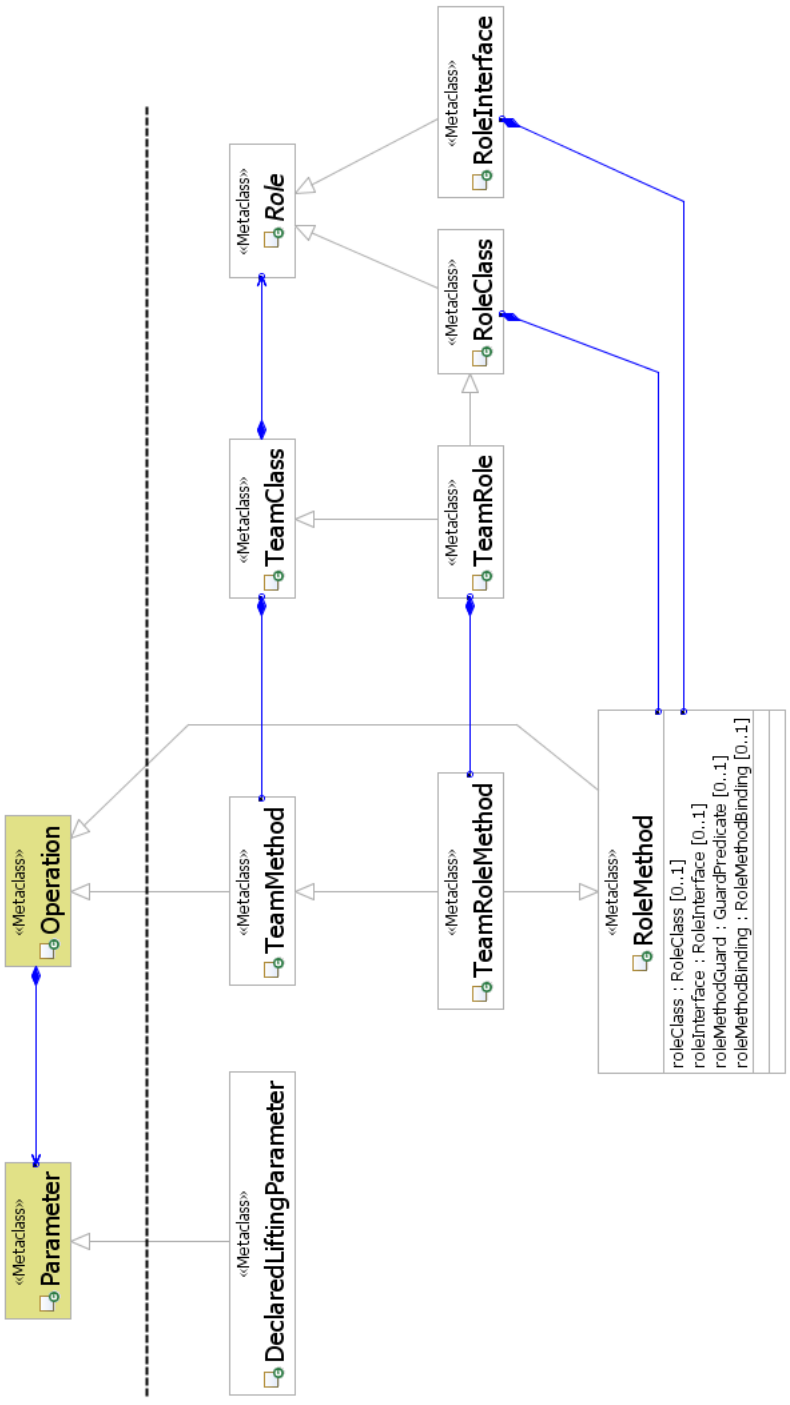


Abbildung 3.7: Modellierung von Team- und Rollenmethoden

Modellierung von OT/J-Programmen die Interpretation und Korrektheitsprüfung von Guard-Ausdrücken sinnvoll ist. Mit dem Einsatz von OCL wäre das durchaus denkbar. Als Beispiel für eine Umsetzung kann die OTML selbst dienen. Die während der Metamodellierung definierten Constraints können geparsed sowie auf Korrektheit untersucht werden und dann zur Laufzeit ausgeführt werden.

Teammethoden sind deswegen in der OTML vorhanden, weil es laut OTJLD möglich sein soll, Parameter in Methoden von Teams als „liftable“ zu deklarieren. Der Sinn ist der, dass der Aufrufer einer Teammethode einen Parameter vom Typ einer Basis-Klasse übergeben kann, die Methode selber jedoch als Parameter die zugehörige Rollen-Klasse empfängt. Die Möglichkeit für das sogennante Declared-Lifting eines Parameters besteht dann, wenn die an die Basis-Klasse gebundenene Rolle direkt in dem Team enthalten ist, in dem auch die Teammethode enthalten ist. Betrachtet man die Modellierung von `TeamMethod`, so stellt man fest, dass sie von `Operation` abgeleitet ist, darüberhinaus aber keine weiteren Features oder Assoziationen definiert. Es ist demnach auch keine Beziehung zu der Klasse `DeclaredLiftingParameter`, welche `Parameter` erweitert, vorhanden. Die Beziehung besteht jedoch indirekt über die geerbte Assoziation `ownedParameter`, über die auch Instanzen von `DeclaredLiftingParameter` referenziert werden können. An dieser Stelle der Modellierung gibt es eine kleine Ungenauigkeit. So wäre es bspw. ebenfalls möglich, eine `DeclaredLiftingParameter`-Instanz von `RoleMethod` oder einer beliebigen anderen Operation aus zu referenzieren, was eigentlich von der OTJLD nicht erlaubt wird. Das lässt sich aber leider nicht vermeiden. Die einzige Möglichkeit besteht darin, mit Hilfe einer Constraint die Referenzierung durch Nicht-`TeamMethod`-Klassen zu invalidieren:

```
1  context DeclaredLiftingParameter
2  inv only_team_methods:
3    operation.oclIsKindOf(otml::TeamMethod)
```

Um die Zugehörigkeit von `TeamMethod` zu `TeamClass` und von `RoleMethod` zu `RoleClass` und `RoleInterface` in der OTML explizit zu machen, wurden die entsprechenden Redefinitionen und Wertebereichseinschränkungen in den geerbten Features (`ownedOperation` aus `Class` und `class` aus `Operation`) vorgenommen. Abbildung 3.7 zeigt, dass `TeamRole` ebenfalls spezielle Methoden enthalten darf. `TeamRoleMethod` ist dabei von `TeamMethod` und `RoleMethod` abgeleitet und darf sowohl `DeclaredLiftingParameter` referenzieren als auch Guards definieren. Durch die notwendige Redefinition von `ownedOperation` in `TeamRole` mit der Einschränkung auf `TeamRoleMethod` ergibt sich in diesem Fall kein Problem daraus, dass `ownedOperation` in den beiden Elternklassen jeweils unterschiedlich definiert wurde.

### 3.2.2 Technische Umsetzung des OTML-Metamodells

Dieser Abschnitt beschäftigt sich mit der technischen Umsetzung des OTML-Metamodells. Die Umsetzung wurde unter Zuhilfenahme der beiden Eclipse-Projekte UML2 und EMF durchgeführt. In diesem Zusammenhang ist der Artikel „Extending UML2: Creating Heavy-weight Extensions“ (siehe [3]) nützlich

gewesen, der den Ablauf von der Erstellung eines Meta-Modells mit UML2 bis hin zur Quellcode-Generierung mit EMF beschreibt.

#### 3.2.2.1 Erstellung des OTML-Metamodells

Die Erstellung des OTML-Metamodells mit dem UML2-Projekt ist relativ unkompliziert. Dazu wird zunächst ein normales UML-Modell erzeugt, was ein Werkzeug aus dem UML2-SDK erledigt. Der Name des Modells soll in diesem Kontext `otml` lauten.Es kann dann bereits mit Hilfe eines weiteren Werkzeugs direkt in ein Metamodell umgewandelt werden. Als Konsequenz der Umwandlung referenziert das Modell jetzt das UML-Metamodell und indirekt dadurch auch das Ecore-Metamodell, welches hier als Implementierung der EMOF das Meta-Metamodell des UML-Metamodells darstellt. Später können durch die Referenzierung z.B. Metamodellelemente erweitert werden. Als nächstes wird dem Modell `otml` der Stereotyp `ePackage` aus dem Ecore-Metamodell hinzugefügt. Dies ist laut Artikel notwendig, damit das Metamodell einen definierten Namespace-Eintrag bekommen kann. Der Namespace sorgt dafür, dass Metamodellelemente später eindeutig identifiziert werden können. Im Prinzip besitzt man nun ein bereits ein Metamodell, allerdings fehlen noch die Modellelemente.

Das UML2-Projekt bietet zu diesem Zweck einen hinreichend komfortablen Editor an. Die Modellelemente sind dabei in einer Baumstruktur angeordnet, dessen Wurzel das Modell an sich darstellt. Als Beispiel zeigt Abbildung 3.8 einen Ausschnitt der Baumstruktur. In dem Screenshot sind einige Elemente hervorgehoben. Ganz oben sieht man den Wurzelknoten `otml`. Darunter wurde für das Beispiel die Klasse `RoleMethod` hervorgehoben, sowie deren Generalisierungsbeziehung zu `Operation` und das Attribut `roleClass`. Zu dem Editor gehört auch noch ein Fenster, das sich im Screenshot unter der Baumstruktur befindet. In diesem kann man weitere Eigenschaften von Elementen, die im Editor ausgewählt wurden, bearbeiten. In diesem Fall gehören die Eigenschaften zum Attribut `roleClass`.

Zum Metamodell lassen sich weitere Modellelemente wie Klassen, Assoziationen, Attribute, Operationen, etc. einfach innerhalb des Editors hinzufügen, indem man auf ein vorhandenes Element rechts-klickt und aus einer kontextbezogenen Auswahl ein Modellelement auswählt. Am Anfang ist immer als Startpunkt das Wurzel-Modellelement vorhanden. Mit Hilfe des Editors lassen sich Beziehungen zwischen Modellelementen in dem Sinne, dass ein Element ein anderes enthält, durch die Baumstruktur übersichtlich darstellen. Wenn es jedoch darauf ankommt, Assoziationen oder andere Beziehungen zwischen Modellelementen auszudrücken, ist die Baumansicht nicht vorteilhaft. Für diesen Fall kann man ein erstelltes Modell auch mit Hilfe des Klassendiagrammeditors aus dem UML2Tools-Projekt visualisieren. Die Abbildungen im vorangegangenen Abschnitt sind allesamt mit diesem grafischen Editor erstellt worden (siehe Abbildungen 3.4, 3.5, 3.6, 3.7). Mit dem Klassendiagrammeditor können aber nicht alle Eigenschaften des Modells verändert werden, wie das etwa mit dem Editor des UML2-Projekts möglich ist. Unter anderen kann die Multiplizität von Assoziationsenden nicht eingestellt werden.

An dieser Stelle soll auch verdeutlicht werden, wieviel einfacher die Bearbei-

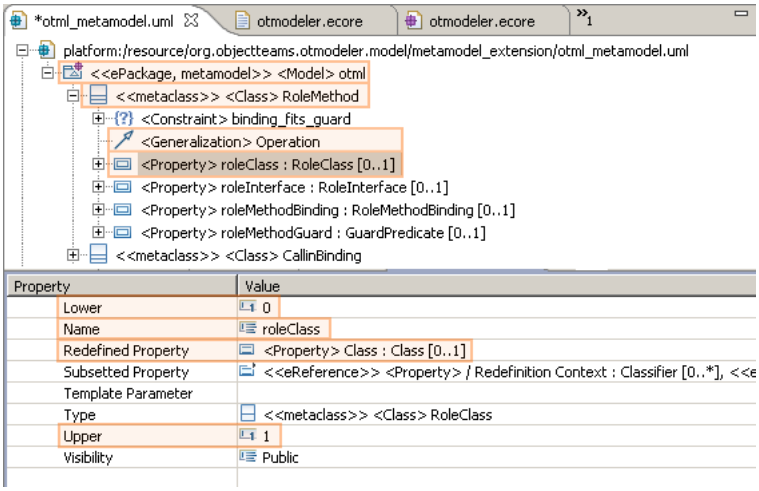


Abbildung 3.8: Beispiel für die Bearbeitung des OTML-Metamodells mit dem UML2-Editor

tung des OTML-Metamodells durch die erwähnten Editoren ist. Das von der OMG als Austauschformat für Modelle und Metamodelle vorgeschlagene Format ist mit XMI ein XML-Derivat. Tatsächlich ist das im letzten Abschnitt beschriebene OTML-Metamodell in diesem Format gespeichert. Es ist also lediglich eine XML-Datei. Das folgende Listing zeigt als Beispiel für das Aussehen des XMI-Formats einen Ausschnitt aus dem OTML-Metamodell. Es wird wie schon im Screenshot des UML2-Editors die Klasse `RoleMethod` in Auszügen dargestellt, wobei die Spezialisierung von `Operation` (7-10), die Definition des Attributes `roleClass` (12-27) und die Redfinition von `class` durch `roleClass` (25-26) zu sehen ist:

```
1  ...
2  <uml:Model xmi:id="_0" name="otml">
3  ...
4    <packagedElement xmi:type="uml:Class"
5      xmi:id="RoleMethod" name="RoleMethod">
6  ...
7      <generalization xmi:id="RoleMethod-_generalization.0">
8        <general xmi:type="uml:Class"
9          href="pathmap://UML_METAMODELS/UML.metamodel.uml#Operation"/>
10     </generalization>

12     <ownedAttribute xmi:id="RoleMethod-roleClass"
13       name="roleClass" type="RoleClass"
14       association="A_ownedOperation_roleClass">

16       <upperValue
17         xmi:type="uml:LiteralUnlimitedNatural"
18         xmi:id="RoleMethod-roleClass-_upperValue"
19         value="1"/>

21       <lowerValue
22         xmi:type="uml:LiteralInteger"
23         xmi:id="RoleMethod-roleClass-_lowerValue"/>

25       <redefinedProperty
26         href="pathmap://UML_METAMODELS/UML.metamodel.uml#Operation-
               class"/>
```

```
27      </ownedAttribute>
28  ...
29    </packagedElement>
30  ...
31  </uml:Model>
32  ...
```

Bevor im nächsten Abschnitt das OTML-Metamodell transformiert werden kann, sollte man es vorher validieren, um sicher zu stellen, dass es korrekt ist. Die Validierung wird durch den UML2-Editor durchgeführt. Dabei werden alle Constraints, die durch die referenzierten Metamodelle definiert werden, ausgeführt. Wenn z.B. in einer Assoziation weniger als ein Member-End vorhanden ist oder ein definierter OCL-Ausdruck syntaktisch nicht in Ordnung ist, so würden entsprechende Problemmeldungen ausgegeben werden.

Sind weiterhin seit der letzten Konvertierung des Modells in ein Metamodell neue Klassen hinzugekommen, so muss dieser Arbeitsschritt wiederholt werden. Durch die erneute Konvertierung in ein Metamodell werden, die neuen Klassen so wie die schon bereits bestehenden als Metaklassen stereotypisiert.

#### 3.2.2.2 Transformation von UML nach EMF

Das OTML-Metamodell wurde im letzten Abschnitt mittels UML2-Editor technisch umgesetzt und kann hier in Java-Quellcode transformiert werden. Dazu wird zunächst ein neues „Empty EMF“-Projekt in Eclipse angelegt. In diesem erstellt man dann mittels eines sogennanten Wizards[3] ein neues EMF-Modell. Während der Erstellung bietet der Wizard die Möglichkeit an, ein bereits vorhandenes UML-Modell zu importieren. In diesem Fall wird das OTML-Metamodell als UML-Modell ausgewählt. Beim Import wird das Metamodell und dessen Modellelemente analysiert und in entsprechende Ecore-Modellelemente übersetzt. Nach der Übersetzung ehält man schließlich ein Ecore-Modell, in welchem die Klassen nicht mehr UML-Klassen entsprechen sondern den Klassen aus der EMOF. Da die UML eine Instanz der MOF ist, und sie darüberhinaus Konzepte definiert, die nicht in der MOF enthalten sind, gibt es einige Unterschiede zwischen dem OTML-Metamodell in der oben beschriebenen Form und der Transformation. Die vorher vorhandenen Assoziationen sind verschwunden, genauer gesagt wurden sie durch das Reference-Konzept ersetzt, wobei eine Klasse ein Attribut als Referenz zu einer anderen Klasse deklarieren kann. Eine Folge davon ist, dass in einer binären Assoziation zwischen zwei UML-Klassen beide Member-Ends in der Assoziation gespeichert sein können, diese aber in der entsprechenden EMOF-Modellierung in die Klassen wandern. EMOF-Klassen haben also gleich viele oder mehr Attribute als die zugehörigen UML-Klassen. Außerdem werden Konzepte, die nicht in der MOF vorkommen, durch Annotationen in den Modellelementen bewahrt. Dazu gehören unter anderem Kommentare, Redefinitionen und Constraint-Definitionen. Abbildung 3.9 zeigt als Beispiel den Ecore-Editor aus dem EMF-Projekt für die oben schon benutzte Klasse `RoleMethod`.

Neben der Erstellung des Ecore-Modells wurde zusätzlich auch ein Model erzeugt, das Informationen für den Quellcode-Generator bereithält, d.h. wie

[3] Ein Wizard wird in anderen Anwendungen üblicherweise auch als Assistent bezeichnet.

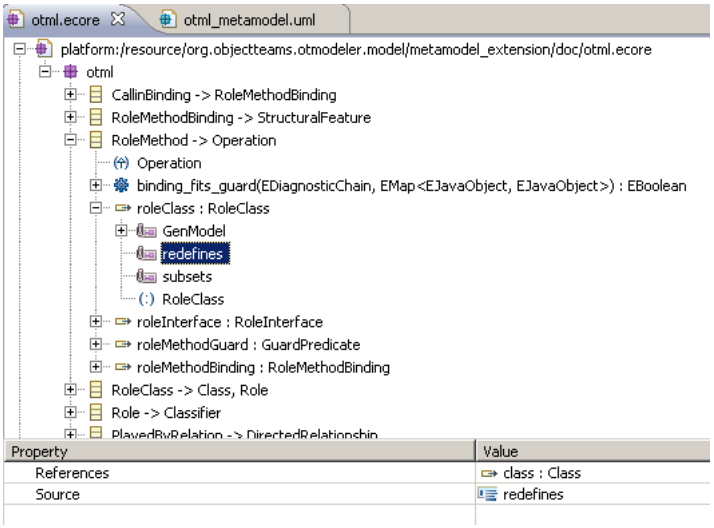


Abbildung 3.9: Ansicht des OTML-Ecore-Modells

welche Klassen, Plugins, usw. generiert werden sollen.

#### 3.2.2.3 Generierung des Quellcodes

Bevor die Quellen des OTML-Metamodells generiert werden können, muss man als erstes das sogenannte GenModel konfigurieren. In dem Artikel [3] wird empfohlen die Einstellungen so anzupassen, dass sie den Einstellungen des GenModels für das UML-Metamodell entsprechen. Dafür muss man die Datei `UML.genmodel` aus dem Plugin `org.eclipse.uml2.uml` im Verzeichnis `/model` öffnen, und die darin enthaltenen Eigenschaften der Elemente von Hand paarweise mit den Einstellungen des hier generierten GenModels abgleichen. Durch das Synchronisieren der Eigenschaften im GenModel erreicht man, dass die Struktur des generierten Quellcodes identisch ist. Damit ist auch die programmiertechnische Benutzung des UML- und des OTML-Metamodells gleich. Dies könnte ein Vorteil bei der weiter unten beschriebenen Erstellung des OTModelers sein, wenn man einen vorhandenen Editor erweitert, der das UML-Metamodell benutzt. Ein weiterer Vorteil annähernd identischer Code-Strukturen ist auch dann gegeben, wenn dieser durch benutzerspezifischen Code erweitert oder geändert werden soll, da mit dem Quellcode des UML-Metamodells Beispiele hierfür vorhanden sind.

