Fakultät für Mathematik, Informatik und Naturwissenschaften
Research Group Software Construction

Bachelorarbeit

Erweiterung des Gargoyle Codegenerators um Semantische Beziehungen

Extending the Gargoyle Codegen with semantic relationships

Tristan Langer

März, 2012

Gutachter:
Prof. Dr. rer. nat. Horst Lichter
Prof. Dr. rer. nat. Bernhard Rumpe

Betreuer:
Dipl.-Inform. Matthias Vianden

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 16. März 2012

— Tristan Langer —

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

1 Einleitung

Inhalt

Seit Beginn der Softwareentwicklung werden Programme immer komplexer. Damit steigt auch der Bedarf an Methoden und Werkzeugen, die es ermöglichen qualitativ hochwertige Software möglichst effizient zu erstellen [LL07].
Einen Ansatz dafür bietet die modellgetriebene Softwareentwicklung. Modelle werden in den Naturwissenschaften schon lange eingesetzt, um bestimmte Zusammenhänge zu beschreiben oder zu dokumentieren. Auch in der Softwareentwicklung werden Modelle immer häufiger zur Planung und Entwicklung eingesetzt. Dabei hat sich die Unified Modeling Langague (UML) als die de facto Modellsprache für Struktur und Verhalten von Software entwickelt. Sie wird von immer mehr Firmen zur modellbasierten Softwareentwicklung eingesetzt. Dabei wird die Software zunächst als Modell beschrieben und anschließend durch Programmierer in Quellcode umgesetzt. Dabei bleiben Modell und Quellcode jedoch unabhängig voneinander, so dass sich Änderungen auf einer Seite nicht unmittelbar auf die andere Seite auswirken [LCM06].

Die modellgetriebene Softwareentwicklung führt diese Idee weiter. Bei ihr besteht ein direkter Zusammenhang zwischen Modell und Quellcode, da hier der Quellcode direkt aus dem Modell generiert wird. Die modellgetriebene Softwareentwicklung ist daher vor allem für große Softwarestrukturen, die überwiegend einfache Funktionalitäten beinhalten, prädestiniert. Insbesondere wenn Software ein leicht aber aufwändig programmiertes Grundgerüst erfordert, ist der zeitliche Aufwand für die Erstellung der Grundfunktionalität sehr hoch im Vergleich zur Komplexität der erstellten Software.

Die praktische Arbeit bei der Entwicklung von Informationssystemen am Lehr- und Forschungsgebiet 3(LuFGi3) der RWTH Aachen weist Eigenschaften auf, die den Einsatz modellgetriebener Softwareentwicklung sinnvoll machen. Die entwickelte Software besteht in der Regel aus einer großen Datenstruktur mit sich wiederholender Zugriffsfunktionalität in Enterprise JavaBeans. Daher wurde in einer vorangehenden Diplomarbeit unter dem Namen „Gargoyle Codegenerator“ eine Möglichkeit entwickelt das Grundgerüst der Software für Informationssysteme in Enterprise Java Beans unmittelbar aus zuvor erstellten Modellen zu generieren [Löw11]. Der damals konzipierte Generator besaß bereits ein einfaches Assoziationskonzept. Dies soll in dieser Arbeit nun um semantische

Beziehungen, im speziellen um Komposition und Generalisierung erweitert werden. Aggregation wird ausgeschlossen, da ihre Bedeutung mehr semantischer Natur ist und daher im dazu generierten Quellcode kein Unterschied zu einer normalen Assoziation bestünde.

1.1 Überblick über die Arbeit

Diese Arbeit ist in acht Kapitel aufgeteilt: *Einleitung*, *Grundlagen*, *Der Gargoyle Codegenerator*, *Verwandte Arbeiten*, *Konzept der Generatorerweiterung*, *Technische Umsetzung*, *Evaluation* und *Zusammenfassug und Ausblick*. Nachdem in der Einleitung die Arbeit motiviert wurde, werden im folgenden Kapitel die Grundlagen für das Verständnis der Arbeit geliefert. Dies umfasst Informationen über Modellgetriebene Softwareentwicklung, einen Überblick über Java Enterprise Edition und Enterprise JavaBeans, eine Einführung in die Unified Modeling Language sowie eine Beschreibung aller für die Entwicklungsarbeit verwendeten Werkzeuge. Im Kapitel „Der Gargoyle Codegenerator“ wird anschließend der EJB-Generator vorgestellt, auf dem diese Arbeit aufbaut.
Für die Entwicklung der Generatorerweiterung werden im folgenden Kapitel zunächst verwandte Arbeiten betrachtet, um einen Überblick über den derzeitigen Stand vergleichbarer Generatortechnik zu erhalten. Darauf aufbauend wird das Konzept der Erweiterung des Gargoyle Codegenerators im Rahmen dieser Arbeit präsentiert. Im Kapitel „Technische Umsetzung“ folgen Informationen über die Realisierung des Konzepts. Um den Entwicklungsprozess abzuschließen, wird das Ergebnis dieser Arbeit im Kapitel „Evaluation“ getestet und auf seine Zweckmäßigkeit überprüft, indem der Generator für die Entwicklung von Informationssystemen verwendet wird.
Im letzten Kapitel folgt eine abschließende Zusammenfassung der Arbeit und ein Ausblick auf mögliche zukünftige Entwicklungsarbeit am Gargoyle Codegenerator.