Nachdem die Einstellungen des GenModels angepasst wurden, ist man nun in der Lage den Quellcode mit Hilfe des EMF-Codegenerators erstellen zu lassen. Dazu wird im Generator-Menü des GenModels einfach die Art des zu generierenden Quellcodes ausgewählt, in diesem Fall sind das der sogenannte *Model Code*, der *Edit Code* und der *Editor Code*. Der Model-Code entspricht dabei dem nach Java transformierten Ecore-Modells. Zusätzlich zu den darin enthaltenen Klassen, werden auch Hilfsklassen erzeugt, darunter Factory-Klassen, die bei der Instanziierung neuer Modellelement-Instanzen helfen, sowie Klassen für Persistenzzwecke und zur Validierung. Der Edit-Code befindet sich in einem

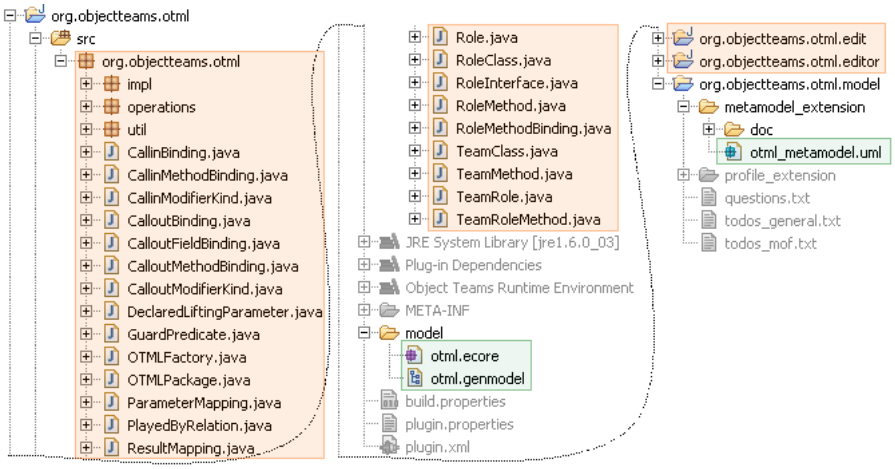


Abbildung 3.10: Übersicht über die generierten Elemente

eigenständigen neuem Plugin, dem Edit-Plugin. Dieses dient unter anderem als Schnittstelle, um die Eigenschaften von Modellelement-Instanzen zur Laufzeit zu ändern oder Icons zur Verfügung zu stellen. Der Editor-Code erzeugt ebenfalls in einem neuen Plugin einen ähnlichen Editor wie den vorher benutzten UML2-Editor. Mit diesem kann man bereits OTML-Modelle erstellen. Allerdings bietet der Editor keine grafische Notation, sondern nur die bekannte Baumansicht. Abbildung 3.10 gibt einen Überblick darüber, welche Plugins und Klassen generiert wurden (orange hervorgehoben) und welche Modelle bereits erstellt wurden (grün hervorgehoben).

Der generierte Java-Code des OTML-Metamodells ist fast vollständig funktionstüchtig. Das bedeutet, dass er nicht weiter manuell angepasst werden muss, außer für einige schon angesprochene redefinierte Features in den Klassen `TeamMethod`, `RoleMethod` und `TeamRoleMethod`. Der Java-Code trennt alle Modell-Klassen in ihre Schnittstellen und diese implementierende Klassen auf. Den Namen der implementierenden Klassen ist bei der Generierung das Suffix „Impl“ angehängt worden, während die Schnittstellen so heißen wie die ursprünglichen Metamodell-Elemente. Sie sind übrigens auch im src-Package in Abbildung 3.10 auf der linken Seite zu sehen. Wenn man sich den Quelltext der Klasse `RoleMethodImpl` anschaut, so fällt auf, dass in einigen Methoden nur eine Exception geworfen wird. Das folgende Listing zeigt z.B. die Methode `setClass_`:

```
1  /**
2   * <!-- begin-user-doc -->
3   * <!-- end-user-doc -->
4   * @generated
5   */
6  @Override
7  public void setClass_(org.eclipse.uml2.uml.Class newClass) {
8    throw new UnsupportedOperationException();
9  }
```

Die in der Methode verwendete Exception dient als Markierung, dass hier eigentlich nur der Methodenrumpf erstellt wurde aber keine Funktionalität. Zur Laufzeit würde ein Aufruf der Methode eine entsprechende Fehlermeldung er-

zeugen. Die Frage ist nun, was man in die Methode schreiben muss, damit sie die von Klienten erwartete Funktionalität anbieten kann. Da keine entsprechende Dokumentation zu diesem Problem existiert, bzw. diese nicht gefunden werden konnte, macht es sich bezahlt, dass der generierte Code mit dem generierten Code des UML2-Metamodells strukturell identisch ist. Für alle betreffenden Methoden, welche nicht mit Funktionalität erzeugt wurden, existieren glücklicherweise entsprechende Pendants. Im obigen Listing, also `setClass_` betreffend, ist das die Methode gleichen Namens in der Klasse `OperationImpl` von der `RoleMethodImpl` abgeleitet ist. Der Inhalt wird aus `OperationImpl` einfach nach `RoleMethodImpl` kopiert. Es müssen dabei lediglich einige Konstanten an den neuen Kontext angepasst werden. Das folgende Listing zeigt das Ergebnis:

```
1  /**
2   * <!-- begin-user-doc -->
3   * <!-- end-user-doc -->
4   * @generated NOT
5   */
6  @Override
7  public void setClass_(org.eclipse.uml2.uml.Class newClass) {
8    if (newClass != eInternalContainer()
9        || (eContainerFeatureID != OTMLPackage.ROLE_METHOD__ROLE_CLASS
               && newClass != null)) {
10     if (EcoreUtil.isAncestor(this, newClass))
11       throw new IllegalArgumentException(
12           "Recursive containment not allowed for " + toString()); //
                 $NON-NLS-1$
13     NotificationChain msgs = null;
14     if (eInternalContainer() != null)
15       msgs = eBasicRemoveFromContainer(msgs);
16     if (newClass != null)
17       msgs = ((InternalEObject) newClass).eInverseAdd(this,
18           OTMLPackage.ROLE_CLASS__OWNED_OPERATION,
19           TeamClass.class, msgs);
20     msgs = basicSetClass_(newClass, msgs);
21     if (msgs != null)
22       msgs.dispatch();
23   } else if (eNotificationRequired())
24     eNotify(new ENotificationImpl(this, Notification.SET,
25         OTMLPackage.ROLE_METHOD__ROLE_CLASS, newClass, newClass));
26 }
```

Damit der Methodeninhalt bei erneuter Generierung des Quellcodes nicht überschrieben wird, muss der im Methoden-Kommentar befindliche Ausdruck `@generated` in `@generated NOT` abgeändert werden.

Wurden alle Methoden mit geeigneter Funktionalität versehen, so ist das OTML-Metamodell nun als Java-Programm verwendbar.

# 4 Der Editor OTModeler

Aus den bisherigen Ausführungen ist bereits ersichtlich geworden, dass der OTModeler in der Lage sein muss, sowohl UML-Klassendiagramme als auch damit ergänzte OT/J-Elemente zu modellieren. Der Vorteil der Wiederverwendung eines vorhandenen Editors wäre, nicht nur den Entwicklungsaufwand zur UML-Modellierung zu sparen, sondern auch, dass künftige Änderungen in der UML von den Entwicklern des Editors durchgeführt werden könnten. Eine Aktualisierung des OTModelers würde so weniger aufwendig ausfallen. Um einen vorhandenen Editor wiederverwenden zu können, muss er aber auch bestimmte Anforderungen erfüllen. So sollte er einer Lizenz unterliegen, die es erlaubt, Änderungen vornehmen zu dürfen. Vorteilhaft wäre auch, dass er im Quellcode vorliegt und man ihn selber kompilieren kann. Ein weiterer Punkt ist, dass er sich ohne weiteres in das OTDT einfügen lässt, also als Eclipse Plugin verfügbar ist. Die letzte wichtige Anforderung an einen wiederverwendbaren Editor ist, dass er eine Modellierungssprache verwendet, die sich einfach und im Sinne der vorher besprochenen OTML-Implementierung anpassen lässt. Für diesen Zweck existieren schon passende Lösungen (siehe Abschnitte 3.1.3 und 3.1.4), die ein Editor im besten Fall verwenden sollte. Als Grundlage zur Evaluierung eines wiederverwendbaren UML-Editors müssen Kenntnisse über den Aufbau der entsprechenden Werkzeuge erarbeitet werden. Weiterhin müssen Bewertungskriterien festgelegt werden, mit Hilfe derer die Auswahl eines Editors vorgenommen werden kann. Anschließend soll die Umsetzung des OTModelers erfolgen.

## 4.1 Grundlagen

### 4.1.1 Das Graphical Editing Framework (GEF)

Die Vorteile der grafischen Interaktion in Bezug auf Darstellung und Bearbeitung von Modellen wurden bereits in den vorangegangenen Kapiteln besprochen. Die Darstellung und Bearbeitung geschieht dabei mit Hilfe von grafischen Editoren. Häufig ist aber das Problem vorhanden, dass es zu einem Metamodell keinen grafischen Editor gibt. In diesem Fall bleibt nur die Möglichkeit, selber einen neuen Editor für dieses Modell zu bauen.

Das *Graphical Editing Framework* (*GEF*[8]) nimmt sich dieser Problematik an, indem es Bestandteile, die in den meisten grafischen Editoren vorhanden sind, selbst zur Verfügung stellt und nur noch um Bestandteile, die zwischen verschiedenen Editoren variabel sind, ergänzt werden muss. Beispiele für konstante Anteile sind Ereignisbehandlung (Mausklicks, Drag-and-Drop, Tastatureingaben), Layouts, Routing von Verbindungen zwischen grafischen Elementen und nicht zuletzt die Logik für die grafische Darstellung im Allgemeinen. Alleine die Implementierung der Darstellungslogik und der Ereignisbehandlung ma-

chen dabei einen erheblichen Teil eines Editors aus, der aber zunächst nichts mit der eigentlichen für den Benutzer erkennbaren Funktionalität zu tun hat. Diese besteht in erster Linie in der Erstellung der im Editor verfügbaren Modellelemente sowie der Erstellung von Beziehungen zwischen ihnen. Nach welchen Regeln Modellelemente benutzt und Beziehungen zwischen ihnen hergestellt werden können, gehört zu der Modellierungssprache, genauer zum Metamodell des Editors und ist dabei von Editor zu Editor variabel. Dies ist der Fall, wenn man davon ausgeht, dass ein Editor nur eine Modellierungssprache unterstützt.

Im GEF sind diese variablen Editorbestandteile im *Model-View-Controller-Muster* (siehe [4]) organisiert. Das Modell kann in dieser Architektur eine beliebige Struktur besitzen, was GEF sehr flexibel in seiner Anwendbarkeit macht. Der View bestimmt die grafische Notation und wird im GEF mit *Figuren* (engl. figures) realisiert, die aus dem Draw2D-Plugin stammen und das ebenfalls zum GEF gehört. Der Controller wird beim GEF *EditPart* genannt und verbindet die Figuren mit den dazugehörigen Modellelementen. Weiterhin ist ein EditPart zuständig für die Interpretation von Benutzeranfragen, die mittels *Tools* oder *Actions* definiert werden können. Tools sind dabei in einer Palette im Editor angesiedelt und können dort ausgewählt werden, während Actions meist aus einem Menü oder über eine Tasteneingabe aufgerufen werden. Der EditPart ermittelt zu einer Interaktion, die durch einen *Request* repräsentiert wird, den dazu passenden Befehl, auch *Command* genannt. Typische Befehle sind z.B. Erzeugen oder Löschen von Figuren und Modellelementen oder das Verschieben einer Figur. Sofern ein EditPart einen passenden Befehl bereitstellen kann, gibt er diesen bspw. einem Tool zurück, welches den Request initiiert hat, damit es ihn letztendlich ausführen kann. Die Auswahl eines passenden Befehls geschieht mit Hilfe sogenannter *EditPolicies*. Requests werden im GEF in bestimmte Bereiche gruppiert, die mit einer Art ID, den *Roles*, versehen sind. Ein EditPart kann dabei zu einer Role immer nur eine EditPolicy besitzen. Typische Rollen sind etwa die `SELECTION_FEEDBACK_ROLE` zur Hervorhebung des Eingabefokus im Editor oder die `LAYOUT_ROLE` für das Layout der Figuren im Editor sowie zur Erstellung neuer Modellelemente.

Besitzt ein EditPart zum Beispiel eine EditPolicy für die Rolle `LAYOUT_ROLE` und bekommt er durch eine Benutzerinteraktion zusätzlich einen Request vom Typ `REQ_CREATE`, so wird er diese EditPolicy fragen, ob sie einen entsprechenden Command zurückgeben kann. Wenn ja, dann wird dieser vom EditPart an seinen Aufrufer bereitgestellt. In dem Command ist das Wissen darüber wie Änderungen durchgeführt werden gekapselt (siehe auch *Command Pattern* in [12]), so dass der Aufrufer diesen ohne weitere Kenntnis ausführen kann. Abbildung 4.1 zeigt die Architektur von GEF und visualisiert den Ablauf einer Benutzerinteraktion.

### 4.1.2 Das Graphical Modeling Framework (GMF)

Das Graphical Modeling Framework ist wie GEF ein Eclipse-Projekt mit dem gemeinsamen Ziel der Erstellung von grafischen Editoren. Tatsächlich basiert GMF zum Teil auf GEF. Dieser Teil, die sogenannte *Runtime*, ist nach dem Vorbild der oben erläuterten Architektur aufgebaut und verwendet auch die

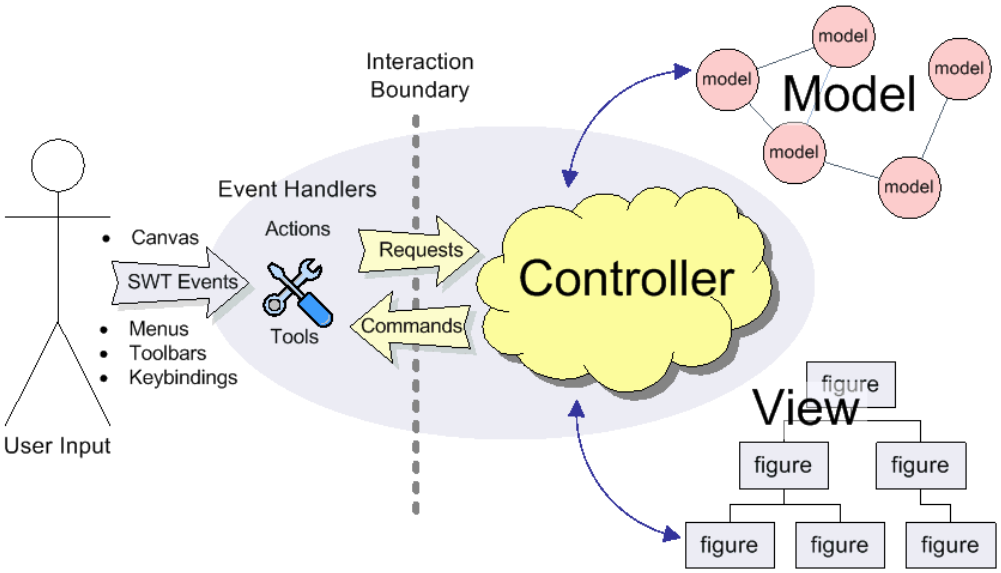


Abbildung 4.1: Überblick über die GEF-Architektur (Quelle: [21])

GEF-Implementierung wieder. Der andere Teil von GMF ist das *Tooling*. Dieses erlaubt eine modellgetriebene Entwicklung, indem man mittels verschiedener Modelle den späteren Editor in seinen variablen Bestandteilen parametrisiert, in anderen Worten welches Metamodell dahintersteckt, wie die grafische Notation (Figuren) aussehen soll, was für Werkzeuge (Tools) der Editor anbietet usw. Aus diesen Modellen erstellt dann ein Generator den Quellcode, aus dem der Editor besteht. Das heisst aber nicht, dass allein die generierten Quellen den Editor ausmachen, vielmehr ergänzen sie die Runtime, die den eigentlichen Kern jedes GMF-Editors bildet.

Mit Hilfe des GMF und dessen modellgetriebenen Ansatzes zur Softwareentwicklung ist eine komplette Editor-Eigenentwicklung weniger aufwendig und fehleranfällig, da ein Großteil der Funktionalität im Editor durch die vorherige Parametrisierung in den Modellen und die daran anschließende Generierung bereits enthalten ist. Es muss entweder gar kein oder nur wenig selbst geschriebener Code hinzugefügt werden. Auch Änderungen an einem generierten Editor sind einfacher, da nur die GMF-Modelle geändert werden müssen und der Quellcode durch erneutes Generieren ersetzt wird. Einzig in dem Fall, wenn der generierte Code geändert wird, muss man darauf achten, dass diese Änderungen mit denen an den Modellen verträglich sind. Ansonsten ist es möglich, dass bei Neugenerierungen Inkonsistenzen auftreten. Das kann übrigens auch bei einer Weiterentwicklung des Generators passieren.

Zur Erzeugung eines GMF-Editors sind die in Abbildung 4.2 dargestellten Modelle in der gezeigten Reihenfolge zu definieren. Unter den allgemeinen, fett gedruckten Modellbezeichnungen sind auch die tatsächlichen Namen der entsprechenden EMF- bzw. GMF-Modelle dargestellt. Dabei ist das dort gezeigte *Domain Model* nichts anderes als ein Metamodell ähnlich dem im letzten Kapitel definierten OTML-Metamodell. Damit ein Metamodell durch einen GMF-

Editor verwendet werden kann, muss es als Ecore-Modell definiert sein. Aus dem Ecore-Modell kann dann mit Hilfe des EMF-Generators und dem *Domain Generation Model* Quellcode erzeugt werden. Die Abbildung zeigt, dass neben dem Metamodell-Code (model) auch ein Edit-Plugin und ein Editor-Plugin generiert werden können (siehe dazu auch Abschnitt 3.2.2).

Das Domain-Model wird weiterhin in den GMF-Modellen benutzt. Diese dienen dazu, die aus dem Metamodell stammenden Konzepte auf die eines grafischen Editors zu übertragen. Zu den Konzepten eines grafischen Editors gehören neben der grafischen Darstellung von Modellelementen in einem Diagramm auch die Bereitstellung von Werkzeugen zur Interaktion mit den Diagrammelementen (siehe auch Abschnitt 4.1.1). Die Definition von Figuren zur Darstellung eines Modellelements als Diagrammelement wird im *Graphical Definition Model* durchgeführt. Damit Diagrammelemente (und auch Modellelemente) erstellt werden können, werden die dafür notwendigen Werkzeuge im sogenannten *Tooling Definition Model* definiert. Das *Mapping Model* hat letztlich die Aufgabe, die im Metamodell vorhandenen Elemente mit ihren Darstellungs- und Werkzeugdefinitionen semantisch in Verbindung zu bringen. Eine typische Aussage des Mapping Models könnte z.B. lauten: „Modellelement X soll im Editor als Kreis dargestellt werden und mit dem Tool namens 'Neues X' erstellt werden können."

Sind alle bisherigen GMF-Modelle und das zugehörige Metamodell hinreichend definiert worden, so kann nun aus dem Mapping-Modell das *Diagram Editor Generation Model* abgeleitet werden. In diesem lassen sich noch einige Implementationsdetails des zu generierenden Editors einstellen. Nachdem auch das erledigt wurde, kann der GMF-Generator aus dem Diagram-Editor-Generation-Modell den Code für den GMF-Editor generieren. Prinzipiell kann dieser ohne weitere Anpassung direkt benutzt werden.

### 4.1.3 Eclipse-Editoren für UML-Klassendiagramme

Hier werden einige bereits vorhandene Eclipse-Editoren für UML-Klassendiagramme vorgestellt, die als Kandidaten für eine Erweiterung zur Modellierung von ObjectTeams/Java-Programmen in Frage kommen. Die folgenden Voraussetzungen müssen dabei von den Editoren erfüllt werden:

- Modellierung von UML-Klassendiagrammen.
- Editor muss als Eclipse-Plugin verfügbar sein.
- Verwendung von Ecore-Modellen als Metamodell.
- Angebot von Erweiterungsmöglichkeiten oder Quelltext verfügbar.
- Lizenz erlaubt Änderung bzw. Erweiterung.

**UML2Tools.** Das *UML2Tools*-Projekt (siehe [10]) ist ein Unterprojekt des *Model Development Tools*-Projektes, zu dem u.a. auch das im vorigen Kapitel

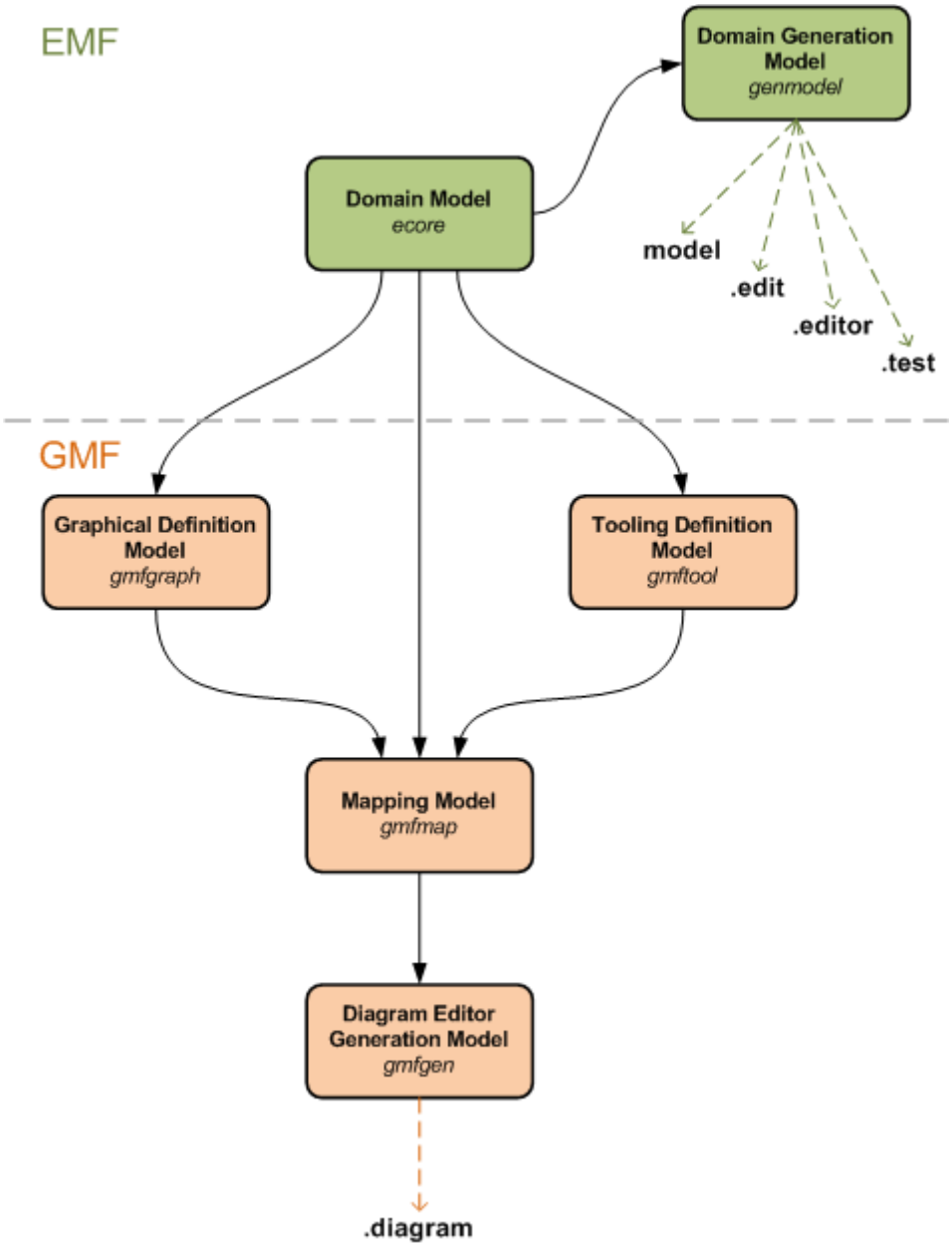


Abbildung 4.2: Modelle und Arbeitsablauf beim GMF

erwähnte UML2-Projekt gehört. Das Ziel des Projekts ist, eine Reihe von Diagrammeditoren für verschiedene UML-Modelle zur Verfügung zu stellen. Neben einem Klassendiagrammeditor gibt es auch Editoren für Komponenten-, Aktivitäts- und Zustandsdiagramme. Das Metamodell aller UML2Tools-Editoren wird durch das UML2-Projekt zur Verfügung gestellt, welches in dieser Arbeit schon für die Umsetzung der OTML verwendet wurde.

Alle Editoren sind GMF-Editoren, so auch der UML2Tools-Klassendiagrammeditor. Zur Darstellung von Klassendiagrammen benötigt dieser zwei Dateien. Dabei werden in der einen Datei die reinen Modellinformationen gespeichert, das heißt welche Modellelemente vorhanden sind, welche Eigenschaften sie besitzen usw. Die andere Datei speichert die notwendigen Daten zur Darstellung der Modellelemente. Dazu gehören neben weiteren Informationen die Koordinaten der Figuren, deren Farbdarstellung oder wie eine Verbindung zwischen zwei Figuren geroutet werden soll. Die Trennung der Klassendiagrammdaten in „Domänendaten“ und „Darstellungsdaten“ liegt darin begründet, dass zu einem Modell verschiedene Sichten definiert werden können. Diese Sichten werden durch die UML2Tools-Editoren unterstützt. Hat man bspw. ein Modell mit dem Klassendiagrammeditor erstellt, so ließe sich aus den Domänendaten auch eine Darstellung im Komponenteneditor erstellen. Die Domänendaten sind in dem Format des UML2-Projekts abgespeichert, so dass diese auch direkt mit dem nicht-grafischen UML2-Editor bearbeitet werden können.

Die Darstellung des Klassendiagrammeditors ist an einigen Punkten gewöhnungsbedürftig. So können in Packages zwar Klassen hinzugefügt werden, aber sie dürfen darin nicht frei positioniert werden und werden immer in einer Liste untereinander dargestellt. Komplexe Diagramme, in denen Klassen aus mehreren Packages miteinander assoziiert sind, erscheinen dadurch unansehnlich. Dieselbe Darstellungsweise trifft auch für interne Klassen zu, jedoch ist sie hier weniger störend, da interne Klassen nicht so häufig auftreten. Ebenfalls unüblich sind die in Klassen und anderen Elementen vorhandenen Compartment-Überschriften. Diese lassen sich allerdings einzeln ein- und ausblenden.

Ein weiteres Manko des UML2Tools-Klassendiagrammeditors besteht darin, dass nicht alle Modelleigenschaften direkt eingestellt werden können. Das trifft besonders auf diejenigen Modellelemente zu, die in anderen enthalten sind. Ein Beispiel dafür sind die Operations-Parameter. In dem Editor kann man bspw. in einer Klasse eine Operation anlegen und zusätzlich dieser dann Parameter zuweisen. Will man jedoch zu einem Parameter die Multiplizität verändern, so ist dies nicht möglich, da der Properties-View nur die Eigenschaften des ausgewählten Elements enthält, in diesem Fall die der Operation. Den Parameter selbst kann man nicht auswählen. Um dennoch solche Einstellungen durchführen zu können, muss man den UML2-Editor verwenden.

Da die UML2Tools-Editoren komplett auf GMF beruhen, fällt die Tatsache, dass von dem Projekt selbst fast gar keine Dokumentation angeboten wird, nicht sehr ins Gewicht. Das GMF bietet dafür einiges an Dokumentation zur Erstellung von Editoren sowie deren Architektur an. Der Schwerpunkt liegt dabei eindeutig auf dem Tooling-Teil des Frameworks und ist mit einigen Tutorials gut beschrieben und nachvollziehbar. Einen eigenen Editor zu generieren fällt so relativ leicht. Zur Runtime-Komponente gibt es dagegen weit weniger

Informationen. In der Hilfe findet man dazu wenige Artikel und die Tutorials und HowTo's dienen nicht gerade als Paradebeispiel für eine umfassende und einfach zu verstehende Dokumentation.

**Topcased.** Das *Topcased*-Projekt (siehe [13]) hat eine weit umfangreichere Zielstellung als das eben beschriebene UML2Tools-Projekt. Der vollständige, etwas umständliche Name lautet „Toolkit In OPen source for Critical Applications & SystEms Development" und deutet bereits daraufhin, dass der Schwerpunkt in der Entwicklung nicht nur auf reiner UML-Modellierung liegt. Vielmehr soll in dem Projekt eine vollständige Werkzeugpalette an Software zur Verfügung gestellt werden, die den kompletten Softwareentwicklungsprozess für kritische eingebettete Systeme abdeckt. Die Initiatoren und Entwickler kommen daher auch aus der Flugzeug-, Weltraum- und Automobilindustrie. In dem Konsortium sind u.a. Firmen wie Airbus, Siemens VDO, EADS Astrium aber auch Forschungseinrichtungen und Universitäten beteiligt.

Der umfangreiche Einsatzbereich spiegelt sich auch in der Anzahl und dem Zweck der verschiedenen grafischen Editoren wieder. Neben UML-Modellen werden auch Modelle für die Metamodelle SAM (Systems Analysis Machines), SysML (System Modeling Language) sowie AADL (Architecture Analysis and Design Language) unterstützt. Weitere Sprachen sollen folgen. Da so eine Vielzahl an grafischen Editoren im Projekt zur Verfügung gestellt werden müssen, bildet das Erstellen von grafischen Editoren ein Schwerpunkt im Topcased-Projekt. Dabei wird ein ähnlicher Ansatz wie bei GMF praktiziert, also die Definition verschiedener Modelle mit anschließender Code-Generierung. Eine weitere Gemeinsamkeit mit GMF besteht darin, dass Topcased ebenfalls auf EMF und GEF basiert und die Modell- und Darstellungsdaten in zwei getrennten Dateien speichert. Die Modelldatei ist wiederum mit dem UML2-Editor des gleichnamigen Eclipse-Projekts bearbeitbar.

Die Verwendbarkeit des Klassendiagrammeditors ist deutlich besser gelungen als beim UML2Tools-Pendant. So lassen sich alle Eigenschaften der Modellelemente auch direkt im Editor, genauer gesagt im Properties View, einstellen. Es ist sogar möglich, einer Operation direkt Constraints oder Methoden-Spezifikationen zuzuordnen. Diese können unter Angabe einer Programmiersprache (Java, C, C++, Python Ada) hinzugefügt werden. Allerdings wird der eingegebene Code nicht direkt im Editor interpretiert. Die Darstellung von Modellelementen im Editorfenster wirkt ausgesprochen bunt. Die Farbgebung lässt sich jedoch für alle Elemente in den Eclipse-Preferences einstellen.

Die Dokumentation von Topcased ist hauptsächlich auf Endanwender zugeschnitten. Zwar findet sich in der Hilfe auch ein Artikel über die Generierung von Topcased-Editoren, allerdings ist dieser nicht sehr anschaulich. Darüberhinaus sucht man vergeblich nach weiteren Informationen über die Architektur der generierten Editoren, einzig eine Javadoc-API ist vorhanden.

**Papyrus UML.** *Papyrus UML* (siehe [14]) ist ein weiteres Projekt, dass verschiedene UML-Editoren unter Eclipse anbietet. Es baut wie die beiden bereits beschriebenen Projekte auf GEF und EMF auf. Ohnehin scheint es einige Ge-

meinsamkeiten mit dem Topcased-Projekt zu geben, wenn man sich z.B. den Editor für Klassendiagramme anschaut. Es fällt auf, dass die Verwendung relativ ähnlich ist und der Funktionsumfang ebenso. Auch hier kann man alle Eigenschaften der Modellelemente im Properties View bearbeiten, wenn auch teilweise deutlich umständlicher als bei Topcased. Die Spezifikation des Verhaltens von Operationen lässt sich hingegen nicht direkt in einer Programmiersprache angeben, dafür aber mit Hilfe anderer Modellkonstrukte, etwa mit einer State Machine. Die Darstellung der Diagrammelemente wirkt im Gegensatz zu Topcased nüchtern. So sind sämtliche Diagrammelemente standardmäßig in gelb gehalten, was sich jedoch auch hier über die Eclipse-Preferences ändern lässt.

Die Dokumentation des Papyrus-UML-Projekts ist sehr übersichtlich. Die in Eclipse integrierten Hilfeseiten sind entweder unvollständig oder erst gar nicht vorhanden. Auf der Webseite findet man größtenteils nur kurze Beschreibungen einzelner Features. Schön ist, dass relativ viele dieser Beschreibungen als Filme präsentiert werden, die durchaus anschaulich sind. Für Entwickler sind dagegen gar keine Informationen vorhanden. Es scheint daher nur für Anwender gedacht zu sein. Der Verdacht verstärkt sich auch dadurch, dass die Sourcen des Projekts im Subversion-Repository zwar über das Internet einsehbar sind, jedoch nicht über einen SVN-Client, wie Subclipse.

#### 4.1.3.1 Auswahl eines Editors

Welcher der vorgestellten Editoren eignet sich nun am besten für eine Erweiterung zum OTML-Editor? Am ehesten scheint aus Anwendersicht der Topcased-Editor geeignet zu sein. Er lässt sich am besten verwenden und integriert sich auch gut mit den Editoren für andere UML-Diagrammarten. Dieser Punkt könnte auch in Hinblick auf die Modellierung von dynamischen ObjectTeams/-Java-Konzepten reizvoll sein. Der Papyrus-Editor ist dem Topcased-Editor sehr ähnlich, weswegen die angesprochenen Punkte ebenso für ihn sprechen können. Allerdings ist die Bedienung nicht ganz so einfach, wie beim Topcased-Editor.

Ein wesentlicher Aspekt, der für die Umsetzung des OTModelers von entscheidender Bedeutung ist, besteht darin wie diese Editoren wiederverwendet werden können. Sowohl der Topcased-Editor als auch der Papyrus-Editor verfügen nicht in dem Rahmen über vorgesehene Erweiterungsmechanismen, als dass sie auf diese Weise um zusätzliche Konzepte erweitert werden könnten. Die Alternativen für das Topcased-Projekt bestehen darin, entweder einen neuen Editor mit dem vorhandenen Generator zu implementieren, wobei auch sämtliche UML-Anteile neu entwickelt werden müßten, oder den Quelltext invasiv bzw. über OT/J zu verändern.

Das UML2Tools-Projekt verfügt dagegen durch die Verwendung von GMF über Erweiterungsmöglichkeiten, mit denen man den vorhandenen Editor um neue Konzepte ergänzen kann. Auch hier sind die Alternativen des Generierens und der Adaptierung gegeben. Ein weiterer Ansatz, der auch für das Topcased-Projekt gilt, ist der, einen Editor zu generieren, der nur die OTML-Konzepte ohne die UML-Anteile enthält. Man könnte dann versuchen mit Hilfe von OT/J diesen Editor mit dem vorhandenen Editor, sei es der Topcased- oder der UML2Tools-Editor, zu verbinden. In diesem Fall müsste man den Anteil des

Codes, der für die ObjectTeams/Java-Konzepte zuständig ist, nicht selbst programmieren, sondern nur noch sogenannten *Glue Code*.

Zusammenfassend bestehen also folgende Möglichkeiten die vorhandenen Editoren wiederzuverwenden und sich dadurch einen erheblichen Teil der Arbeit für die Umsetzung der UML-Konzepte zu sparen:

1. Erweiterung des UML2Tools-Editors über die GMF-Extension-Points.

2. Generierung eines OTML-Editors ohne UML-Anteile und Verküpfung mit UML2Tools-Editor bzw. Topcased-Editor über OT/J.

3. Erweiterung eines beliebigen Editors über invasive Änderung der Quellen oder Adaptierung mit OT/J, wobei die OTML-Anteile selbst programmiert werden müssen.

Vom Programmieraufwand her sind die Punkte 1 und 3 durchaus ähnlich. Denn auch bei der ersten Möglichkeit müssen die OTML-Code-Anteile selber geschrieben werden. Der große Vorteil von 1. besteht allerdings darin, dass eine definierte Schnittstelle zur Verfügung steht, so dass auch bei zukünftigen Versionen des UML2Tools-Editors, die einmal programmierten OTML-Code-Anteile (im besten Fall) nicht geändert werden müssen. Bei Möglichkeit 3 muss man mit hoher Wahrscheinlichkeit davon ausgehen, dass mit der nächsten Version der Programmieraufwand umsonst gewesen ist. Auch bei der zweiten Möglichkeit der Wiederverwendung besteht diese Gefahr, jedoch sind die Auswirkungen nicht ganz so schlimm, da nur der Glue-Code betroffen ist, der im Vergleich zum Gesamtumfang nur einen kleinen Teil ausmacht. Die generierten Anteile können nämlich ebenfalls neu generiert werden.

Für die Umsetzung in dieser Diplomarbeit wurde sich aufgrund folgender Gründe für den Klassendiagrammeditor aus dem UML2Tools-Projekt entschieden. Zunächst sind hier die Erweiterungsmöglichkeiten am besten, da verschiedene Varianten zur Verfügung stehen. Der wohl wichtigste Grund ist aber die im Vergleich mit den anderen Projekten beste (wenn auch nicht überragende) Dokumentation zur Funktionsweise und Interna der generierten Editoren. Bei den anderen Projekten war zwar die Handhabung der fertigen Editoren besser, allerdings birgt ihre Erweiterung aufgrund geringerer Verfügbarkeit von Informationen mehr Risiko und der Einarbeitungsaufwand in die Details der Editoren fällt somit höher aus. Papyrus ist hier übrigens ein gutes Beispiel dafür, wie groß der Einfluss der Dokumentation auf die Wiederverwendbarkeit von Programmen ist.

### 4.1.4 GMF-Grundlagenvertiefung

In Hinblick auf die Auswahl des UML2Tools-Klassendiagrammeditors als zu erweiternder Editor wird an dieser Stelle auf die Details des GMF eingegangen. Besonders interessant sind dabei die vorhandenen Erweiterungsmechanismen sowie der interne Aufbau der sogenannten GMF-Runtime. Auf Grundlage der hier erworbenen Kenntnisse soll später eine Entscheidung für die tatsächliche Umsetzungstrategie des OTModelers erfolgen.

#### 4.1.4.1 Die GMF-Runtime-Architektur

Wie schon in Abschnitt 4.1.2 erwähnt wurde, besteht das GMF aus zwei Teilen. Der erste Teil, das sogenannte Tooling, bestehend aus diversen Modellen und einem Generator wurde in dem genannten Abschnitt bereits kurz vorgestellt. Der zweite Teil ist die Runtime-Komponente von GMF, die für eine Umsetzung des OTModelers noch von Bedeutung sein könnte. Sie soll hier beschrieben werden. Die hier geschilderten Informationen zur Runtime sind aus [19] übernommen worden.

Der Kern der GMF-Runtime ist das sogenannte *Notation Metamodel*. Dieses stellt die eigentliche Verbindung zwischen dem mit Ecore definierten Metamodell und dem auf GEF-basierenden Teil eines GMF-Editors her. Da GEF mit jeglicher Art von Modellen zurechtkommt, in GMF-Editoren jedoch standardmäßig nur mit Ecore-Modellen gearbeitet wird, sind die aus GEF stammenden Code-Anteile in der GMF-Runtime an diese spezielle Gegebenheit angepasst worden. So sind die GEF-Klassen meist durch entsprechende GMF-Klassen abgeleitet, und mit spezifischem Code ergänzt. In Abbildung 4.3 wird eine Übersicht der wichtigsten in GMF vorkommenden Konzepte dargestellt.

Man kann erkennen, dass das Modell unterteilt ist in Elemente, die mit `semantic` annotiert sind, und Elemente mit dem Stereotyp `notational`. Der semantische Teil repräsentiert das eigentliche Modell, welches auch von anderen Programmteilen, nicht nur vom Editor, verwendet werden kann. Die Elemente in diesem Modell sind Instanzen eines Ecore-Modells, weswegen in der Abbildung das Element auch als EObject bezeichnet wurde. Generell werden alle Ecore-Modellinstanzen von EObject abgeleitet, analog zur Object-Klasse in Java. Die semantischen Modellinstanzen besitzen kein Wissen darüber, wie sie in einem Diagramm dargestellt werden. Im Abschnitt zum UML2Tools-Editor wurde auch erwähnt, dass sie in einer eigenen Datei gespeichert werden (dort „Domänendaten“ genannt).

Die ebenfalls in dem Abschnitt erwähnten „Darstellungsdaten“ definieren wie welches semantische Modellelement dargestellt werden soll. Konkret sind sie zur Laufzeit Instanzen des oben angesprochenen Notation Metamodel, die mit den Einstellungen des GMF-Mapping-Models und den anderen GMF-Modellen parametrisiert wurden. In Abbildung 4.4 ist das Modell dargestellt. Das grundlegende Konzept bildet die abstrakte Klasse `View`, die eine Referenz auf ein zugehöriges Objekt aus dem semantischen Modell hält. View und alle abgeleiteten Klassen definieren nun zu ihren referenzierten Modellelementen weitere Eigenschaften, die für eine Darstellung im Editor relevant sind.