1.2 Danksagung

Ich möchte mich bei Prof. Dr. rer. nat. Horst Lichter dafür bedanken, dass ich meine Bachelorarbeit an seinem Lehrstuhl schreiben durfte. Des Weiteren danke ich Prof. Dr. Bernhard Rumpe als Zweitgutachter. Ein ganz großer Dank geht an Dipl.-Inform. Matthias Vianden für die außergewöhnlich engagierte und gute Betreuung bei meiner Arbeit und an Simona Jeners M.Sc. für ihre Hilfe beim Evaluieren und Testen.
Ich danke zudem ganz herzlich meiner Familie, die mir stets zur Seite steht und mich bei meinem Studium unterstützt und allen Studenten, die ebenfalls mit mir am Gargoyle Codegenerator gearbeitet haben für die tollen Gargoyle Meetings.

2 Grundlagen

Inhalt

Im Anschluss an die Einleitung werden nun die für diese Arbeit wichtigen Grundlagen vorgestellt. Dazu wird zunächst die modellgetriebene Softwareentwicklung im Allgemeinen näher beschrieben. Anschließend werden die verwendete Entwicklungsumgebung, sowie die UML als Eingabesprache und Enterprise JavaBeans als Zielsprache der Codegenerierung dieser Arbeit erläutert. Abschließend werden die für den Generierungsprozess genutzten Werkzeuge vorgestellt. Sprachen sowie Werkzeuge wurden durch die vorangegangene Arbeit „Generierung von Web-basierten Prototypen für Geschäftsanwendungen“ festgelegt [Löw11].

2.1 Modellgetriebene Softwareentwicklung

In der heutigen Zeit hat der Großteil aller Unternehmen eine eigene IT-Abteilung. Das bedeutet, dass die Wirtschaftlichkeit der meisten Unternehmen unter anderem auf Software basiert. Bei der Entwicklung von Software werden meist Modelle zur Dokumentation und Planung eingesetzt, um die komplexe Struktur der Programme zu abstrahieren. Die modellgetriebene Softwareentwicklung befasst sich mit der automatischen Generierung von Software aus Modellen und bietet somit einen neuen Ansatz, der vor allem Fehler in sich wiederholenden, einfachen Programmabschnitten vermeidet.

Zu Beginn der Entwicklungsphase wird die Software zunächst durch ein Modell beschrieben. Durch die visuelle Modellierung mit einer geeigneten Sprache wie zum Beispiel UML(siehe Abschnitt 2.3), kann die komplexe Struktur einer Software abstrahiert werden. Wird die Software in mehreren Modellen mit verschiedenen Abstraktionsstufen modelliert, können einzelne Modelle in anderen, ähnlichen Projekten wiederverwendet werden, wenn das Problem ähnlich ist und nur Details neu modelliert werden müssen. Aus den Modellen werden dann der Quellcode und gegebenenfalls für die Software relevante Artefakte generiert. Das können zum Beispiel, wie in dieser Arbeit, Datenbankabfragen der Software sein. In der Regel ist es zudem sinnvoll Tests zu generieren, die die

Entwickler beim weiteren Entwicklungsprozess unterstützen [Con12]. Durch die Generierung wichtiger Artefakte aus dem Modell bleiben außerdem Modell und Quellcode synchron. Wird der Quellcode nicht unmittelbar aus dem Modell generiert, sind Änderungen im Modell, die oftmals gerade in frühen Entwicklungsphasen auftreten, zeitaufwändig auf den Quellcode zu übertragen. [PTN+07]

Dieses Vorgehen bietet mehrere Vorteile:

- **Reduzierter Aufwand:** Wird die modellgetriebene Softwareentwicklung sinnvoll in einem Projekt eingesetzt, indem die Software ein großes Softwaregerüst mit viel einfacher Grundfunktionalität beinhaltet, wird ein beträchtlicher Zeitanteil durch dessen Generierung eingespart.
- **Synchronisation von Quellcode und Modell:** Gerade in frühen Entwicklungsstadien durchlaufen Softwarestrukturen oft Veränderungen. Da Modell und Quellcode synchron bleiben, müssen Änderungen am Modell nicht mühselig in den Quellcode übertragen werden.
- **Qualitätssteigerung:** Der generierte Quellcode enthält in der Regel weniger Fehler, als von Hand erstellter Code. Voraussetzungen dafür sind natürlich, dass der Generator richtig funktioniert und die resultierenden Artefakte zuvor korrekt modelliert wurden.
- **Bessere Wartbarkeit:** Dadurch, dass generierter Quellcode einheitlich erzeugt wird, verbessert sich Übersichtlichkeit und Verständlichkeit des Quellcodes. Da sich Quellcode dieser Art besser weiterentwickeln lässt, resultiert daraus bessere Wartbarkeit der Software. Dennoch hängt die Wartbarkeit von generiertem Quellcode auch stark von der Einhaltung von Programmierprinzipen im generierten Quellcode, sowie im Modell der Software, ab.

2.2 Java Enterprise Edition und Enterprise JavaBeans

Nach der Einführung in die modellgetriebene Softwareentwicklung wird nun die Zielarchitektur des Gargoyle Codegenerators vorgestellt. *Enterprise JavaBeans (EJBs)* sind Teil der *Java Enterprise Edition (JavaEE)*, die für die Entwicklung von Anwendungen für mehrschichtige verteilte Systeme in Java entwickelt wurde. Ziel des Gargoyle Codegenerators ist es Grundfunktionalität eines Informationssystems in EJB-Code zu generieren.

Java Enterprise Edition (JavaEE) spezifiziert eine Reihe weiterer Schnittstellen, sogenannte *Application Programming Interfaces* (APIs), die die Entwicklung mehrschichtiger verteilter Systeme auf Unternehmensebene vereinfachen sollen. Ziel dabei ist es Komponenten so zu kapseln, dass sie projektübergrei-

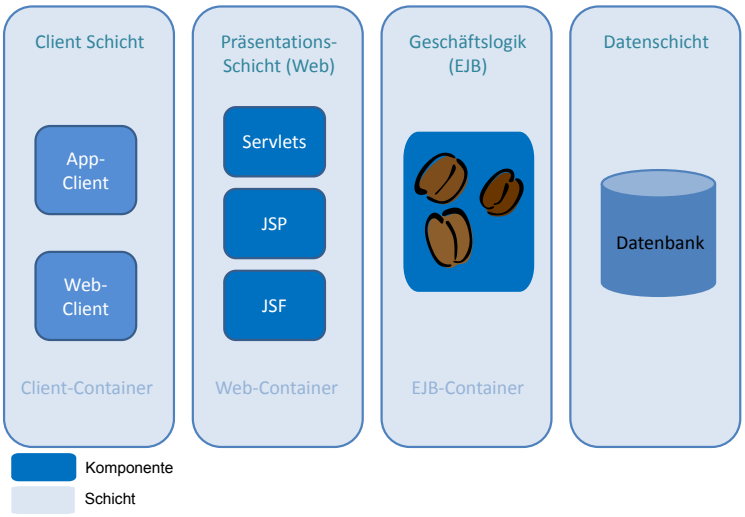


Abbildung 2.1: JEE Standardarchitektur basierend auf [Via11]

fend kompatibel bleiben. JavaEE Komponenten erfordern daher einen JavaEE Zertifizierten Server, der über eine bestimmte Infrastruktur und Bestandteile (Client-Container, Web-Container und EJB-Container) verfügt, die für die Ausführung der Komponenten benötigt werden. Die bekanntesten Beispiele sind der IBM WebSphere Application Server, sowie der Oracle GlassFish Server und der JBoss Application Server [IBM] [Gla] [JBo]. Abbildung 2.1 zeigt die Architektur einer typischen JavaEE Anwendung nach [Via11]. Zugriffe von Benutzern erfolgen aus der Client Schicht. Diese kommunizieren mit der Präsentationsschicht. Die Präsentationsschicht selbst verwendet die Java EE spezifischen APIs Servlets, JavaServer Pages (JSP) und JavaServer Faces (JSF) um die Ausgabe von Daten gestaltet.

Enterprise JavaBeans (EJBs) werden in der Geschäftslogik verwendet. EJBs implementieren die Methoden von verteilten Systemen und sollen dabei Funktionen wie Datenbankzugriffe und Sicherheit standardisieren. Instanzen der EJBs werden zur Laufzeit nach Bedarf, also Abhängig von der Anzahl der Client-Anfragen, dynamisch vom Server erstellt, wiederverwendet und gelöscht. EJBs gliedern sich in der aktuellen Version 3.0 in zwei Haupttypen: Session Beans und Message Driven Beans.
Session Beans können Geschäftsprozesse darstellen, die dem Client direkt zugänglich sind. Die Kommunikation mit Session Beans erfolgt über Interfaces. Session Beans, die im selben EJB-Container ausgeführt werden, können über ein Local Interface kommunizieren, während Session Beans in verschiedenen Containern nur über das Remote Interface kommunizieren.

- **Stateful Session Beans:** Stateful Session Beans können für Geschäftsprozesse verwendet werden, die einen Zustand beinhalten. Technisch bedeutet das, dass die Instanzvariablen nach Aufruf einer Methode nicht verloren gehen und dem Client während seiner gesamten Session zur Verfügung stehen. Dadurch lässt sich zum Beispiel der Einkauf in einem Web-Store realisieren, bei dem der Zustand des Einkaufswagens während der gesamten Session eines Clients gespeichert bleiben muss.

- **Stateless Session Beans:** Stateless Session Beans können Geschäftsprozesse abbilden, deren Zustand nicht gespeichert wird. Jeder Aufruf hinterlässt ein Stateless Session Bean unverändert. Dadurch sind Stateless Session Beans nicht an einen bestimmten Client gebunden.

Message Driven Beans stehen im Gegensatz zu Session Beans nicht in direktem Kontakt zum Client. Sie werden stattdessen mit einem Nachrichtendienst assoziiert und ausgeführt sobald dort eine neue Nachricht eingeht. Diese Art von Beans erlaubt es daher Methoden ereignisbasiert auszuführen.

Die Datenbank enthält persistente Daten eines Systems in Form von Entitäten. Deren Zugriff wurde in früheren EJB Versionen in Form von EntityBeans realisiert, die noch zu Kompatibilitätszwecken existieren. Datenbankzugriffe werden jedoch in der aktuellen EJB Version nicht mehr behandelt und stattdessen in der Java Persistence API (JPA) neu spezifiziert. Die JPA ist in der Lage Java Objekte direkt in relationale Datenbanken zu überführen und kann daher auch unabhängig von EJB eingesetzt werden. [Löw11] [Via11] [IHHK07]

2.3 Klassendiagramme der Unified Modeling Language (UML)

Im Folgenden wird kurz die UML im Allgemeinen und anschließend UML-Klassendiagramme beschrieben. Modelle dienen im Gargoyle Codegenerator zur Beschreibung der Architektur, aus der anschließend EJB-Code generiert wird. Des Weiteren werden Klassendiagramme zum Bau der Meta-Modelle des Generators benutzt.