Die View-Elemente sind dabei in drei grundlegende Klassen eingeteilt. In der Regel besitzt ein grafischer Editor eine Zeichenfläche, in der weitere Diagrammelemente platziert werden können. Diese Zeichenfläche kann bspw. im semantischen Modell durch ein Package definiert sein. Im notationellen Modell wird das Package über die Klasse `Diagram` repräsentiert, welche dann eine Referenz zum Package besitzt. Die Klasse `Diagram` spannt nun eine Hierarchie über die in ihr enthaltenen Elemente auf. Diese Elemente werden durch die Klassen `Node` und `Edge` typisiert. Dabei können `Node`-Objekte weitere `Node`-Objekte enthalten, während Edges immer im `Diagram`-Objekt enthalten sind. Sie stellen

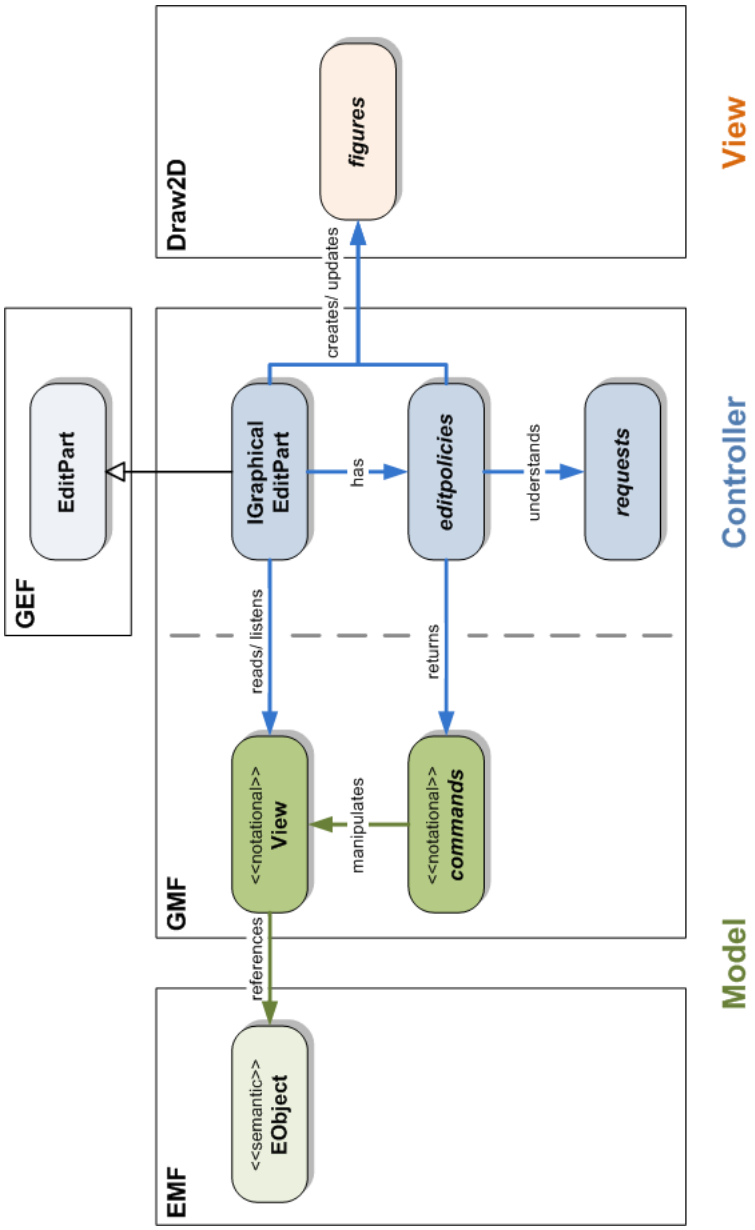


Abbildung 4.3: Übersicht der GMF-Architektur

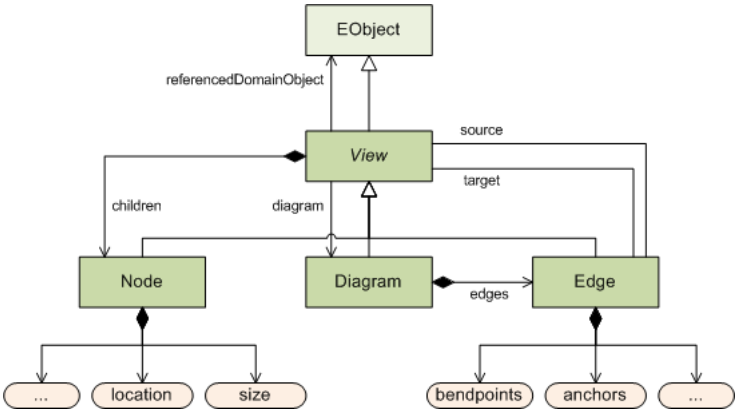


Abbildung 4.4: GMF Notation Metamodel

die Verbindungen zwischen zwei View-Elementen dar und besitzen deswegen je eine Referenz auf ein Start-View-Element (`source`) und ein Ziel-View-Element (`target`). So sind auch Verbindungen zwischen zwei Edges möglich. Im Kontext von UML-Klassendiagrammen würden Edge-Elemente z.B. die Darstellungseigenschaften von Assoziations-, Aggregations- oder Generalisierungsbeziehungen enthalten. Zu den Darstellungseigenschaften gehören etwa die sogenannten *Bendpoints*, das sind die Punkte bei Verbindungen, an denen Knicke vorkommen, oder die sogenannten *Anchorpoints*, die den Punkt definieren, an dem eine Verbindung bei einem Start- oder Ziel-View-Element „andockt“. Typische UML-Elemente für Nodes sind dagegen Klassen und Interfaces. Sie sind in der Diagrammdarstellung durch Eigenschaften wie Position und Größe definiert. Darüber hinaus können sie mit Hilfe von Layouts angeben wie die in ihnen enthaltenen Nodes angeordnet werden sollen. Allgemeine Informationen wie die Farbgebung, die für alle View-Elemente einstellbar sein sollen, werden über die `View`-Klasse definiert.

Wie man in Abbildung 4.3 sehen kann, kapseln die Views auch den Zugriff der EditParts (Controller) auf die semantischen Modellobjekte. Durch die definierte Schnittstelle zum Modell wird die Benutzung vereinfacht und die von GEF geerbte Architektur kann daran angepasst werden. Das bedeutet, dass große Teile der Infrastruktur schon fertig implementiert werden können und nicht mehr generiert werden müssen. Das betrifft insbesondere alles, was mit der Manipulation der Darstellung von Diagrammelementen zu tun hat.

Hauptsächlich werden also bei der Generierung eines GMF-Editors nur die Teile generiert, die etwas mit dem semantischen Modell zu tun haben. Dazu gehören die `Command`-Klassen zur Erzeugung neuer semantischer Modellelemente; die EditParts, da sie die spezifischen Figuren zu einem Modellelement bereitstellen; die EditPolicies, welche die Commands an die EditParts zurückgeben; und diverse `Provider`-Klassen, die zusammen mit sogenannten `Factory`-Klassen für die Erstellung von Views, EditParts und EditPolicies verantwortlich sind. Auf die Provider nimmt der nächste Abschnitt eingehend Bezug.

Als Beispiel soll hier die Erstellung eines neuen Diagrammelementes durch den Benutzer verdeutlichen, wie die erwähnten Klassen der GMF Runtime zu-

sammenarbeiten. Der Benutzer wählt zunächst ein Werkzeug (`CreationTool`) zur Erstellung eines neuen Elements (z.B. eine Klasse) aus der Palette aus und klickt auf ein vorhandenes Diagrammelement (z.B. die Zeichenfläche selbst) im Editor. Über das Diagrammelement kennt das Werkzeug den zugehörigen EditPart (`PackageEditPart`). Das Werkzeug fordert nun vom EditPart durch einen sogenannten `CreateViewAndElementRequest` das entsprechende `Command`-Objekt zur Erstellung einer Klasse an. Dieser Request wird an eine EditPolicy (`CreationEditPolicy`) weitergeleitet, die das Erstellen des entsprechenden `Command`-Objektes für den EditPart übernimmt. Im Unterschied zu GEF werden an dieser Stelle aber zwei Erzeugungsbefehle erstellt, einer für das semantische Modellelement und ein anderer für das View-Element. Der erste Befehl wird tatsächlich von einer weiteren EditPolicy (`PackageItemSemanticEditPolicy`) erzeugt, die für den GMF-Editor generiert wurde. Beide Befehle werden gekapselt an das `CreationTool` zurückgeliefert und von ihm ausgeführt. Der Ablauf dieses Beispiels ist sehr ähnlich zu dem geschilderten Fall im Abschnitt über GEF. Der Vorteil ist der, dass durch die Verwendung des Notation Metamodels sehr viel weniger Code generiert werden muss. Wird etwa ein vorhandenes Diagrammelement im Editor verschoben oder dessen Größe geändert, so wäre der im Beispiel gegebene Ablauf von Aufrufen derselbe, jedoch wird in diesem Fall gar kein generierter Code benutzt, da nur das entsprechende View-Element geändert werden müßte.

#### 4.1.4.2 GMF-Erweiterungsmechanismen

Die GMF-Erweiterungsmechanismen dienen dazu, vorhandene GMF-Editoren, um bestimmte Elemente zu erweitern. Dies geschieht mit Hilfe des Extension-Point-Mechanismus, der in Kapitel 2 vorgestellt wurde. GMF bietet verschiedene Extension-Points an. So können u.a. das Metamodell des Editors um Elemente erweitert werden sowie EditParts, EditPolicies oder View-Elemente hinzugefügt werden. Prinzipiell sind also weitreichende Änderungen an vorhandenen GMF-Editoren möglich. Im folgenden sollen nun einige Erweiterungsmöglichkeiten erläutert werden (siehe auch [19]).

**Extensible-Type-Registry.** Als erster Erweiterungsmechanismus soll hier die *Extensible Type Registry* beschrieben werden (siehe [20]). Sie erlaubt, das von einem GMF-Editor verwendete Metamodell durch ein weiteres Plugin um neue Modellelemente zu erweitern bzw. vorhandene Modellelemente in der Art, wie sie bearbeitet werden, anzupassen. Dazu werden sogenannte *Element Types* definiert, die die Metamodellelemente innerhalb von GMF erweitern. Sie ermöglichen zusätzliche Funktionalität festzulegen, um die Art der Darstellung des Modellelements im Editor anzupassen oder bei Erstellung, Bearbeitung oder Löschung weitere Aktionen auszulösen. Ein Beispiel für solche zusätzliche Funktionalität ist etwa, dass ein Modellelement unter bestimmten Umständen nicht gelöscht werden darf.

Die Spezifikation des zusätzlichen Verhaltens geschieht entweder mit Hilfe der *Edit Helper* oder *Edit Helper Advices*. Die Unterscheidung beruht darauf, dass Element Types in zwei Kategorien eingeteilt werden, in *Metamodel Types*

und *Specialization Types*. Metamodell-Typen korrespondieren direkt mit einem Modellelement aus dem Metamodell und definieren über den ihnen zugehörigen EditHelper das Standardverhalten des Modellelements im Editor. Dabei darf es zu einem Modellelement immer nur einen Metamodell-Typ geben. Soll das Verhalten außerhalb des Editor-Plugins weiter spezifiziert werden, so muss dieses mittels eines Spezialisierungs-Typen geschehen. Dieser kann EditHelperAdvices definieren, die das Standardverhalten ergänzen, wobei noch entschieden werden kann, ob ein Advice vor oder nach dem Standardverhalten ausgeführt werden soll.

Die Element Types werden mit den erforderlichen Klassen für die Bearbeitung und Darstellung, dem referenzierten Metamodell-Typ sowie einer VisualID (Hint) üblicherweise in der `plugin.xml`-Datei des Editor-Plugins über einen Extension-Point angegeben. Die Element-Type-Registry kann zur Laufzeit diese Einträge aus der `plugin.xml` auslesen und weiteren Objekten des Editors zur Verfügung stellen. Element-Types werden z.B. bei der Erstellung von Werkzeugen in der Palette, in den SemanticEditPolicies oder bei der Instanziierung von View-Objekten benutzt.

**Services.** Die *Services*-Infrastruktur bietet sich neben der Extensible-Type-Registry zur Erweiterung von GMF-Editoren an. Dabei bilden Services ein grundlegendes Architekturkonzept der GMF-Runtime. Die Aufgabe eines konkreten Service besteht darin, Objekte bestimmten Typs zu instanziieren und zur Verfügung zu stellen. Unter anderen gibt es Services für View-Elemente, EditParts, EditPolicies oder die Werkzeugpalette.

Ein Service ist aber nicht selbst für die Erstellung zuständig, sondern sogenannte `Provider`. Dabei können mehrere Provider gleichzeitig zu einem Service gehören. Der Service fragt in diesem Fall der Reihe nach alle Provider ab, ob sie ein bestimmtes Element bereitstellen können, und benutzt dann den ersten Provider der dazu in der Lage ist. Um die Reihenfolge zu bestimmen, können Provider priorisiert werden, so dass in dem Fall, da zwei Provider ein Element erstellen können, der höher priorisierte Provider vom Service genutzt wird.

Die Services-Infrastruktur ist direkt in der GMF-Runtime integriert und muss nicht bei der Erstellung eines GMF-Editors generiert werden. Allerdings werden die Provider-Klassen und die Factory-Klassen, die von den Providern zur Erstellung neuer Elemente benutzt werden, generiert. Da die Services die generierten Provider nicht direkt referenzieren können, werden die Provider über spezifische Extension-Points den Services bekannt gemacht. Über diese Extension-Points ist es auch möglich, zusätzliche Provider beim Service anzumelden. Benutzt z.B. ein zu erweiternder Editor einen Provider, um Element X zu erstellen, so kann man einen weiteren Provider mit höherer Priorität als Extension deklarieren, der ein anderes Element vom gleichen Typ jedoch anderer Implementierung bereitstellt.

Ein Beispiel soll die Verwendung von Services verdeutlichen. Dazu wird das im vorigen Abschnitt verwendete Beispiel der Erstellung einer neuen Klasse im Editor benutzt. Wie dort beschrieben wurde, werden durch eine EditPolicy zwei Command-Objekte zur Erstellung eines semantischen und eines notationellen

Modellelementes an das aufrufende Werkzeug zurückgegeben. Der Command für das notationelle Element muss anders ausgedrückt ein View-Element erstellen. Diese Aufgabe delegiert er an den ViewService, der wiederum seine zugordneten ViewProvider aufruft. Existiert ein ViewProvider, so wird mit Hilfe der entsprechenden ViewFactory für das Klassen-View-Element ein Node-Element erstellt und über die ganze Aufrufkette wieder zurückgegeben.

## 4.2 Durchführung

### 4.2.1 Vorgehensweise zur Entwicklung des OTModelers

In diesem Abschnitt sollen die Varianten zur Entwicklung des OTModelers beschrieben werden. Ausgehend von diesen wird anschließend versucht, eine optimale Strategie in Hinblick auf gewisse Kriterien zu finden. Zunächst soll der gesamte Aufwand für die Umsetzung möglichst minimal sein, sowohl in Bezug auf die Einarbeitung in notwendige Konzepte als auch in Bezug auf den Programmieraufwand. Aus diesem Grund wurde sich bereits in Abschnitt 4.1.3 für den UML2Tools-Klassendiagrammeditor entschieden, da dieser einige Möglichkeiten zur Erweiterung anbietet. Es besteht damit die Möglichkeit, bereits vorhandenen Code für den OTModeler wiederzuverwenden.

Ein weiteres Kriterium besteht darin, dass der OTModeler gut anpassbar sein sollte. Durch die Wiederverwendung eines vorhandenen Editors wird eine Abhängigkeitsbeziehung zu diesem eingeführt, so dass Änderungen bei diesem zu Änderungsaufwand beim OTModeler führen könnte. Ein Grund für die Weiterentwicklung des wiederverwendeten Editors, wäre die Weiterentwicklung seines Metamodells, in diesem Fall der UML. Weiterhin könnten auch Änderungen oder Erweiterungen der OTML etwa aufgrund der Einführung neuer ObjectTeams/Java-Konzepte Gründe für eine spätere Anpassung sein. Diese Kriterien sollen hauptausschlaggebend für die Entwicklung des OTModelers sein und bei der Bewertung der folgenden Varianten helfen.

**Erweiterung über Extensions.** Die erste Möglichkeit zur Umsetzung des OTModelers besteht darin, den wiederzuverwendenden Editor mit Hilfe der GMF-Erweiterungsmechanismen um die OTML-Konzepte zu erweitern. Dazu müssten alle in Frage kommenden Extension-Points untersucht werden. Über die Extension-Points müssen dann Klassen angegeben werden, welche die erforderliche Funktionalität in dem entsprechenden Kontext implementieren. Zur Verdeutlichung sei hier nochmal auf das in Abschnitt 4.1.4.2 vorgestellte Szenario mit dem ViewService verwiesen.

Eine Frage bei diesem Ansatz der Erweiterung ist, woher die Funktionalität der Klassen kommt, die in den Extension-Points deklariert werden. Zum einen kann man diese per Hand schreiben. Dazu ist es notwendig, die Schnittstellen zu studieren, die sie implementieren müssen (z.B. AbstractViewProvider). Hilfreich könnte in diesem Zusammenhang auch der Code sein, der in dem vorhandenen Editor existiert. Dieser kann als Beispiel für eine Implementierung dienen. Eine andere Möglichkeit ist, einen separaten GMF-Editor auf Grundlage der OTML zu generieren, ohne das dieser Konzepte aus der UML berücksichtigt. Die so

generierten Quellen ließen sich evtl. als Klassen für die Extension-Points wiederverwenden, was aber nicht garantiert ist.

Die Vorteile dieses Verfahrens sind, dass die Erweiterung des UML-Editors über eine definierte Schnittstelle (die Extension-Points) geschieht. So sinkt die Wahrscheinlichkeit und das Ausmaß für Änderungen am OTModeler, wenn diese über Versionen hinweg stabil bleibt. Absehbar ist noch nicht wie gut das Verfahren in der Praxis umsetzbar ist. Einerseits müssen die Klassen für die Implementierung der Erweiterung entwickelt werden, wobei im besten Fall die Klassen eines weiteren generierten GMF-Editors mit OTML-Metamodell wiederverwendet werden können und im schlechtesten Fall die Klassen manuell erstellt werden müssen. Der zweite Fall würde einen erheblichen Mehraufwand bedeuten. Andererseits ist nicht gesichert, ob überhaupt eine komplette Erweiterung mittels Extension-Points durchführbar ist.

**Erweiterung über OT/J.** Die Erweiterung mittels ObjectTeams/Java ist auf den ersten Blick nicht so unterschiedlich zu der Erweiterung über die GMF-Extension-Points. Der wesentliche Unterschied besteht darin, dass man nicht auf vorgegebene Punkte festgelegt ist, über die die Erweiterung zu erfolgen hat. Vielmehr können beliebige Stellen zur Erweiterung genutzt werden, weswegen dieses Verfahren viel flexibler ist. So gibt es z.B. auch nicht das Problem das beim obigen Verfahren entsteht, wenn Extension-Points nicht vorhanden sind. Extension-Points können aber als Beispiele für Joinpoints dienen, da sie explizit die Stellen kennzeichnen, die für Erweiterungen in Frage kommen.

Ein weiterer Vorteil gegenüber dem obigen Verfahren ist, dass bessere Möglichkeiten hinsichtlich der Wiederverwendung von generiertem Code vorhanden sind. So könnte man wie oben einen GMF-Editor mit OTML-Metamodell generieren und dessen Klassen über Callout-Bindings an den Stellen benutzen, wo man sonst eigenen Code zur Adaptierung schreiben müßte.

Ein Nachteil einer reinen Adaptierung mit Hilfe von OT/J besteht darin, dass wenn der adaptierte Editor sich ändert, die Änderungen an den Adaptern größer sein könnten, als bei der Erweiterung mit Extension-Points. Der Grund dafür ist, dass man sich nicht an eine festgelegte Schnittstelle halten muss. Adaptierungen von Klassen die nicht einer Schnittstelle für Klienten genügen müssen, könnten sich durchaus zwischen Versionen ändern.

**Erweiterung über Extensions und OT/J.** In dieser Diplomarbeit wurde sich nun für die Kombination der beiden vorgestellten Varianten entschlossen. Dabei können die Vorteile von beiden Verfahren übernommen werden. Durch die Verwendung der Extension-Points wird eine Robustheit gegenüber Änderungen am erweiterten Editor erzielt. Durch die aspektorientierten Eigenschaften von ObjectTeams/Java wird die Möglichkeit zur Wiederverwendung von Code erreicht, so dass die Generierung eines GMF-Editors mit dem OTML-Metamodell sinnvoll erscheint. Weiterhin sind diese Eigenschaften nützlich, wenn es sich herausstellen sollte, dass die GMF-Erweiterungsmechanismen nicht durchgängig einsetzbar sind.