Die Unified Modeling Language (UML) ist eine Modellierungssprache, die zur visuellen Beschreibung von Struktur und Verhalten von Softwareartefakten in der objektorientierten Programmierung dient. Die Modelle dienen oft zur Spezifikation von Artefakten eines Softwaresystems, können aber auch direkt zur Erstellung durch Codegenerierung beitragen oder abschließend für Dokumentationszwecke genutzt werden. Die UML umfasst mehrere Diagrammtypen, die entweder den Aufbau eines Softwaresystems darstellen oder dessen Abläufe und Verhalten beschreiben.

Klassendiagramme sind ein Diagrammtyp der UML. Sie dienen zur Beschreibung der Struktur eines Systems und bilden somit den Ansatz für weitere Modelle, die Abläufe innerhalb des Systems genauer beschreiben. Im Folgenden werden die für die Arbeit relevanten Teile von Klassendiagrammen beschrieben. Für eine vollständige Beschreibung wird auf die Quelle [RJB99] verwiesen.

2.3.1 Klassen

Wie auch in der objektorientierten Programmierung, ist die Klasse eines Klassendiagramms ein Element, dass Typen von Objekte mit gleichen Eigenschaften beschreibt. Die Eigenschaften umfassen dabei sowohl die Merkmale (Attribute und Beziehungen) der Objekte, als auch deren Verhalten (Methoden). Außerdem gehören die Objekte in der Regel sinngemäß zusammen.

In Abbildung 2.2 ist die Klasse „Forum“ zu sehen. Sie ist in drei Abschnitte unterteilt. Der oberste Abschnitt enthält den Namen der Klasse. Der Name muss eindeutig sein, so dass die Klasse identifizierbar ist. Der Abschnitt kann zudem Schlüsselworte enthalten, die angeben, ob es sich bei der Klasse zum Beispiel um eine abstrakte Klasse oder ein Interface handelt. Der mittlere Abschnitt enthält die Attribute der Klasse. Diese haben immer einen Namen und einen Typ. Weitere Angaben, wie die Sichtbarkeit sind optional und für diese Arbeit nicht relevant. Der letzte Abschnitt enthält die Methoden der Objekte. Methoden haben eine Signatur und einen Rumpf. Meist wird im Klassendiagramm jedoch nur die Signatur angegeben und der Rumpf durch andere Diagrammtypen beschrieben. Ist einer der unteren Abschnitte leer, so wird er oft ganz weggelassen.

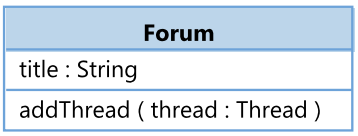


Abbildung 2.2: Beispielklasse eines Klassendiagramms

Neben den normalen Klassen gibt es noch sogenannte *abstrakte Klassen*. Dies sind Klassen, die nicht instanziiert werden sollen. Sie werden in der Regel in Verbindung mit der Generalisierung(siehe Abschnitt 2.3.3) benutzt. In diesem Fall sollen nur spezielle Objekte erzeugt werden, die die Attribute des allgemeinen abstrakten Objekts erhalten. Das macht Sinn, wenn zum Beispiel Attribute für die Klasse „Gebäude“ beschrieben wurden, es sollen aber nur Objekte der speziellen Gebäude wie „Mehrfamilienhaus“, „Krankenhaus“ etc. erzeugt werden. [RJB99]

2.3.2 Aufzählungstypen (Enumeration Types)

Aufzählungstypen dienen dazu eigene Datentypen mit endlichem Wertebereich zu erstellen und werden in einigen Meta-Modellen dieser Arbeit verwendet. So lässt sich zum Beispiel ein Typ „Gewürz“ mit den Werten „salz“ und „pfeffer“ erstellen. Die Darstellung in der UML ist der von Klassen sehr ähnlich. Im Abschnitt mit dem Namen wird zusätzlich die Bezeichnung «enumeration» angegeben. In dem Abschnitt darunter werden anschließend die möglichen Werte aufgeführt. [RJB99]

2.3.3 Beziehungen

Beziehungen haben die Funktion die Objekte zweier Klassen miteinander zu verknüpfen. Dadurch ist es möglich größere Systeme zu konstruieren. Klassendiagramme können eine Vielzahl verschiedener Beziehungen mit jeweils bestimmten Bedeutungen beinhalten. Im Weiteren werden die für diese Arbeit relevanten Beziehungen erläutert.

Assoziationen

Assoziationen verknüpfen die Objekte zweier Klassen miteinander. Assoziierte Objekte können in Assoziationsrichtung auf einander zugreifen. Eine Assoziation besitzt in der Regel einen Namen und ein Tupel von Referenzen, die die Assoziationsenden beschreiben. Jede Referenz wiederum besitzt eine Rolle, eine Kardinalität und die Angabe, ob diese Richtung navigierbar ist. Bei unidirektionalen Assoziationen gibt ein Pfeil an der Spitze einer der Assoziationsenden die Navigationsrichtung an. Bei bidirektionalen Assoziationen sind beide Navigationsrichtungen durch Pfeile markiert, üblicherweise werden diese aber auch ganz weggelassen.