Aus den genannten Punkten lassen sich nun einige Aufgaben für die Umset-

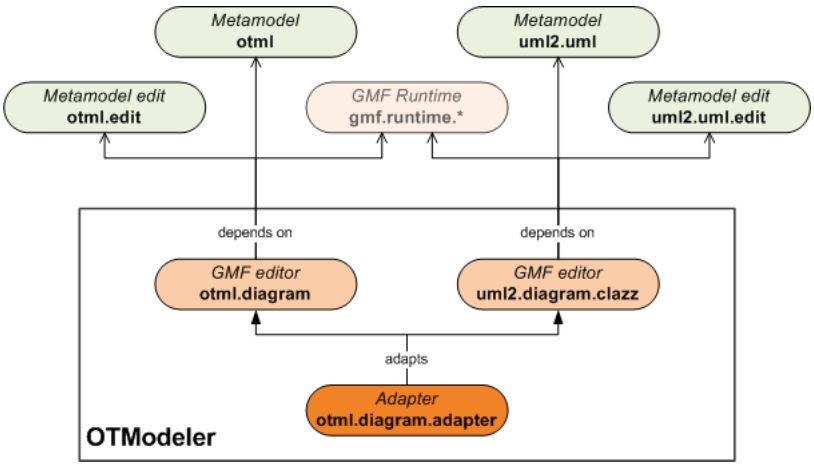


Abbildung 4.5: Übersicht über die Komponenten des OTModelers

zung des OTModelers ableiten. Zunächst wird ein GMF-Editor erstellt, dessen Metamodell lediglich die Konzepte aus der OTML und nicht aus der UML enthält. Dieser Editor wird in den nächsten Abschnitten als OTModeler-Kern bezeichnet. Weiterhin wird ein Adapter-Plugin benötigt, dessen Zweck es ist, die notwendigen Erweiterungen vorzunehmen, sei es durch die angesprochenen Extension-Points oder über OT/J. In Abbildung 4.5 sind die am Ende der Umsetzung vorhandenen Komponenten bzw. Plugins dargestellt sowie diejenigen, auf denen sie basieren. Wie man sieht, wird der OTModeler also nicht eine einzige Komponente darstellen, sondern aus drei Komponenten, dem OTModeler-Kern (otml.diagram), dem UML2Tools-Klassendiagrammeditor (uml2.diagram.-clazz) und dem Adapter-Plugin (otml.diagram.adapter), bestehen.

### 4.2.2 Generierung des OTModeler-Kerns

In diesem Abschnitt wird die Generierung eines Teils des OTModelers beschrieben. Dieser Teil umfasst die Modellierungskonzepte, die aus der OTML stammen und demnach nicht in dem zu erweiternden Editor enthalten sein können. Zu diesen Konzepten gehören etwa Teams, Rollen, Rollen-Bindungen usw. Die Vorgehensweise bei der Generierung mit GMF erfolgt gemäß dem in Abschnitt 4.1.2 vorgestellten Arbeitsablauf. Es werden also zunächst diverse Modelle erarbeitet, dann der Quellcode generiert und zuletzt der generierte Code angepasst, so weit das notwendig ist.

#### 4.2.2.1 Metamodell (ecore)

Das Metamodell ist für einen Diagrammeditor von großer Bedeutung, da dieses die vorhandenen Modellelemente sowie deren Beziehungen zueinander definiert. Im Falle des OTModelers ist das OTML-Metamodell relevant, welches in Kapitel 3 definiert wurde. Es liegt dabei sowohl als Ecore-Modell als auch in Quellcode-Form vor. Das Ecore-Modell ist wichtig für das weiter unten beschriebene Mapping von Werkzeugen, grafischer Notation und Modellelementen,

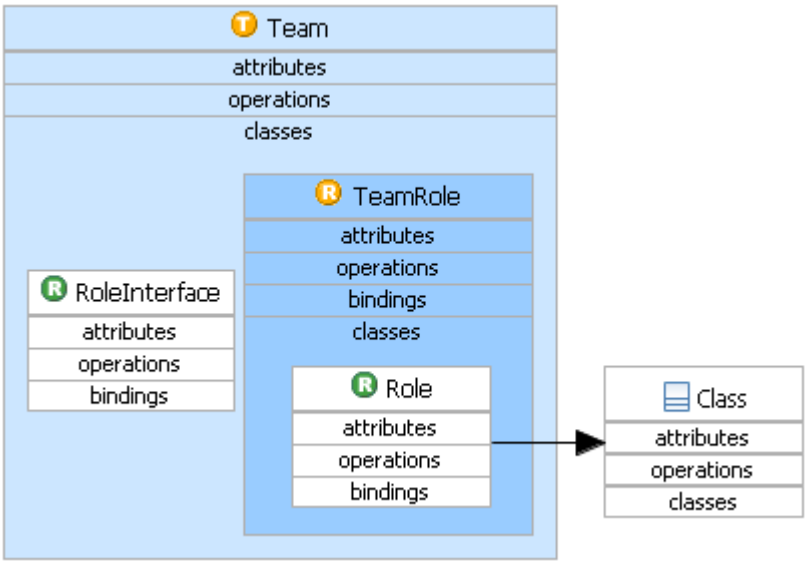


Abbildung 4.6: Darstellung der grafischen Notation

während der Quellcode von den generierten GMF-Klassen benutzt wird.

#### 4.2.2.2 Grafische Notation (gmfgraph)

Im gmfgraph-Modell werden grafische Repräsentationen definiert, die später den Modellelementen im Mapping-Modell zugeordnet werden. Im Rahmen der Modellierungssprache OTML bilden diese Repräsentationen also die grafische Notation, was bedeutet, dass in OTML-Modellen Elemente durch diese Grafiken repräsentiert werden. Dabei soll die Notation der UML-Notation ähneln. Der Grund dafür ist, dass später in OTML-Modellen eben nicht nur OT/J-Konzepte modelliert werden, sondern auch Java- oder allgemeiner objektorientierte Konzepte durch die UML. Damit ein OTML-Modell einen einheitlichen Charakter besitzt, muss also die OTML-Notation angeglichen werden. Weiterhin wurden das Aussehen von OT/J-Modellen in vorhandenen Diagramme (siehe [17] und [18]) mitberücksichtigt.

Abbildung 4.6 zeigt das Ergebnis, wie die grafische Notation für die OTML per Definition auszusehen hat. Dies betrifft die Metamodell-Klassen `TeamClass`, `RoleClass`, `RoleInterface`, `TeamRole` und `PlayedByRelation`.

Da `TeamClass` von der UML-Metaklasse `Class` erbt, sollte die grafische Notation der von `Class` ähneln. Tatsächlich sieht die `TeamClass`-Instanz in der Abbildung (hat den Namen Team) fast so aus wie die Klasse mit dem Namen Class. Der augenscheinlichste Unterschied ist der, dass ein Team eine hellblaue Hintergrundfarbe besitzt. Diese wurde aus den oben zitierten Quellen übernommen. Die Anzahl der vorhandenen Abschnitte (engl. compartments) ist wiederum identisch. Auch die in den jeweiligen Abschnitten möglichen zu enthaltenen Elemente sind fast gleich. So werden im ersten Abschnitt, der sich direkt unter dem Namen der Klasse befindet, die Attribute angezeigt. Im nächsten Abschnitt die Methoden, und im letzten die enthaltenen Klassen. Wie im

Abschnitt 3.2.1 erwähnt wurde, darf ein Team allerdings nur Rollen enthalten und keine anderen Klassen. Wie man sehen kann, werden Rollen wiederum als Rechtecke analog zur Team-Notation dargestellt. Die Darstellung von Klassen-Abschnitten im Klassendiagrammeditor der UML2-Tools ist dagegen textuell. In der UML-Notation werden für die Darstellung von Abschnitten mit internen Klassen keine Vorgaben gemacht, weswegen der Unterschied hier in Kauf genommen wird. Die grafische Darstellung von Rollen als Rechteck innerhalb von Teams oder Team-Rollen hat den großen Vorteil, dass man so auch Beziehungen zwischen ihnen und anderen Elementen definieren kann. Hervorzuheben ist in diesem Zusammenhang die PlayedBy-Beziehung, ohne die eine Modellierung von OT/J-Programmen quasi den Sinn verlieren würde.

Rollen und Rollen-Interfaces werden in OTML-Modellen fast genau wie Klassen gezeichnet. Sie besitzen im Gegensatz zu Team und Team-Rolle einen weißen Hintergrund. Der Klassen-Abschnitt für interne Klassen fällt hingegen weg, da Rollen keine Klassen enthalten dürfen. Dafür bekommen sie aber einen Abschnitt, in dem die Bindungen von Rollen-Methoden an ihre Basis-Features aufgeführt werden. Dieses geschieht rein textuell, z.B. in der Form für Callin-Bindungen:

```
1  roleMethod <- after baseMethod
```

Dabei ist `roleMethod` der Name einer Rollenmethode und `baseMethod` der Name der zugehörigen Basis-Klassenmethode.

Team-Rollen erweitern, wie aus Abschnitt 3.2.1 bekannt ist, sowohl `TeamClass` als auch `RoleClass`. Daher sieht ihre Notation wie eine Mischung aus den beiden Klassen aus. So besitzen sie einen Klassen-Abschnitt, in dem Rollen enthalten sein können, sowie einen Abschnitt für Bindungen. Für beide Abschnitte gilt das gleiche wie oben geschildert. Darüberhinaus haben sie eine dunkelblaue Hintergrundfarbe.

Die Abschnitte sind in allen Elementen mit einer Überschrift versehen. Die Überschriften sind in der UML-Notation nicht vorhanden, weswegen diese Darstellung im Vergleich zu anderen Klassendiagrammen möglicherweise abweichen kann. Im UML-Klassendiagrammeditor des UML2Tools-Projekts sind diese standardmäßig aktiv, weswegen bei der OTML-Notation diese ebenfalls angezeigt werden können. Es ist aber auch möglich für einzelne bzw. alle Abschnitte die Überschriften wegzulassen, was auch vom OTModeler unterstützt wird. Außerdem ist in der aktuellen Version der OTML die Anzeige von Stereotypen nicht vorgesehen.

Die Umsetzung der Notation geschieht im gmfgraph-Modell. In Abbildung 4.7 sieht man eine Ansicht auf das Modell in einem vom GMF-Projekt bereitgestellten Editor. Dieser ähnelt dem schon bekannten UML-Editor und dem Ecore-Editor. Hervorgehoben ist als Beispiel die Definition der Team-Rollen-Notation. Sie besteht aus einem Rechteck, in dem weitere Rechtecke enthalten sind. Die enthaltenen Rechtecke sind die Abschnitte in den Diagrammelementen, wobei das erste Rechteck den Namen enthält. Außerdem kann man hier das Layout der enthaltenen Rechtecke angeben, also wie die Anordnung aussehen soll. Verwendet wird momentan das Flow-Layout in einer Spaltenausrichtung. Das bedeutet, dass alle Rechtecke untereinander mit derselben Breite

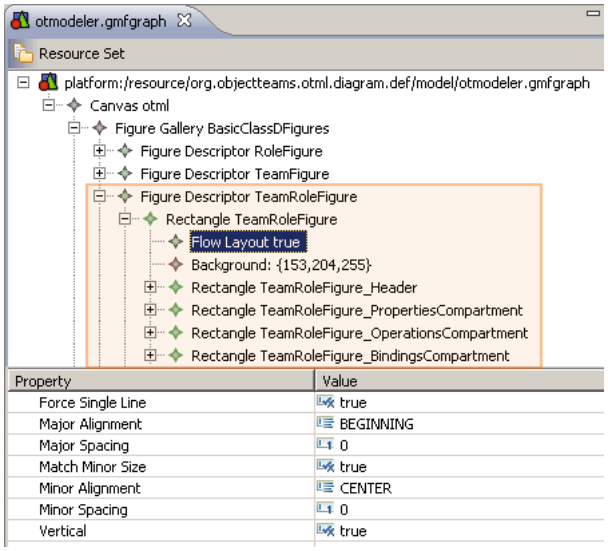


Abbildung 4.7: Ausschnitt aus dem Modell der grafischen Notation

angeordnet werden. Weierhin kann man sehen, dass die Hintergrundfarbe für `TeamRoleFigure` als dunkelblau im RGB-Farbmodell definiert wurde.

#### 4.2.2.3 Werkzeugdefinition (gmftool)

Werkzeuge sind in einem GMF-Editor von Bedeutung, wenn man neue Elemente zum Modell hinzufügen (Erstellungswerkzeuge) oder einfach vorhandene Elemente im Modell selektieren möchte (Auswahlwerkzeug). Während das Auswahlwerkzeug zum Standardrepertoire eines GMF-Editors gehört, sind konkrete Erstellungswerkzeuge abhängig von den Elementen einer Modellierungssprache, und deswegen auf spezifische Editoren beschränkt. Werkzeuge befinden sich in GMF-Editoren immer in einer sogenannten Werkzeugpalette. In Abbildung 4.8 sieht man die Werkzeugpalette (im rechten Teil der Abb.) so, wie sie im aktuellen OTModeler aussieht. Die oberen Werkzeuge in der Palette, gemeint ist der Teil vom Eintrag „Team Class“ bis zum Eintrag „PlayedBy Relation“, sind das Ergebnis des hier beschriebenen Generierungsvorgangs. Die darunter befindlichen Werkzeuge entstammen dem UML2Tools-Editor.

In der linken Hälfte der Abbildung 4.8 befindet sich das gmftool-Modell. Hier werden die später in der Palette benötigten Werkzeuge definiert. Man kann gut erkennen, wie die hier gruppierten Einträge sich auch in der Palette wiederfinden. Die jeweils zusammengehörenden Gruppen sind farblich hervorgehoben. Durch Verschachteln von Gruppen im gmftool-Modell kann man in der Palette aufklappbare Menüs erstellen, zu sehen an der Gruppe mit den Methoden-Bindungen.

#### 4.2.2.4 Mapping-Modell (gmfmap)

In diesem Abschnitt werden alle bisherigen Modelle im Mapping-Modell mit einander verknüpft. Wie man in Abbildung 4.2 erkennen kann, gehört dazu das

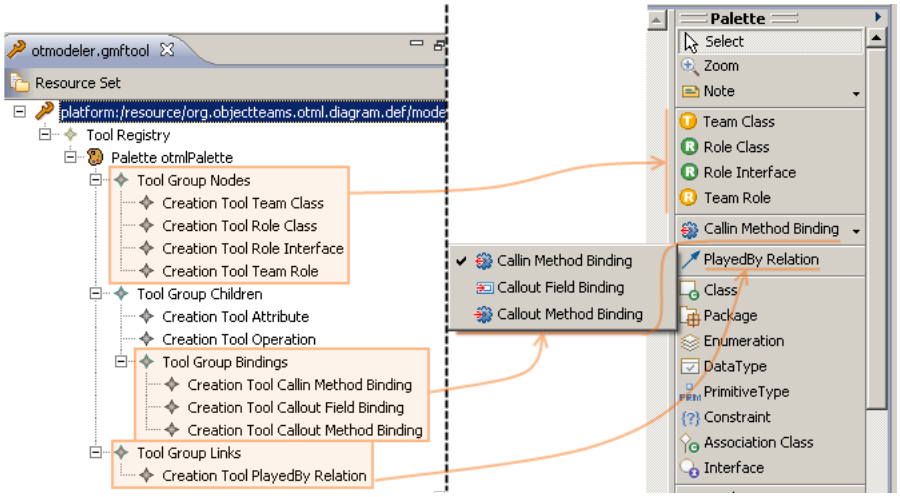


Abbildung 4.8: Zuordnung von Werkzeugdefinition zu Paletteneintrag

Ecore-Modell, die grafische Notation (gmfgraph) und die Werkzeugdefinition (gmftool). Ein Modellelement aus dem Metamodell wird also eine Notation und ein Werkzeug zugeordnet, so dass es im Editor später erstellt und dargestellt werden kann.

Als Beispiel, wie ein Mapping im Detail definiert wird, soll hier das Team-Konzept dienen. Abbildung 4.9 zeigt den entsprechenden Ausschnitt aus dem Mapping-Modell. Übrigens sieht man hier, dass die genannten, referenzierten Modelle mitaufgeführt sind. Der betreffende Abschnitt ist mit einer „1“ markiert. Generell ist in der Struktur der Baumansicht an oberster Stelle des Modells der Wurzelknoten „Mapping“ vorgegeben. Darunter befinden sich die Kinder von „Mapping“, die „Top Node Reference“-Knoten und die „Link Mapping“-Knoten. „Top Node Reference„-Knoten können wiederum „Child Reference“-Knoten enthalten. Man kann sich diese Struktur erklären, indem man sich vorstellt, dass grafische Modelle immer auf einer Zeichenfläche dargestellt werden. Auf dieser Zeichenfläche können Modellelemente gemalt werden. Aber nur bestimmte Elemente darf man direkt auf die Fläche malen. Dieser Umstand ergibt sich daraus, dass die Zeichenfläche selbst ebenfalls ein Modellelement darstellt und dadurch festlegt, welche weiteren Elemente sie laut des zugehörigen Metamodells enthalten kann. Für OTML-Modelle ist in diesem Fall als Modellelement für die Zeichenfläche die Klasse `Package` aus der UML gewählt worden. Ein `Package` kann außer weiteren `Package`s auch Klassen, Interfaces und weitere Elemente vom Typ `PackageableElement` enthalten. Die Zuordnung von `Package` zur Zeichenfläche befindet sich im „Mapping“-Kind „Canvas Mapping“.

Schaut man sich das OTML-Metamodell genau an, so stellt man fest, dass fast alle definierten Elemente in anderen Elementen des Metamodells enthalten sind. Die einzige Ausnahme bildet die Klasse `TeamClass`. Diese erbt durch `Class` indirekt die Möglicheit, einem `Package` hinzugefügt zu werden. Damit ist `TeamClass` das einzige Element das direkt auf dem Zeichenblatt platziert werden darf, was durch die Zuordnung als „Top Node Reference“-Knoten ausgedrückt

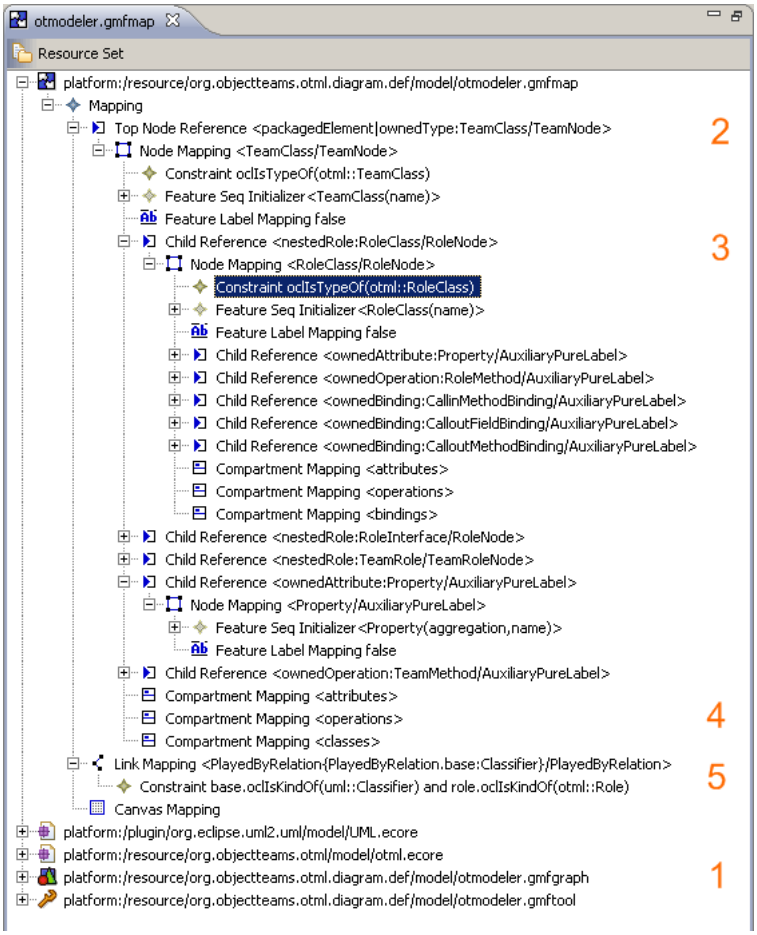


Abbildung 4.9: Ansicht des Mapping-Modells in GMF

wird. In der Abbildung sieht man, dass bei der „2“ `packagedElement` hinter dem Eintrag „Top Node Reference“ steht. Dies ist das Feature von `Package`, in dem ein Team enthalten ist, wenn es dem Zeichenblatt hinzugefügt wird. Theoretisch könnten eigentlich auch Rollen in einem `Package` vorhanden sein, da diese `Classifier` und dadurch ebenfalls `PackageableElements` sind. Jedoch ist die Beziehung von Rollen zu Teams im OTML-Metamodell so definiert, dass zu einer Rolle immer ein Team vorhanden sein muss, dass die Rolle enthält. Diese Beschränkung wird auch auf alle anderen Rollen vererbt, also `RoleClass`, `RoleInterace` und `TeamRole`.