Die Abbildung 2.3 zeigt eine bidirektionale Assoziation zwischen den Klassen *Forum* und *User*. Die Assoziation hat den Namen *forumUser*. Die Referenz auf die Klasse Forum verweist auf eine beliebige Anzahl von Objekten des Typs *Forum* mit der Rolle *forum*, was durch die Kardinalität * angegeben wird. Die Referenz auf die Klasse User verweist ebenfalls auf eine beliebige Anzahl von Objekten. Diese Objekte des Typs User haben die Rolle *users*. [RJB99] [Rum04]

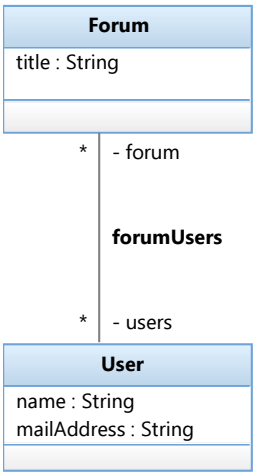


Abbildung 2.3: Assoziation eines Klassendiagramms

Semantische Beziehungen

Neben den einfachen Assoziationen existieren für Klassendiagramme auch noch Beziehungen mit speziellen Bedeutungen. In dieser Arbeit werden zunächst nur die Komposition und die Generalisierung betrachtet, da diese die am häufigsten verwendeten semantischen Beziehungen sind, die sich ebenfalls auf generierten Quellcode auswirken.

Komposition Die Komposition ist eine spezielle Form der Assoziation. Sie beschreibt die Abhängigkeit eines Objekts als Teil eines Ganzen. Dies wird visuell als gefüllte Raute am Ende des Objekts markiert, das das Ganze repräsentiert. Ein konkretes Teilobjekt kann immer nur von einem anderen konkreten Objekt abhängen. Zudem hängt auch die Existenz des Teilobjekts von der Existenz des Objekts ab, welches das Ganze verkörpert. Wird ein Objekt gelöscht, so gilt dies auch für alle seine Teilobjekte. Außerdem kann ein Teilobjekt nicht alleine existieren. Das Objekt, das das Ganze repräsentiert nennt man *Kompositum*, das Teilobjekt heißt *Komponente*. In Abbildung 2.4 ist die Komposition von der Klasse *Thread* zur Klasse *Post* zu sehen. Thread und Post stellen die aus Foren bekannten Threads und Posts dar. Zunächst teilt diese Komposition alle Eigenschaften einer Assoziation. Sie besitzt den Namen *posts* und zwei Referenzen. Die Referenz zu der Klasse *Thread* ist durch eine Raute gekennzeichnet. Das bedeutet, dass Objekte der Klasse *Post* von jeweils einem Objekt der Klasse *Thread* abhängen. Ein Post kann nicht alleine und nur in einem Thread gleichzeitig existieren. Wird der Thread gelöscht, so gilt dies auch für den Post. [Rum04]

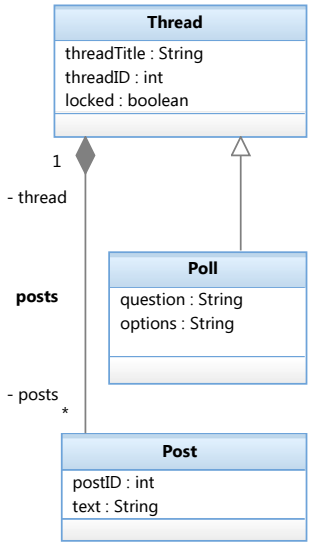


Abbildung 2.4: Komposition und Generalisierung eines Klassendiagramms

Generalisierung Die Generalisierung ist eine gerichtete Beziehung von einer speziellen zu einer allgemeineren Klasse. Sichtbar wird dies durch eine leere Pfeilspitze am Beziehungsende der allgemeineren Klasse. Werden zwei Klassen durch eine Generalisierung in Beziehung gesetzt, so bedeutet dies, dass die spezielle Klasse die Attribute, Methoden und Assoziationen der allgemeineren Klasse erhält. Diese Attribute und Methoden werden im Klassendiagramm jedoch nicht in die spezielle Klasse eingetragen. Abbildung 2.4 zeigt eine Generalisierung zwi-

schen der Klasse *Poll* und der Klasse *Thread*. Polls sind spezielle Threads, die zusätzlich zu den normalen Attributen und Posts noch eine durch den Ersteller spezifizierte Frage mit Antwortmöglichkeiten enthält, über die andere Benutzer des Forums abstimmen können. Die Klasse *Poll* erhält durch die Generalisierung also implizit die Attribute *ThreadTitle*, *TheadID* und *locked*, sowie die Komposition *posts*. Dem fügt sie die eigenen Attribute *question* und *options* hinzu. [RJB99]

2.4 Entwicklungsumgebung

Im Folgenden wird die für diese Arbeit verwendete Entwicklungsumgebung erläutert. Der Gargoyle Codegenerator und somit auch dessen Erweiterung werden als Eclipse Plug-In entwickelt. Der Rational Software Architect (RSA) dient in dieser Arbeit zum erstellen von Meta-Modellen in UML. Für die Umwandlung erstellter Meta-Modelle in Modelle des Generators und der Transformation zwischen Modellen, sowie der Generierung von Quellcode aus einem Modell werden EMF, Xtend, XPand und Check verwendet.