Bei der „2“, unter dem Knoten „Top Node Reference“, befindet sich der Knoten „Node Mapping <TeamClass/TeamNode>“. Das „Node Mapping“ enthält die eigentliche Zuordnung der Domänen-Klasse aus dem Metamodell zu der grafischen Notation und dem Werkzeug. Diese Eigenschaften können im Eigenschaften-Fenster („Properties View“) ausgewählt werden, der nicht in der Abbildung zu sehen ist. Die gesetzten Eigenschaften sind aber bis auf das Werkzeug aus dem Knotennamen erkennbar. Bei dem speziellen „Node Mapping“ ist die `TeamClass` einem `TeamNode` zugeordnet. Der `TeamNode` wurde im gmfgraph-

Modell definiert und verweist auf die tatsächliche grafische Notation, in diesem Fall: Rechteck mit Abschnitten und blauer Hintergrundfarbe.

Unterhalb des „Node Mapping“-Knotens befinden sich weitere Kinder-Knoten. Diese umfassen einen „Constraint“-, einen „Feature Seq Initializer“- sowie einen „Feature Label Mapping“-, mehrere „Child Reference“- und „Compartment Mapping“-Knoten. Der Knoten „Constraint“ beschreibt, welche Bedingung erfüllt sein muss, damit eine Instanz von `TeamClass` im Editor erstellt werden kann. Die Bedingung wird hier mit OCL definiert, kann aber ebenso mit Java oder regulären Ausdrücken angegeben werden. Der Vorteil der OCL-Variante ist, dass der Ausdruck direkt im gmfmap-Modell angegeben werden kann. Bei Java ist im generierten Editor die Implementierung einer entsprechenden Methode notwendig. Der zweite genannte Kind-Knoten dient zur Initialisierung von Attributen eines im Editor neu erstellten Teams. Aus der Abbildung kann man erkennen, dass hier das `name`-Attribut initialisiert werden soll. Der dort nicht sichtbare Ausdruck, der das bewerkstelligen soll, ist wiederum in OCL geschrieben. Er setzt den Teamnamen mit „Team“, wobei, wenn schon eine andere Instanz mit demselben Namen vorhanden ist, so wird eine „1“ angehängt. Falls es „Team1“ auch schon gibt, so wird die Instanz „Team2“ genannt usw. Der „Feature Label Mapping“-Knoten ordnet einem Attribut der im „Node Mapping“ angegebenen Domänen-Klasse einem im gmfgraph-Modell definierten Textfeld zu. In diesem Fall wird der Teamname dem Textfeld zugeordnet, welches im ersten Abschnitt enthalten ist (siehe auch 4.6).

Die „Child Reference“-Knoten sind analog zu den „Top Node Reference“-Knoten aufgebaut. Sieht man sich in der Abbildung 4.9 die Kinder-Knoten des „Child Reference“-Knotens mit der „3“ daneben an, so fällt auf, dass diese den bisher besprochenen Knoten sehr ähnlich sind. Daher wird hier auf die weitere Erläuterung dieser Knoten verzichtet. Ansonsten beschreiben die „Child Reference“-Knoten in diesem konkreten Fall die Zuordnungen der in einem Team enthaltenen Elemente wie bspw. Rollen-Klassen oder Attribute. Die Beziehung zwischen Team und Rollen-Klasse im Kontext des Mapping-Modells ist analog zu der oben beschriebenen Beziehung von Zeichenfläche/ `Package` zu `TeamClass`. Ein Team bietet für Rollen ebenfalls eine Art Zeichenfläche an, der nur Rollen hinzugefügt werden können. Für Team-Attribute gilt ähnliches mit dem Unterschied, dass Attribute nicht grafisch sondern rein textuell in einer im Editor gezeichneten `TeamClass`-Instanz dargestellt werden. Die mit „4“ markierten „Compartment Mapping“-Knoten definieren dabei die Zuordnung der im gmfgraph-Modell angesprochenen Abschnitte mit den „Child Reference“-Knoten.

Zu guter Letzt noch ein paar Worte zum „Link Mapping“. Dieser Knoten beherbergt alle Mapping-Informationen zu jenen Metamodellelementen, welche im späteren Editor die Beziehungen zwischen den Klassen wie Teams und Rollen beschreiben. Im OTML-Metamodell gibt es nur eine solche Beziehung und zwar die PlayedBy-Beziehung. Demzufolge ist in der Abbildung (siehe „5“) auch nur eine solche Zuordnung vorhanden.

#### 4.2.2.5 Generierungsmodell (gmfgen)

Das Generierungsmodell in der gmfgen-Datei dient wie das in Abschnitt 3.2.2.3 beschriebene GenModel zur Einstellung von Parametern für den Quellcode-Generator. Die Einstellungen werden in einer Baumansicht präsentiert und sind in verschiedene Gruppen eingeteilt. Im Wurzelelement sind ganz allgemeine Eigenschaften zu finden, etwa ob die späteren Modelle in einem oder in zwei Dateien getrennt nach Modellelementen und deren grafischer Darstellung gespeichert werden sollen, welcher Copyright-Text verwendet werden soll und wie das Java-Package heißen soll, in dem die Quellen generiert werden.

Unter dem Wurzelknoten befindet sich ein Knoten (Gen Plugin) mit Einstellung für das generierte Editor-Plugin. Es lassen sich Plugin-ID, -Name, -Provider und -Version eintragen, aber auch, ob es möglich sein soll, dass der OTModeler Diagramme direkt drucken kann. Diese Option ist übrigens beim UML2Tools-Klassendiagrammeditor abgeschaltet. Für den OTModeler ist die Option aktiviert, das bedeutet, man kann dann auch reine UML-Diagramme mit ihm ausdrucken. Weitere Unterknoten der Wurzel bündeln Eigenschaften zur Benutzung des „Properties View“, der Bereitstellung einer Baumansicht im „Navigator View“ sowie Klassen zur Interpretation der OCL-Ausdrücke.

Daneben existiert noch ein wichtiger Knoten (Gen Diagram), der hauptsächlich die Informationen aus dem Mapping-Modell wiederspiegelt. Diese Informationen befinden sich in weiteren Kinderknoten und sind ergänzt um zusätzliche Einstellungen, etwa welche Klassen zu den jeweiligen Mappings erstellt werden sollen und wie sie heißen. Beispiele für solche Klassen sind EditPart-, EditPolicy- oder ViewFactory-Klassen. Diese bilden zusammen mit jeweils einer Metamodell-Klasse eine semantische Einheit (siehe Abschnitt 4.1.4). In jeder EditPart-Klasse befindet sich zudem eine sogenannte `VisualID`, die in einigen bestimmten Klassen des Editors benutzt wird um Modellelemente zu identifizieren.

Der hier erläuterte „Gen Diagram“-Knoten ist auch der einzige, in dem für den OTModeler Änderungen vorgenommen werden müssen. Diese Anpassungen sind bedingt durch das Problem, das später entsteht, wenn der OTModeler-Kern mit dem UML2Tools-Klassendiagrammeditor zum OTModeler vereinigt werden muss. Da für jeden GMF-Editor standardmäßig eine separate sogenannte EditingDomain existiert, in der die Metamodell-Instanzen enthalten sind, ist es nicht möglich diese miteinander in Beziehung zu setzen. Im Falle des OTModelers kommen solche Instanzen aus dem UML-Metamodell und aus dem OTML-Metamodell. Man muss also für den OTModeler dieselbe EditingDomain benutzen, die auch der UML2Tools-Editor verwendet, was in den Eigenschaften des „Gen Diagram“-Knoten erledigt werden kann. Eine Anpassung des UML2Tools-Editors ist ausgeschlossen, da dieser zwar vom OTModeler wiederverwendet wird, aber trotzdem durch den Benutzer vom Hersteller bezogen werden könnte. Die Strategie zur Anpassung der EditingDomain im Allgemeinen wird in [1] erläutert. Hier können einige Schritte entfallen, da davon ausgegangen wird, dass zwei separate Editoren zum Einsatz kommen. In dieser Arbeit soll nur ein Editor benutzt werden, der Teile des anderen wiederverwendet. Auf jeden Fall müssen die VisualIDs der EditParts in diesem Generierungsmodell so

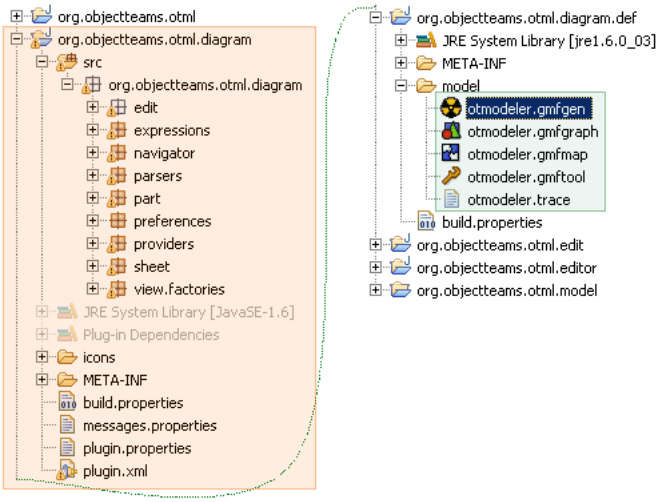


Abbildung 4.10: Übersicht über die generierten Elemente

eingestellt werden, dass sie sich nicht mit den EditPart aus dem UML2Tools-Editor überschneiden. Ein Beispiel für eine Überschneidung ist etwa die VisualID von `PlayedByRelationEditPart` und `GeneralizationEditPart`, die in beiden Fällen standardmäßig den Wert „4001“ besitzt. Code, der Elemente aufgrund ihrer VisualID verwendet, würde nicht mehr korrekt funktionieren. Damit man die VisualIDs nicht von Hand ändern muss und dies jedesmal, wenn das Generierungsmodell neu erstellt wird, kann man sie in einer trace-Datei einstellen. Diese wird bei der Erstellung des Generierungsmodells einmal erzeugt, hat man das „org.eclipse.gmf.bridge.trace“-Plugin installiert. Hier wurde den ursprünglich vergebenen VisualIDs einfach eine 100 vorangestellt, so dass die VisualID von `PlayedByRelationEditPart` nun „1004001“ ist. Weiterhin müssen im generierten Quellcode noch Anpassungen vorgenommen werden, auf die im nächsten Abschnitt eingegangen wird.

#### 4.2.2.6 Quellcode-Generierung

Nachdem in den letzten Abschnitten alle GMF-Modelle laut Abbildung 4.2 definiert wurden, kann nun die Generierung des OTModeler-Kerns erfolgen. Dazu muss man auf den Wurzelknoten des Generierungsmodells rechts-klicken und den Eintrag „Generate UML2Tools Diagrams“ im Kontext-Menü auswählen. Danach wird ein neues Plugin erzeugt, in dem sich die generierten Quellen des OTModeler-Kerns befinden. Abbildung 4.10 gibt einen Überblick über alle Plugins, die hier und bei der Generierung des OTML-Metamodells eine Rolle gespielt haben. Hervorgehoben sind die GMF-Modelle (grün) und die erstellten Quellen (orange).

Jetzt muss der generierte Editor noch so angepasst werden, damit dieselbe EditingDomain verwendet wird wie beim UML2Tools-Editor (siehe voriger Abschnitt letzter Absatz). Laut [1] muss dazu die Methode `createEditingDomain` in der Klasse `OTModelerDocumentProvider` so geändert werden, dass sie eine neue EditingDomain mit derselben ID erzeugt, wie sie im Generierungsmodell

eingetragen wurde. Weitere Änderungen sind nach der Anleitung in der Datei `plugin.xml` durchzuführen. So müssen alle Elemente im Extension-Point `*.elementTypes`, die auch in der `plugin.xml` des UML2Tools-Editors vorkommen, hier von Metamodell-Typ in Spezialisierungs-Typ umdefiniert werden. Konkret betrifft das in diesem Fall nur das Element, in dem das Package behandelt wird.

### 4.2.3 Erstellung des OTModeler

In diesem Abschnitt wird die Implementierung des Adapter-Plugins erläutert. Dieses stellt die eigentliche Vereinigung zwischen dem OTModeler-Kern und dem UML2Tools-Editor her.

#### 4.2.3.1 Werkzeuge

Damit im OTModeler Diagrammelemente erstellt werden können, existieren sogenannte Werkzeuge (Tools). Diese wurden für den OTModeler-Kern im Abschnitt zur Werkzeugdefinition beschrieben und danach generiert. Der OTModeler muss aber nicht nur OTML-Elemente erzeugen können, sondern auch UML-Elemente. Für diese existieren ebenfalls Werkzeuge und eine zugehörige Palette im UML2Tools-Editor, die für den OTModeler wiederverwendet werden können.

GMF bietet dafür sogar einen Extension-Point an, den Palette-Service. In der Extension-Definition muss eine Klasse vom Typ `PaletteFactory.Adapter` angegeben werden, in der die Methode `createTool` entsprechend der bereitzustellenden Werkzeuge implementiert ist. Wenn in anderen Plugins bereits Extensions für den Palette Service definiert sind, können diese hier einfach referenziert werden mit Angabe der dort vorhanden `PaletteFactory.Adapter`-Klasse. Allerdings wird zur Zeit bei der Generierung von GMF-Editoren eine Implementierung für den Palette Service nicht unterstützt. Das heisst, dass sowohl im UML2Tools-Editor als auch im OTModeler-Kern keine solche Klassen bzw. Extensions vorhanden sind.

Um trotzdem eine Palette mit den Werkzeugen aus beiden Plugins im OTModeler zur Verfügung zu stellen, ohne dass der dort vorhandene Code neu implementiert werden muss, bietet sich die Möglichkeit der Wiederverwendung über ObjectTeams/Java an. Die zu adaptierenden Klassen in den jeweiligen Plugins heißen `UMLPaletteFactory` bzw. `OTModelerPaletteFactory`. Dabei dient die `OTModelerPaletteFactory` sozusagen als Einstiegspunkt in die Adaptierung. Dort wird durch den OTModeler-Kern beim Start des Editors die Methode `fillPalette` aufgerufen, die dann eine Instanz vom Typ `PaletteRoot` zurückgibt. In der Instanz sind dann die jeweiligen Werkzeuge als Kinder definiert. Der Aufruf von `fillPalette` wird jedoch per Callin-Methoden-Bindung von der Rolle `OTModelerPaletteFactoryRole` abgefangen. Der Grund dafür ist, dass nicht nur die OTModeler-Werkzeuge vorhanden sein müssen, sondern auch die UML2Tools-Werkzeuge. Das wird in der Callin-Methode durch den Aufruf beider `fillPalette`-Methoden in den jeweils ursprünglichen Klassen mit der Übergabe derselben `PaletteRoot`-Instanz erreicht. Zur Verdeutlichung

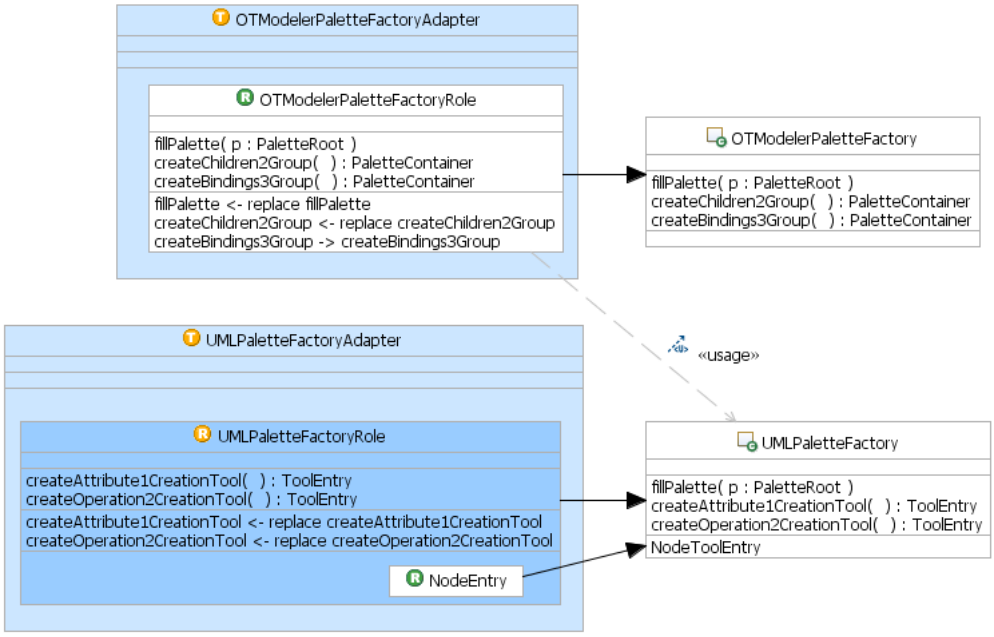


Abbildung 4.11: Adaptierung der Werkzeugpalette

zeigt Abbildung 4.11 die vorhandenen Klassen und deren Beziehung.

Wie man dort sieht, sind weitere Bindungen vorhanden. Der Grund ist, dass zwei Werkzeuge sonst doppelt vorhanden wären, das für Attribute und das für Operationen, da beide sowohl im OTModeler-Kern als auch im UML2Tools-Editor definiert werden. Da ein Werkzeug zur Erstellung verschiedener Elemente geeignet ist, können an dieser Stelle die betreffenden Werkzeuge aus einem der beiden Plugins weggelassen werden. Da ein Werkzeug aber die Elemente „kennt“, die es erstellen kann, muss den verbleibenden Werkzeugen dieses zusätzliche Wissen weitergegeben werden. In diesem Fall wurde sich dazu entschieden, die OTModeler-Werkzeuge wegzulassen und die von ihnen zu erstellenden Elemente in den UML2Tools-Werkzeugen zu definieren. Die nötigen Callin-Bindungen befinden sich dafür in der Rolle `UMLPaletteFactoryRole`. Das nachfolgende Listing zeigt exemplarisch für das Attribut-Werkzeug die dort gemachten Anpassungen:

```
1  callin ToolEntry createAttribute1CreationTool(){
2    List/*<IElementType>*/types = new ArrayList/*<IElementType>*/(10);

4    // These types originate from UMLPaletteFactory:
5    types.add(UMLElementTypes.Property_3001);
6    types.add(UMLElementTypes.Property_3019);
7    types.add(UMLElementTypes.Property_3014);
8    types.add(UMLElementTypes.Property_3021);
9    types.add(UMLElementTypes.Property_3023);
10   types.add(UMLElementTypes.Property_3028);

12   // These types originate from OTModelerPaletteFactory:
13   types.add(OTModelerElementTypes.Property_1003002);
14   types.add(OTModelerElementTypes.Property_1003008);
15   types.add(OTModelerElementTypes.Property_1003014);
```

```
16    types.add(OTModelerElementTypes.Property_1003019);

18    ToolEntry entry = new NodeEntry(Messages.
          Attribute1CreationTool_title, Messages.
          Attribute1CreationTool_desc, types);
19    entry.setSmallIcon(UMLElementTypes.getImageDescriptor(
          UMLElementTypes.Property_3001));
20    entry.setLargeIcon(entry.getSmallIcon());
21    return entry;
22  }
23  createAttribute1CreationTool <- replace createAttribute1CreationTool;
```

Der in der Methode verwendete Quelltext wurde aus der adaptierten Methode kopiert und um das Hinzufügen der Element-Typen aus dem OTModeler ergänzt. Der Grund dafür ist, dass die Liste von unterstützten Element-Typen eines Werkzeugs (`NodeEntry`) nur beim Erzeugen eines solchen zugewiesen und nicht im Nachhinein geändert werden kann. Das Hinzufügen der OTModeler-Element-Typen ist also nicht über die von der adaptierten Methode zurückgegebene `NodeEntry`-Instanz möglich. Das wäre prinzipiell besser gewesen, da kopierter Code immer nachgepflegt werden muss, wenn sich das Original ändert.