2.4.1 Eclipse

Eclipse ist eine Open Source Softwareentwicklungsumgebung. Während Eclipse selbst wenig Funktionalität liefert, liegt der eigentliche Wert in der Erweiterbarkeit durch die umfangreiche Plug-In Architektur. Diese erlaubt es Eclipse benötigte Plug-Ins zu finden und auszuführen. Daher existieren bereits viele freie, aber auch viele kostenpflichtige Erweiterungen für verschiedenste Aufgaben.
Eclipse wurde zunächst als Java Entwicklungswerkzeug erstellt, dient aber mittlerweile dank seiner Erweiterbarkeit und dem riesigen Angebot an Plug-Ins für eine Vielzahl von Entwicklungsaufgabe. Da Eclipse selbst in Java programmiert ist, ist es zudem plattformunabhängig, sofern eine geeignete Runtime Umgebung für das Betriebssystem existiert. [Wel]

2.4.2 Rational Software Architect

Der Rational Software Architect ist ein von IBM entwickeltes Werkzeug zur umfangreichen Modellierung mit UML. RSA basiert auf der Open Source Entwicklungsumgebung Eclipse und verfügt über Funktionalitäten um UML Modelle auf Fehler zu prüfen, sowie aus Diagrammen Quellcode für verschiedene Programmiersprachen und aus Quellcode wiederum Diagramme zu generieren. Außerdem verfügt RSA über eine Reihe weiterer Funktionalitäten, die für diese Arbeit jedoch nicht relevant sind. [Rat]

2.4.3 Eclipse Modeling Framework

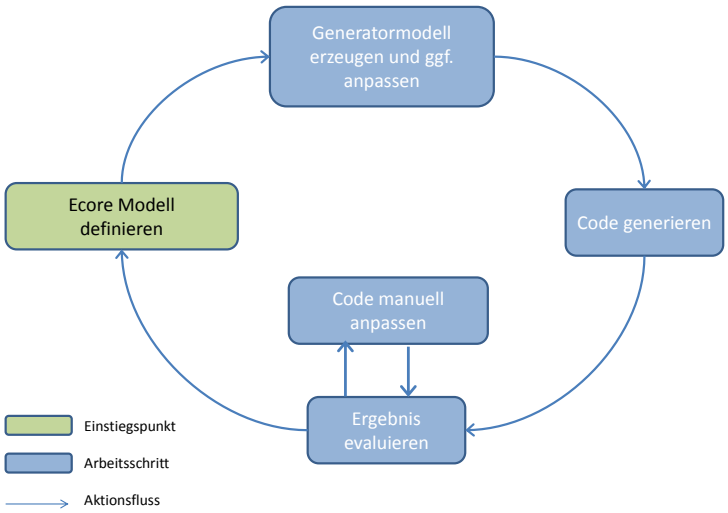


Abbildung 2.5: Vereinfachte Darstellung der Arbeitsschritte mit EMF zur Generierung von Quellcode aus einem Modell basierend auf [BG05].

Das Eclipse Modeling Framework (EMF) ist ein Open Source Werkzeug für Eclipse und Teil von Eclipse Tools. Dieses Framework kann zur Modellierung und Codegenerierung eingesetzt werden. EMF ist nach der Model Driven Architecture (MDA) entwickelt, die ein Leitfaden zur Entwicklung von Software durch modellgetriebene Softwareentwicklung beinhaltet. Wie in 2.1 beschrieben, verfolgt daher auch EMF das Ziel den Hauptanteil der Entwicklung von Software auf Modellebene zu verlagern.

Zu diesem Zweck bietet EMF seine eigene Modellsprache „ecore“, deren Meta-Modell ähnlich zum Meta-Modell von UML-Klassendiagrammen ist. Darin könne Meta-Modell neu spezifiziert oder alternativ bestehende Modelle in zum Beispiel UML oder XML importiert werden. Dadurch ist es möglich auch bereits bestehende Systeme unkompliziert mit EMF zu nutzen. In ecore werden Modelle platformunabhängig definiert.

Abbildung 2.5 zeigt eine vereinfachte Darstellung der Arbeitsschritte mit EMF. Aus dem platformunabhängigen Modell erstellt EMF ein Generatormodell (GenModel), das zu den Informationen des ecore-Modells weitere platformabhängige Informationen wie Dateipfade enthält. Das GenModel wird zudem ständig mit dem ecore-Modell synchronisiert, so dass sich Änderungen am ecore-Modell stets auch auf das GenModel auswirken.

Ist ein Modell erstellt, generiert EMF drei Eclipse Plug-Ins. Zum einen das „model“ Plug-In, dass den zugehörigen Java Quellcode für das Meta-Modell enthält. Dieser kann anschließend manuell angepasst werden, um zum Beispiel Methodenrümpfe zu implementieren. Modell und Quellcode können auch weiterhin durch EMF synchron gehalten werden. Manuell implementierte Teile des Quellcodes sind davon jedoch ausgeschlossen und bleiben unverändert bestehen. Sie müssen nachträglich von Hand synchronisiert werden.
Zusätzlich zu dem Quellcode erstellt EMF ein „editor“ und ein „edit“ Plug-In für das Modell. Das editor Plug-In enthält Code zur visuellen Darstellung von Modellen in einem Baum-basierten Editor. Darin können in einer Baumstruktur Instanzen des ursprünglichen Meta-Modells erstellt werden. Das edit Plug-In enthält vor allem Editor unabhängige Adapter, die nötig sind, damit das Modell auch unabhängig des editor Plug-Ins genutzt werden kann. [BG05]

2.4.4 Modell zu Modell und Modell zu Text Transformationen

Für Modell zu Modell (M2M) und Modell zu Text (M2T) Transformationen werden im Gargoyle Codegenerator Xtend, Xpand und Check benutzt. Alle drei Werkzeuge wurden ursprünglich als Teil der openArchitectureWare entwickelt, werden aber mittlerweile unter dem Namen „Modeling Workflow Engine“ im Eclipse Modeling Framework weiterentwickelt [ope]. Die Werkzeuge basieren alle auf der Untersprache „oAW expression“, die eine Mischung aus Java und Object Constraint Language (OCL, siehe auch Abschnitt 2.4.4) ist. Sie benutzen zudem ein flexibles Typsystem, das ihre Verwendung mit beliebigen Modellen erlaubt.