Zuletzt muss noch das Hizufügen des Attribut- bzw. Operation-Werkzeugs zur OTModelerPalette durchgeführt werden. Das geschieht in der Callin-Methode `createChildren2Group` der Rolle `OTModelerPaletteFactoryRole`, welche die adaptierte Methode ersetzt. Im Prinzip wurde wieder der Code kopiert, und die dortigen Aufrufe der Methoden, welche für die relevante Werkzeugerstellung zuständig sind, weggelassen:

```
1  callin PaletteContainer createChildren2Group(){
2    PaletteGroup paletteContainer = new PaletteGroup(
3        Messages.Children2Group_title);
4    //paletteContainer.add(createAttribute1CreationTool());
5    //paletteContainer.add(createOperation2CreationTool());
6    paletteContainer.add(createBindings3Group());
7    return paletteContainer;
8  }
9  createChildren2Group <- replace createChildren2Group;
```

Das Ergebnis der Vereinigung beider Paletten ist in Abbildung 4.8 zu sehen. Dabei stammt der obere Teil aus der OTModeler-Palette und der untere (ab dem „Class“-Eintrag) aus der UML2Tools-Palette.

#### 4.2.3.2 View-Elemente

Wie in 4.1.4 beschrieben wurde, bilden die View-Elemente die Brücke zwischen dem Ecore-Metamodell eines GMF-Editors und dem auf GEF basierenden Teil (EditParts, etc.). Die Erstellung neuer View-Objekte wird zur Laufzeit über sogenannte View-Provider geregelt. Das sind Klassen, die einen bestimmten Typ haben müssen und über eine Extension-Definition in der `plugin.xml` referenziert werden. Der betreffende Extension-Point ist der View-Service, der vom GMF bereitgestellt wird. Es können zur Laufzeit mehrere View-Provider existieren, wobei dann alle der Reihe nach „gefragt“ werden, ob sie das geforderte View-Element erstellen können. In diesem Fall existieren zwei View-Provider, der `UMLViewProvider` aus dem UML2Tools-Editor und der `OTModelerViewProvider` aus dem generierten OTModeler-Kern.

Beide liefern in den jeweiligen Methoden entweder eine `Diagram`-, `Node`- oder `Edge`-Instanz zurück. Ein Blick in die Methode `getNodeViewClass`, zeigt aber, dass dort ein Problem existiert. In dieser Methode wird geprüft, ob die ModelID eines übergebenen Container-View-Elementes mit der ModelID des View-Provider-Kontexts übereinstimmt. Ist das nicht der Fall wird keine Node-Instanz zurückgegeben. Die ModelID ist für jeden GMF-Editor eindeutig und eine öffentliche Konstante im EditPart der Zeichenfläche. Konkret befindet sich die ModelID hier im `PackageEditPart`. Wenn also die ModelID des übergebenen Containers eine andere ist, so gehört der Container demnach zu einem anderen GMF-Editor. Daraus folgt, dass der `UMLViewProvider` keine View-Elemente für den OTModeler erstellt.

Es gibt aber eine Möglichkeit, wie man trotzdem im OTModeler die View-Elemente aus dem UML2Tools-Editor benutzen kann. Dazu muss die Methode `getModelID` in der Klasse `UMLVisualIDRegistry` abgeändert werden, welche vom `UMLViewProvider` genutzt wird, um die ModelID des Container-Objekts zu ermitteln. Die Methode wird dazu mit Hilfe von OT/J per Callin-Replace-Bindung in `UMLVisualIDRegistryRole` adaptiert und liefert nun als Rückgabewert die ModelID des `PackageEditParts` aus dem UML2Tools-Editor.

Abbildung 4.12 zeigt die Klasse `UMLViewProvider` sowie die Benutzung der Klasse `UMLVisualIDRegistry`. Die relevanten Teams und Rollen sind ebenfalls eingezeichnet. Die Methoden-Signaturen wurden dabei aus Übersichtsgründen weggelassen.

Die Aufgaben der `UMLVisualIDRegistry`-Klasse bestehen zusätzlich darin, außer die ModelID zu einem View-Element zu ermitteln, die VisualIDs zu den einzelnen Modellelementen zu liefern und zu überprüfen, ob ein Modellelement zu einem anderen Modellelement als Kind hinzugefügt werden kann. Durch die Adaptierung der Methode `getModelID` ist aber jetzt der Fall eingetreten, dass der `PackageEditPart` aus dem UML2Tools-Projekt tatsächlich als EditPart der OTModeler-Zeichenfläche benutzt wird. Aus diesem Grund existieren auch die anderen Rollen-Methoden in `UMLVisualIDRegistryRole`. Diese erlauben es indirekt, dass der `PackageEditPart` die EditParts aus dem OTModeler-Kern enthalten kann. Dazu werden einige Methodenaufrufe in der `UMLVisualIDRegistry` per Callin-Replace-Bindung abgefangen und gegebenfalls über die Rolle `OTModelerVisualIDRegistryRole` an die `OTModelerVisualRegistry` aus dem OTModeler-Kern delegiert. Normalerweise sind an dieser Stelle in OT/J nicht zwei Rollen in jeweils separaten Teams notwendig. Stattdessen könnte man auch beide Rollen in einem Team unterbringen. Da hier die Adaptierung aber über Plugin-Grenzen hinweg erfolgt, müssen die Teams mittels *Aspect Binding* an konkrete Base-Plugins gebunden werden. Das Problem besteht darin, dass die jeweils adaptierten Klassen in verschiedenen Plugins liegen und so zwei Teams notwendig sind.

#### 4.2.3.3 Diagramm-Knoten

Mit den bisher gemachten Anpassungen ist es bereits möglich, dass der OTModeler UML-Modellelemente darstellen kann, so wie der UML2-Tools-Editor. Durch die im vorigen Abschnitt durchgeführte Adaptierung der `UMLVisualID-`

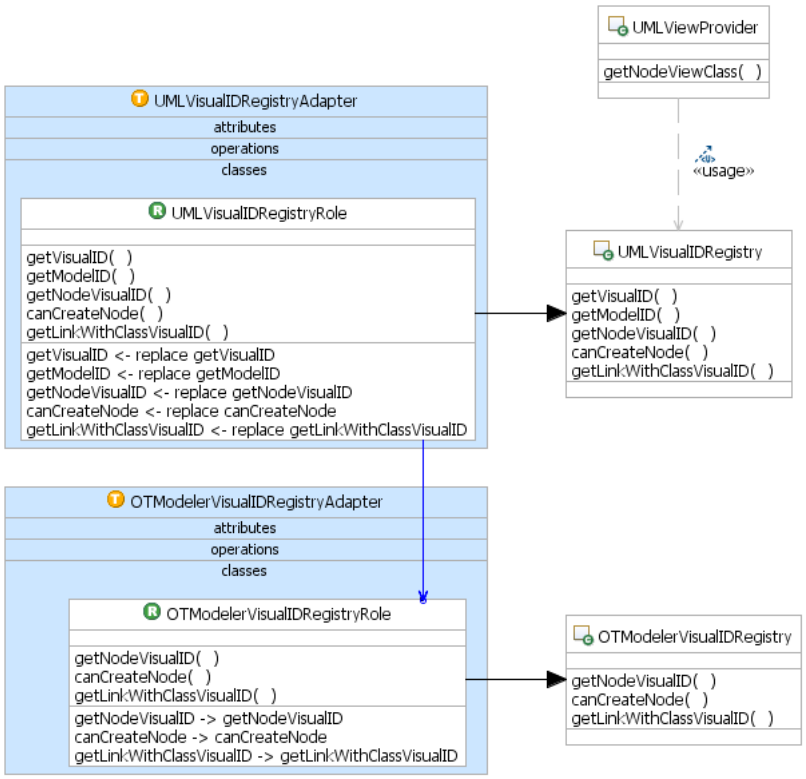


Abbildung 4.12: Adaptierung der UMLVisualIDRegistry

`Registry`-Klasse wird aber nun nicht mehr der OTModeler-`PackageEditPart` benutzt, weswegen keine Teams mehr erstellt werden können.

Generell entscheidet ein EditPart, welche Elemente er enthalten kann, indem er bei bestimmten Requests (`CreateElementRequest`) entsprechende `Command`-Objekte zurückgibt. Diese können ausgeführt werden und führen dann die eigentliche Erstellung des neuen EditParts durch, falls dieser im ersten enthalten sein darf. Die Entscheidung, ob ein EditPart einen anderen enthalten kann, wird an die ihm zugehörige *ItemSemanticEditPolicy* delegiert. Beim `PackageEditPart` ist das konkret die `PackageItemSemanticEditPolicy`. Diese Policy enthält die Methode `getCreateCommand`, in der anhand des übermittelten Requests geprüft wird, für welchen neuen EditPart ein `Command`-Objekt zurückgegeben werden soll. Das nachfolgende Listing zeigt die `getCreateCommand`-Methode aus dem OTModeler-`PackageEditPart`:

```
1  protected Command getCreateCommand(CreateElementRequest req) {
2    if (OTModelerElementTypes.TeamClass_1002001 == req.getElementType())
         {
3      if (req.getContainmentFeature() == null) {
4        req.setContainmentFeature(UMLPackage.eINSTANCE.
             getPackage_PackagedElement());
5      }
6      return getGEFWrapper(new TeamClassCreateCommand(req));
7    }
8    return super.getCreateCommand(req);
9  }
```

Diese Methode wird durch die Adaptierung im vorigen Abschnitt gar nicht mehr verwendet, da nun der UML2Tools-`PackageEditPart` und somit dessen ItemSemanticEditPolicy benutzt wird. In deren `getCreateCommand`-Methode kann kein `Command`-Objekt für ein Team-Element erstellt werden. Damit dies dennoch möglich ist, wird die Methode mittels OT/J adaptiert:

```
1  final OTModelerPackageItemSemanticEditPolicyAdapter adapter = new
       OTModelerPackageItemSemanticEditPolicyAdapter();
2  private OTModelerPackageItemSemanticEditPolicyRole<@adapter> policy =
       adapter.getRole();

4  public class UMLPackageItemSemanticEditPolicyRole playedBy
       PackageItemSemanticEditPolicy {

6    callin Command getCreateCommand(CreateElementRequest req) {
7      Command cmd = base.getCreateCommand(req);

9      if (cmd == null) {
10       cmd = policy.getCreateCommand(req);
11     }
12     return cmd;
13   }
14   getCreateCommand <- replace getCreateCommand;

16 }
```

Der Aufruf von `getCreateCommand` wird per Callin-Replace-Bindung abgefangen. Zunächst wird jedoch die Original-Implementierung aufgerufen. Führt das zu keinem Ergebnis, wird der Aufruf an die `getCreateCommand`-Implementierung in der `PackageItemSemanticEditPolicy` aus dem OTModeler-Kern delegiert. Wenn der Request die Erstellung eines Teams beinhaltet, wird jetzt das entsprechende `Command`-Objekt zurückgeliefert.

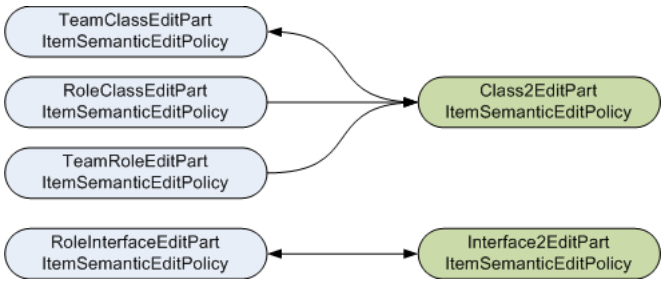


Abbildung 4.13: Äquivalenz von Elementen bzgl. ihrer Beziehungen

Von der Verwendung des EditPolicyService wurde hier abgesehen, da pro EditPart nur eine EditPolicy vom selben Typ vorhanden sein darf. Wird eine neue EditPolicy beim EditPart registriert, so wird die alte vorhandene verworfen. Ein einfaches Hinzufügen der `PackageItemSemanticEditPolicy` aus dem OTModeler-Kern hätte also nicht ausgereicht. Stattdessen hätte eine neue EditPolicy geschrieben werden müssen, die sämtliche Funktionalität der beiden urspünglich vorhandenen vereint. Dieser Vorgang wäre sehr viel komplizierter gewesen als die hier benutzte Adaptierung, so dass der Extension-Point nicht benutzt wird.

#### 4.2.3.4 Diagramm-Links

Da nun UML-Modellelemente und OTML-Modellelemente als Diagramm-Knoten dargestellt werden können, sollen diese auch Beziehungen miteinander eingehen können. Dabei definiert die OTML nur eine Beziehung zwischen Rollen und `Classifier`-Klassen, womit die PlayedBy-Relation gemeint ist. Aus der UML kommen ungleich mehr Beziehungsarten. An welchen Beziehungen ein Element beteiligt sein kann, wird letztendlich im Metamodell entschieden. Da Teams Klassen sind, genau wie Rollen-Klassen und Team-Rollen auch, können sie auch dieselben aus der UML stammenden Beziehungen eingehen. Das gleiche gilt für Rollen-Interfaces und Interfaces. Weiterhin sollen UML-Klasse und UML-Interface auch Ziel einer PlayedBy-Beziehung sein können.

Was im Metamodell bereits definiert wurde, ist für die generierten GMF-Editoren zur Laufzeit nicht möglich. Das liegt daran, dass beispielsweise in den GMF-Modellen des OTModeler-Kerns außer der PlayedBy-Beziehung keine weiteren Beziehungen vorkommen. Dadurch kann etwa ein `TeamClassEditPart` auch nicht Teil einer Assoziation sein. Ähnlich wie im vorangegangenen Abschnitt wird auch hier in der *IteamSemanticEditPolicy* des jeweiligen EditParts darüber entschieden, ob er eine Beziehung eingehen kann. Und genauso lässt sich auch hier die Implementierung aus den entsprechenden EditPolicies wiederverwenden. Abbildung 4.13 zeigt, welche konkrete EditPolicy eine andere ergänzt.

Dabei werden in der Umsetzung mit OT/J für jeden Pfeil zwei Teams mit jeweils einer Rolle benötigt. Bspw. soll ein Interface das Ziel einer PlayedBy-Beziehung werden. Der EditPart des Interfaces ruft dann in der Klasse `Interface2ItemSemanticEditPolicy` die entsprechende Methode auf, welche das

Command-Objekt zurückgeben soll. Der Aufruf wird von UMLInterface2ItemSemanticEditPolicyRole abgefangen und über OTModelerRoleInterfaceItemSemanticEditPolicyRole an eine RoleInterfaceItemSemanticEditPolicy delegiert. Nebenbei könnte nicht nur an dieser Stelle über ein Refactoring der Klassennamen nachgedacht werden.

Interessant sind die Wiederverwendungsbeziehungen, die in beide Richtungen gehen. Dort existieren in einer Rolle für eine adaptierte Basis-Klassenmethode jeweils eine Callin- und eine Callout-Beziehung, wie z.B. für die Methode getCreateRelationshipCommand in der Klasse UMLInterface2ItemSemanticEditPolicyRole. Dabei wird in einer Callin-Methode von Rolle A die Callout-Methode der Rolle B aufgerufen, wodurch indirekt wieder die Callin-Methode von B aufgerufen wird, die wiederum die Callout-Methode von A aufruft usw. Es besteht also die mögliche Gefahr einer Endlosschleife. Zur Laufzeit kann dieser Fall auftreten, wenn ein Erstellungs-Request einen Beziehungstyp enthält, der von beiden EditPolicies nicht unterstützt wird. Zur Vermeidung einer Endlosschleife wird vor dem Asuführen eines Callouts ein Flag gesetzt, der die sonst erfolgende Callin-Bindung deaktiviert. Das Flag wird dazu in einem Base-Guard für die Rolle geprüft. Hier ist ein Ausschnitt aus der Implementierug des Teams UMLInterface2ItemSemanticEditPolicy, wobei die Bezeichner teilweise geändert wurden, um die Lesbarkeit zu verbessern:

```
1  public team class UMLInterface2ItemSemanticEditPolicy {

3    private static boolean isDoingCallout = false;

5    private OtherRole<@OtherTeam.INSTANCE> externalRole =
6      OtherTeam.INSTANCE.getRole();

8    ...

10   /* Callin Bindings only get active,
11    * if they are not triggered by own Callouts */
12   public class EditPolicyRole playedBy
         Interface2ItemSemanticEditPolicy
13   when (false == UMLInterface2ItemSemanticEditPolicy.isDoingCallout)
14   {
15     ...

17     /* Return requested command from base class or try
18      * to get it from externalRole's base. */
19     callin Command callinGetCreateRelationshipCommand(Request req) {
20       Command command = base.callinGetCreateRelationshipCommand(req);

22       if (command == null) {
23         command = externalRole.getCreateRelationshipCommand(req);
24       }
25       return command;
26     }
27     callinGetCreateRelationshipCommand <- replace
           getCreateRelationshipCommand;


30     public Command getCreateRelationshipCommand(Request req) {

32       startCallout(); // set isDoingCallout = true
33       Command cmd = calloutGetCreateRelationshipCommand(req);
34       endCallout();   // set isDoingCallout = false

36       return cmd;
```

```
37      }
38      Command calloutGetCreateRelationshipCommand(Request req) ->
39        Command getCreateRelationshipCommand(Request req);
40    }

42    ...
43  }
```

Auch in diesem Fall hätte man analog zum vorhergehenden Abschnitt über den EditPolicyService neue EditPolicies an die EditParts binden können. Und auch hier wurde sich aufgrund des erheblichen Mehraufwands gegen die Benutzung des Service entschieden.

#### 4.2.3.5 Neudarstellung von Diagramm-Elementen

Sowohl beim Öffnen eines gespeicherten Modells als auch beim Löschen von Elementen in einem Modell wird der Inhalt des Container-Elements neu dargestellt. In diesem Zusammenhang wird auch geprüft, ob enthaltene Elemente noch gültig sind. Ist das nicht der Fall, werden diese aus dem Editor entfernt. Zuständig für die Bearbeitung von Requests zur Neudarstellung von Container-Inhalten sind die sogenannten *CanonicalEditPolicies*. Jeder EditPart, der andere EditParts enthalten kann, besitzt in der Regel eine CanonicalEditPolicy. Die Logik, ob ein EditPart andere EditParts enthalten kann, ist in GMF allerdings größtenteils in die DiagramUpdater-Klasse verlagert.

Die Anpassungen, die in den beiden vorangegangenen Abschnitten durchgeführt wurden, finden in den entsprechenden Methoden der jeweiligen DiagramUpdater-Klassen ihre Fortsetzung. So werden in der Klasse `UMLDiagramUpdater` die für `Package`, `Class` und `Interface` relevanten Methoden angepasst sowie in `OTModelerDiagramUpdater` die zu `TeamClass`, `RoleClass`, `RoleInterface` und `TeamRole` gehörenden Methoden. In Abbildung 4.14 sieht man, dass für beide DiagramUpdater-Klassen jeweils eine Rolle erstellt wurde. Diese fangen die Aufrufe der oben erwähnten Methoden mittels Callin-Replace-Binding ab. Der Rückgabewert der Callin-Methoden enthält dabei Werte, die aus beiden DiagramUpdater-Klassen stammen und die konkrete Element-Typen darstellen. Z.B. gibt die Methode `getPackage_SemanticChildren` in `UMLDiagramUpdater` eine Liste von Element-Typen aus dem UML2Tools-Editor zurück. Darin sind u.a. die Element-Typen zu den Metamodellelementen `Class`, `Interface`, usw. Diese Element-Typen werden, um den Element-Typ zur `TeamClass` ergänzt, indem die Rückgabewerte der `getPackage_SemanticChildren`-Methode aus der Klasse `OTModelerDiagramUpdater` sowie aus der Klasse `UMLDiagramUpdater` miteinander vereinigt werden. Die Vereinigung selbst wird mit Hilfe der Helfer-Klasse `DiagramUpdaterUtil` durchgeführt. Die Typen der in den Listen enthaltenen Objekte sind nämlich nicht kompatibel. Das gleiche Verfahren wird analog für die anderen Methoden angewendet.