Xtend

Mit Xtend lassen sich sogenannte Extensions erstellen. Darin lassen sich Meta-Modelle, die zum Beispiel mit EMF erstellt wurde, mit zusätzlicher Logik versehen. Zusätzlich existiert in Xtend eine create Komponente, die es erlaubt neue Objekte in Modellen zu erstellen. Daher eignet sich Xtend sehr gut für Modell zu Modell Transformationen. Extensions lassen sich auch aus anderen Sprachen wie zum Beispiel Java, Xpand oder Check aufrufen.
Codebeispiel 2.1 enthält eine Methode „printAttributeName“, die das Attribut eines EntityModels als Parameter erhält und dessen String Attribute „name“, mit dem ersten Buchstaben als Großbuchstaben, zurückgibt. Codebeispiel 2.2 zeigt die create-Methode für eine Entität. Diese erhält eine Klasse als Parameter. Die Methode erstellt dann eine Entität, setzt deren Namen auf den Namen der Klasse und gibt das neue Objekt zurück. Dabei ist zu beachten, dass mehrfache Methodenaufrufe mit den selben Parametern immer das selbe Objekt zurückgeben. Das heißt, dass nur beim erstmaligen Aufruf ein neues Objekt erstellt wird und weitere Aufrufe mit den selben Parametern keine neuen Objekte erzeugen, sondern das beim ersten Aufruf erstellte Objekt zurückgeben. [EFH+10]

Quelltext 2.1: Xtend Transformation Codebeispiel

```
1 String printAttributeName(entitymodel::Attribute attr):
2         attr.name.toFirstUpper();
```

Quelltext 2.2: Xtend Create Codebeispiel

```
1 create entitymodel::Entity this transform(Class class):
2         this.setName(class.name);
```

Xpand

Bei Xpand handelt es sich um eine Template basierte Sprache, die aus Modellen Text generiert. Xpand verfügt nur über wenige eigene Befehle, kann jedoch beliebigen Text und somit Quellcode in jeder Programmiersprache erzeugen. Um Xpand Befehle und Ausgabetexte zu unterscheiden, verwendet Xpand die Französischen Anführungszeichen „«“ und „»“. Befehle innerhalb der Anführungszeichen werden von Xpand interpretiert. Quellcode 2.3 zeigt beispielhaft den Aufbau eines Templates und zugleich die wichtigsten Befehle von Xpand. Zunächst werden relevante Meta-Modelle und in Xtend erstellte Extensions importiert. Anschließend können für Objekte des Modells Ausgaben definiert werden. Im Beispiel wird das Ausgabemuster, eine sogenannte Cartridge, „Class“ für ein Objekt „Entity“ des Modells definiert. Die Ausgabe wird in eine Datei geschrieben, deren Namen über die Extension „getFileName“ erstellt wird. Innerhalb der Cartridge wird eine weitere Cartridge „Constructor“ für die selbe Entität aufgerufen. Quellcode 2.4 zeigt den für die Entität „Forum“ im Package „forumsystem“ aus dem Template resultierenden Code in Java. [EFH+10]

Quelltext 2.3: Xpand Templatebeispiel

```
1  «IMPORT a::meta::model»
2  «EXTENSION my::Extension»
3  «DEFINE Class FOR a::meta::model::Entity»
4  «FILE this.getFileName()»
5  package «this.getPackage()» ;
6
7  public class «this.name» {
8    «EXPAND Constructor FOR this»
9  }
10 «ENDFILE»
11 «ENDDEFINE»
12
13 «DEFINE Constructor FOR a::meta::model::Entity»
14   public «this.name»(){
15         }
16 «ENDDEFINE»
```

Quelltext 2.4: Generierter JavaCode

```
1 package forumsystem;
2
3 public class Forum{
4   public Forum(){
5         }
6 }
```

Check

Check basiert auf der Object Constraint Language(OCL), die Teil der UML ist. Die OCL wird benutzt, um Invarianten für Klassendiagramme zu definieren. Analog dazu lassen sich mit Check Bedingungen für Modelle überprüfen. Dadurch lässt sich sicherstellen, dass ein Modell zum Beispiel vor einer Transformation bestimmte Richtlinien erfüllt, die für den Ablauf der Transformation wichtig sind. Außerdem ist es zusätzlich möglich zuvor in Xtend erstellte Extensions in Check aufzurufen. Codebeispiel 2.5 prüft zum Beispiel, ob eine Beziehung, die im GeneratorModel des Gargoyle Generators erstellt wurde, eine Zielklasse besitzt. Im Rumpf wird dazu die Bedingung angegeben, die erfüllt sein soll. In diesem Fall soll jede Beziehung eine Zielklasse besitzen. Wird diese Bedingung nicht erfüllt, wird die darüber definierte Fehlermeldung ausgegeben. [EFH+10]

Quelltext 2.5: Check Codebeispiel

```
1 context Relationship ERROR
2         "A relationship needs a target class." :
3         targetClass != null;
```

3 Der Gargoyle Codegenerator

Inhalt

Nachdem in den Grundlagen die Prinzipien der modellgetrieben Softwareentwicklung sowie relevante Sprachen und Werkzeuge vorgestellt wurden, wird nun der Gargoyle Codegenerator beschrieben, auf dem diese Arbeit aufbaut. Dazu wird die Zielarchitektur erläutert, die von der Standard JEE Architektur abweicht und anschließend wird auf den Generatoraufbau und Ablauf des Generierungsprozesses eingegangen.