#### 4.2.3.6 Feature-Darstellung

In diesem letzten Abschnitt zur Erstellung des OTModelers soll noch auf die Formatierung der Features eingegangen werden, also die Darstellung von Attributen, Operationen und Bindungs-Definitionen. Wenn man sich die bisher

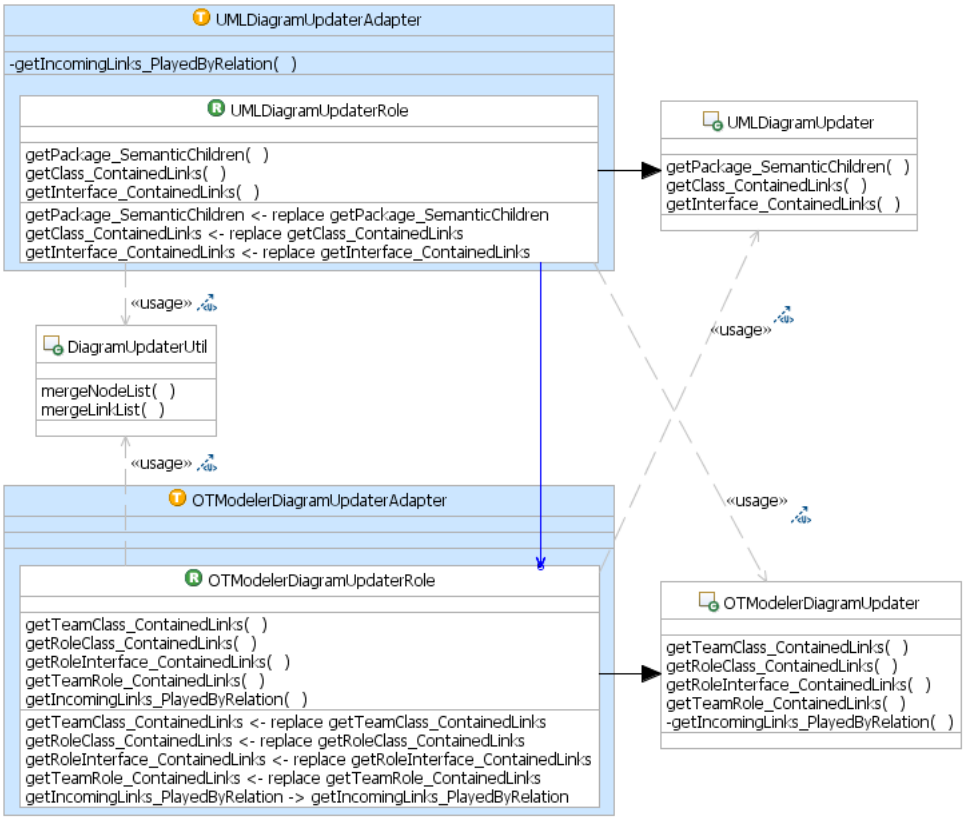


Abbildung 4.14: Adaptierung der DiagramUpdater-Klassen (Methoden-Signaturen wurden weggelassen und Bezeichner gekürzt)

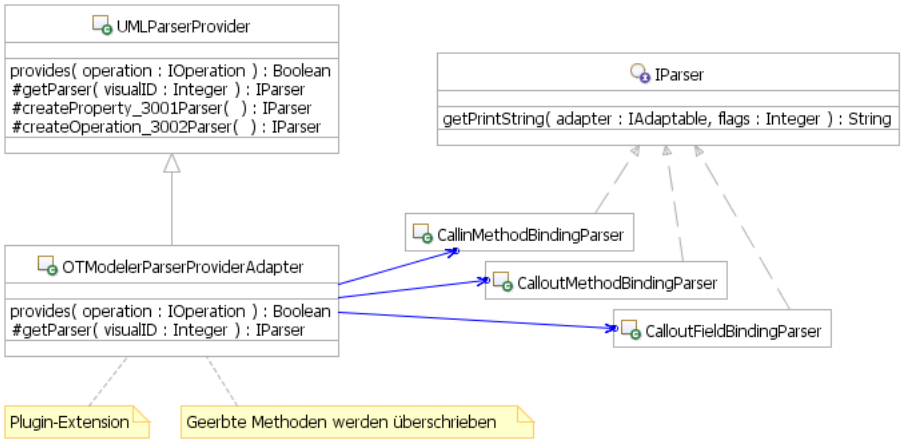


Abbildung 4.15: Erstellung eines neuen ParserProviders

abgedruckten Klassendiagramme hier anschaut, so stellt man fest, dass alle Features in einem speziellen Format wiedergegeben werden. Attribute werden bspw. mit Sichtbarkeit, Name sowie Typ und Operationen mit Sichtbarkeit, Name, Parameter-Definitionen und Typ des Rückgabewert dargestellt. In Abbildung 4.15 ist die Formatierung von Operationen schön zu erkennen.

Eine formatierte Darstellung ist dabei nicht selbstverständlich in einem GMF-Editor vorhanden, da die Reihenfolge und Art der darzustellenden Informationen durch keine Editor-übergreifende Regel festgelegt werden kann. Allerdings ist es möglich eigene Ausgabeformate für Textfelder vor der Generierung im Mapping-Modell zu definieren. Die Implementierung des Ausgabeformats wird dann automatisch generiert. Die Formate werden in den Eigenschaften der „Feature Label Mapping“-Knoten z.B. mit Hilfe von regulären Ausdrücken formuliert. Es können dabei nur Attribute des zum „Feature Label Mapping“ gehörenden Fetures ausgewählt werden. Weiterhin sind nicht alle Attribute verfügbar sondern nur diejenigen, die einen primitiven Datentyp haben. Die Festlegung einer Regel für Callin-Methoden-Bindungen ist so unmöglich, da die Namen von Rollen- und Basis-Methoden nicht zur Verfügung stehen.

Eine Untersuchung des UML2Tools-Mapping-Modells des Klassendiagrammeditors zeigt, dass auch dort keine Ausgabeformate für Attribute und Operationen vorhanden sind. Stattdessen wird die Darstellung der Strings über Parser-Klassen formatiert, die von den EditParts benutzt werden. Ein EditPart hat aber keine feste Beziehung zu seinem Parser-Objekt, sondern fordert diese von sogenannten Parser-Providern an. Da Parser-Provider über den Extension-Point-Mechansimus von Eclipse zur Laufzeit zur Verfügung gestellt werden, kann man auch ohne neue Generierung die Ausgabeformate durch Einbinden eines neuen Parser-Providers ermöglichen. Die Schnittstelle eines Parser-Providers sieht dabei zwei Methoden vor. Die eine (`provides`) soll darüber Auskunft geben, ob ein konkreter Parser bereitgestellt werden kann, und die andere (`getParser`) führt die Bereitstellung durch.

In diesem Fall wurde ein solcher Parser-Provider (`OTModelerParserProvider-`

`Adapter`) erstellt. Dieser wurde vom Parser-Provider des UML2Tools-Editors abgeleitet. Die zur Schnittstelle gehörenden Methoden wurden dabei überschrieben, so dass der `OTModelerParserProviderAdapter` nun Parser für Operationen, Attribute und Bindungen bereit stellt, wenn diese zu einem OTModeler-Element gehören. Die Parser für Operationen und Attribute werden aus dem UML2Tools-Editor wiederverwendet und müssen nicht neu implementiert werden. Dazu werden die relevanten Methoden (`createProperty_3001Parser`, `createOperation_3002Parser`) in der Superklasse des `OTModelerParserProviderAdapter` aufgerufen.

Für die Methoden-Bindungen gibt keine fertigen Parser, so dass diese selbst implementiert werden müssen. Dafür wurde ein relativ simpler Ansatz gewählt. Das folgende Listing zeigt, wie die Formatierung des Ausgabestrings für eine Callin-Methoden-Bindung in der Klasse `CallinMethodBindingParser` umgesetzt ist:

```
1  public String getPrintString(IAdaptable adapter, int flags) {
2    StringBuffer s = new StringBuffer();
3
4    EObject element = (EObject) adapter.getAdapter(EObject.class);
5    if (element != null) {
6      if (element instanceof CallinMethodBinding) {
7
8        // get domain element
9        CallinMethodBinding callinBinding = (CallinMethodBinding)
             element;
10
11       // append name of associated role method
12       if (callinBinding.getRoleMethod() != null) {
13         s.append(callinBinding.getRoleMethod().getName());
14       } else {
15         s.append("UNDEFINED");
16       }
17
18       // append arrow
19       s.append(" <- ");
20
21       // append callin modifier
22       s.append(callinBinding.getCallinModifier().toString().
             toLowerCase());
23       s.append(" ");
24
25       // append name of associated base method
26       // (TODO: support multiple base methods)
27       if (callinBinding.getBaseMethods().size() > 0) {
28         s.append(callinBinding.getBaseMethods().get(0).getName());
29       } else {
30         s.append("UNDEFINED");
31       }
32     }
33   }
34   return s.toString();
35 }
```

# 5 Zusammenfassung

Hier folgt eine Zusammenfassung der Diplomarbeit in Hinblick auf die erreichten Ergebnisse sowie deren Bewertung. Anschließend werden im Ausblick noch an diese Arbeit anschließende Tätigkeiten erörtert.

## 5.1 Fazit

Im Rahmen der Diplomarbeit wurde ein Metamodell für die Modellierungssprache OTML entwickelt. Die OTML soll dabei die Grundlage für die spätere Umsetzung eines Diagrammeditors für OT/J-Klassendiagramme bilden. Erreicht wurde, dass in der OTML alle statischen Konzepte berücksichtigt und die wichtigsten detailliert modelliert sind, so dass sie in der jetzigen Version bereits von praktischem Nutzen sein kann.

Es fehlt aber noch ein detaillierter Entwurf zu einigen Konzepten, die aus der OTJLD ermittelt werden konnten und bislang nicht richtig unterstützt werden. Dazu zählen insbesondere das Result- und Parameter-Mapping sowie eine bessere Umsetzung von Guards. Ebenfalls müssen noch viele Constraints formuliert werden, die bereits identifiziert wurden.

Darüberhinaus wurde das OTML-Metamodell mit Hilfe von EMF in Java-Quellcode transformiert. Damit wurde es erst möglich, dass es von anderen Programmen wie dem OTModeler benutzt werden kann. Es wurde gezeigt, welche Schritte für die Transformation bzw. für die Generierung des Codes notwendig sind und wie die Anpassung des generierten Codes durchgeführt wird.

Im zweiten Teil der Diplomarbeit wurde die Erstellung eines grafischen Editors mit Hilfe von modellgetriebenen und aspektorientierten Ansätzen erläutert. Der OTModeler baut auf zwei GMF-Editoren auf. Der eine ist der Klassendiagramm-Editor aus dem UML2Tools-Projekt. Der andere wurde hier als OTModeler-Kern bezeichnet. Während der UML2Tools-Editor dazu geeignet ist UML-Klassendiagramme zu bearbeiten, beschränkt sich der OTModeler-Kern rein auf Klassendiagramme, die Elemente aus ObjectTeams/Java enthalten. Durch GMF-Erweiterungsmechanismen, aber hauptsächlich durch ObjectTeams/Java, wurde die Funktionalität der beiden ursprünglichen Editoren im OTModeler vereinigt. In diesem können nun sowohl Elemente der UML als auch der OTML in Diagrammen verwendet und auch miteinander in Beziehung gesetzt werden. Die Klassendiagramme, die in dieser Diplomarbeit dargestellt wurden, sind alle mit Hilfe des OTModelers erstellt worden.

Aber auch wenn man mit dem OTModeler schon OTML-Modelle beschreiben kann, so gibt es noch viele Fehler zur Laufzeit, die darauf schließen lassen, dass weitere Teile adaptiert werden müssen, die bis jetzt noch nicht ermittelt wurden. Dies ist auch das größte Handicap bei der gewählten Entwicklungsstrategie. Zur Adaptierung ist es schlicht notwendig, zunächst die Konzepte

aus EMF und GMF zu erarbeiten, was durch teilweise fehlende Dokumentation nicht einfach ist. Dazu kommt, dass man auch die Umsetzung der Konzepte auf Implementierungsebene genau verstehen muss. Das bedeutet, dass nicht nur der Quelltext von allen Klassen, die adaptiert werden sollen, analysiert werden muss, sondern auch meist derjenigen Klasse, die in irgendeiner Beziehung zu ihnen stehen. Daraus können komplexe Probleme entstehen und man verliert leicht den Überblick.

Die Verwendung der GMF-Extension-Points stellte zwar manchmal eine Alternative zur Entwicklung des OTModelers dar, sie sind aber ebenfalls nur leidig dokumentiert, fordern vom Entwickler mehr Aufwand bei der Implementierung und sind teilweise erst gar nicht einsetzbar. Insgesamt bleibt der Eindruck, dass das Konzept der GMF-Extension-Points vom Ansatz her gut durchdacht ist, in der Praxis aber halbherzig umgesetzt wurde.

Der positive Aspekt der Entwicklungsstrategie ist, dass mit minimalem Aufwand an selbst geschriebenem Quelltext eine maximale Wiederverwendung der vorhandenen, generierten Quellen erreicht wird. Das ist nicht zuletzt ein Verdienst von ObjectTeams/Java, da es durch diese Sprache auch möglich ist, statische und private Member wiederzuverwenden.

## 5.2 Ausblick

Die hier entwickelte OTML wurde mit Schwerpunkt auf die statischen Elemente von ObjectTeams/Java entworfen. Abgesehen von einigen wenigen Unfertigkeiten stellt sie bereits eine gute Grundlage für weitere Entwicklungen dar. So könnte man z.B. Generatoren bauen, damit OTML-Modelle in andere Modelle wie Quellcode transformiert werden können. Andererseits wäre auch die Transformation von OT/J-Quellcode oder UML-Modellen in ein OTML-Modell denkbar. Die Definition von Transformationen wäre in Hinblick auf die in der Einleitung formulierten Ausführungen zur MDA sicherlich sinnvoll.

Außerdem könnte sie um dynamische Elemente erweitert werden, damit auch diese modelliert werden können. Dabei ist die Erweiterung mit Hilfe des UML2-Plugins sehr einfach und die Umsetzung in Java-Quellcode funktioniert ohne Probleme. Mit den dynamischen Elementen ließen sich auch detaillierte Modelle erstellen, die die Grundlage für eine vollständigere Generierung von Anwendungen bilden könnte. Insgesamt muss noch ein separates Dokument zur Verfügung gestellt werden, in dem die Spezifikation der OTML enthalten ist.

Die Umsetzung des OTML-Metamodells als UML-Profil ist ein weiterer Punkt, der wünschenswert ist. Dadurch könnten auch OTML-Diagramme in anderen UML-Editoren als dem OTModeler erstellt werden. Zwar wäre die Darstellung dann weniger übersichtlich, da Elemente nur durch Stereotypen annotiert wären. Andererseits würde eine größere Anzahl von Benutzern in der Lage sein, OTML-Diagramme zu bearbeiten, da kein spezieller UML-Editor wie der OTModeler notwendig ist.

Der OTModeler ist im Gegensatz zur OTML noch unausgereift. Einerseits sind Fehler vorhanden, andererseits fehlt noch einiges an Funktionalität, damit man ihn als sinnvolles Werkzeug zur Modellierung von ObjectTeams/Java-

Programmen einsetzen kann. Allerdings hat er gezeigt, wie die Umsetzung der Kombination zweier Editoren zu einem neuen Editor erfolgen kann.

Die Erstellung des OTModelers könnte dabei auch in Zukunft mit Hilfe eines Generators erfolgen, der ähnlich wie der GMF-Generator aus einem Modell Quell-Code erzeugt. Dieses Modell müsste dann die Konzepte aus zwei Mapping-Modellen miteinander verknüpfen. Aus der Verknüpfung ließe sich bspw. OT/J-Code erzeugen, der die entsprechenden Adaptierungen realisiert. Mit einem solchen Generator wäre es auch möglich andere GMF-Editoren als den OTModeler zu erstellen, die nach dem gleichen Prinzip einen schon vorhandenen GMF-Editor um eigene Konzepte erweitern.

# Literaturverzeichnis


[1] Bokowski, B. ; Niemiec, S. ; Shatalin, A. u. a.: *Sharing single EditingDomain instance across several diagrams.* http://wiki.eclipse.org/GMF_Tips#Sharing_single_EditingDomain_instance_across_several_diagrams

[2] Bruck, J. ; Hussey, K. : *Customizing UML: Which Technique is Right for You?* http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html

[3] Bruck, J. ; Hussey, K. : *Extending UML2: Creating Heavy-weight Extensions.* https://bugs.eclipse.org/bugs/attachment.cgi?id=55337

[4] Buschmann, F. ; Meunier, R. ; Rohnert, H. ; Sommerlad, P. : *Pattern-Oriented Software Architecture. A System of Patterns.* Bd. 1. Wiley & Sons, 1996

[5] Eclipse Foundation, Inc.: *AMW Homepage.* http://www.eclipse.org/gmt/amw/

[6] Eclipse Foundation, Inc.: *Eclipse Homepage.* http://www.eclipse.org

[7] Eclipse Foundation, Inc.: *EMF Homepage.* http://www.eclipse.org/emf/

[8] Eclipse Foundation, Inc.: *GEF Homepage.* http://www.eclipse.org/gef/

[9] Eclipse Foundation, Inc.: *GMF Homepage.* http://www.eclipse.org/gmf/

[10] Eclipse Foundation, Inc.: *UML2Tools Homepage.* http://www.eclipse.org/modeling/mdt/?project=uml2tools#uml2tools

[11] eclipse.modeling.mdt.uml2 Newsgroup: *UML and multiple/diamond inheritance.* news://news.eclipse.org/fb2a50$e4k$1@build.eclipse.org

[12] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J. : *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, 1996

[13] Gaufillet, P. u. a.: *Topcased Homepage.* http://www.topcased.org/

[14] Gérard, S. u. a.: *Papyrus UML Homepage*. http://www.papyrusuml.org

[15] Herrmann, S. : *ObjectTeams Homepage*. http://www.objectteams.org

[16] Herrmann, S. : Object Teams: Improving Modularity for Crosscutting Collaborations. In: *Proc. of Net.ObjectDays* (2002). http://www.objectteams.org/publications/NODe02.pdf

[17] Herrmann, S. ; Hundt, C. ; Mosconi, M. : ObjectTeams/Java Language Definition / ObjectTeams.org. Version: 1.0. http://objectteams.org/publications/OTJLDv1.0.pdf (2007/03). – Technical Guide

[18] Herrmann, S. ; Mosconi, M. : Integrating Object Teams and OSGi: Joint Efforts for Superior Modularity. In: *Journal of Object Technology* 6 (2007), October, Nr. 9, 105-125. http://www.jot.fm/issues/issue_2007_10/paper6.pdf

[19] IBM Corporation u. a.: *Developer Guide to Diagram Runtime Framework*. http://help.eclipse.org/help33/topic/org.eclipse.gmf.doc/prog-guide/runtime/Developer%20Guide%20to%20Diagram%20Runtime.html. Version: 2005

[20] IBM Corporation u. a.: *Developer's Guide to the Extensible Type Registry*. http://help.eclipse.org/help33/topic/org.eclipse.gmf.doc/prog-guide/runtime/Developers%20Guide%20to%20the%20Extensible%20Type%20Registry/Developers%20Guide%20to%20the%20Extensible%20Type%20Registry.html. Version: 2005

[21] IBM Corporation u. a.: *GEF Programmer's Guide*. http://help.eclipse.org/help33/topic/org.eclipse.gef.doc.isv/guide/guide.html. Version: 2007

[22] Kolb, B. ; Efftinge, S. ; Völter, M. ; Haase, A. : GMF Tutorial. In: *iX* (2006), Nr. 12. http://www.voelter.de/data/articles/ix-gmf2.pdf

[23] Miller, J. ; Mukerji, J. : MDA Guide / OMG. Version: 1.0.1. http://www.omg.org/cgi-bin/apps/doc?@omg/03-06-01.pdf (omg/2003-06-01). – Technical Guide

[24] Object Management Group, Inc.: *MDA Homepage*. http://www.omg.org/mda/

[25] Object Management Group, Inc.: *OMG Homepage*. http://www.omg.org

[26] Object Management Group, Inc.: *UML Homepage*. http://www.uml.org/

[27] Object Management Group, Inc.: Meta Object Facility (MOF) Core Specification / OMG. Version: 2.0. http://www.omg.org/cgi-bin/apps/doc?formal/06-01-01.pdf (formal/06-01-01). – Technical Guide

[28] Object Management Group, Inc.: Unified Modeling Language: Infrastructure / OMG. Version: 2.1.1. http://www.omg.org/cgi-bin/apps/doc?formal/07-02-06.pdf (formal/2007-02-06). – Technical Guide

[29] Object Management Group, Inc.: Unified Modeling Language: Superstructure / OMG. Version: 2.1.1. http://www.omg.org/cgi-bin/apps/doc?formal/07-02-05.pdf (formal/2007-02-05). – Technical Guide

[30] Warmer, J. ; Kleppe, A. : *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003 (The Addison-Wesley Object Technology Series)