3.1 Zielarchitektur

In Abschnitt 2.2 wurde die Standardarchitektur für JEE Anwendungen beschrieben. Der Gargoyle Codegenerator generiert jedoch eine vom LuFGi3 entwickelte Architektur, die insbesondere die Geschäftslogik detaillierter beschreibt. Es ist anzumerken, dass die aktuelle Version sich geringfügig von der Version unterscheidet, für die der Generator ursprünglich entwickelt wurde. Das liegt vor allem an der Verwendung neuer Technologien. Diese Architektur ist in der Abbildung 3.1 zu sehen. Geschäftsschicht und Datenschicht entsprechen den aus der Standardarchitektur bekannten Schichten. Die Client-Schicht entspricht hingegen der Präsentationsschicht der Standardarchitektur. Die Pfeile zeigen den Kontrollfluss, der durch Aktionen des Benutzers ausgelöst wird.

Die Client-Schicht stellt in diesem Modell die Benutzeroberflächen bereit. Diese können auf die Anwendungsfacade zugreifen. Die Facade wiederum enthält die Signaturen aller Controller-Methoden. Facadenaufrufe werden dann an die assoziierten Controller-Methoden weitergeleitet. Die Controller-Methoden selbst implementieren die Funktionalitäten des Systems auf Geschäftsebene. Durch sie werden zum Beispiel Entitäten erstellt, verändert oder gelöscht. Eine Controller-Methode enthält dabei nicht nur die eigentliche Aktion, also zum Beispiel im Falle des Erstellens einer Entität den „createEntity“ Aufruf, sondern prüft in diesem Fall auch, ob identifizierende Attribute vorhanden sind, ob die Entität noch nicht existiert und ruft, nach dem Erstellen der Entität, die Methode zum Speichern der Entität in der Datenbank auf.
Der Großteil aller Methoden, die aus der Codegenerierung dieser Arbeit entste-

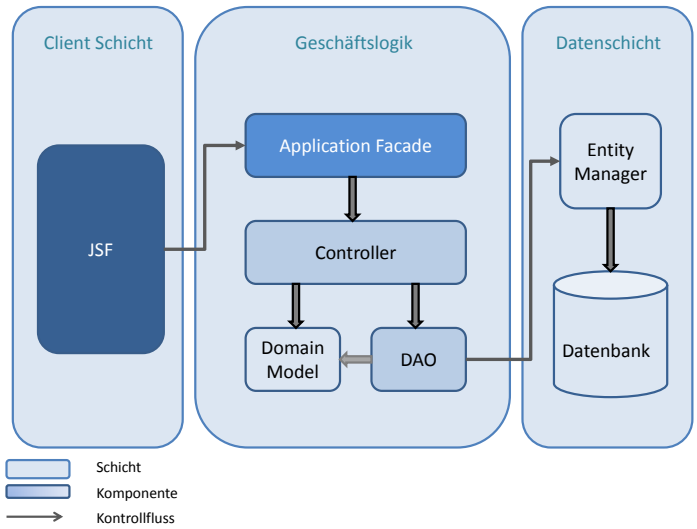


Abbildung 3.1: Anwendungsarchitektur des LuFGi3 basierend auf [Via11]

hen, werden daher Controller-Methoden sein, die die Funktionalität der verschiedenen Beziehungen beinhalten. Zur Ausführung der Funktionalitäten benutzen Controller-Methoden das Domain Model. Daraus werden benötigte Entitäten ermittelt und deren Data Acces Object(DAO) benutzt, um die zugehörigen persistenten Daten aus der Datenbank abzurufen.

Der hohe Detailgrad der Geschäftslogik und Einteilung der Zuständigkeiten in mehrere Schichten bietet einige Vorteile. Es ist ohne großen Aufwand möglich Teile des Systems auszutauschen und gegebenenfalls in ähnlichen Projekten wiederzuverwenden. Dadurch wird ein hohes Maß an Flexibilität während der Entwicklung erreicht. Zudem erleichtert die Architektur die spätere Wartung und Erweiterung des Systems. Einzelne Teile können außerdem leichter auf ähnliche Systeme übernommen werden. Die klare Trennung von Funktionalität und Datenbankzugriffen liefert eine gute Übersicht über das gesamte System und trägt somit zur Fehlervermeidung und Fehlerfindung bei. [Via11][Löw11]

3.2 Aufbau des Gargoyle Codegenerators

Neben seiner speziellen Zielarchitektur verfügt der Gargoyle Codegenerator zudem über einen besonderen Aufbau. Eine Besonderheit in den Anforderungen des Generators besteht darin, dass die Ergebnisse aller Generierungsschritte einsehbar und anpassbar seien sollen. Das bedeutet, dass der Gargoyle Codegenerator ein UML-Modell nicht unmittelbar in EJB-Code überführt, sondern jeder Generierungsschritt in einem eigenen Modell festgehalten wird.

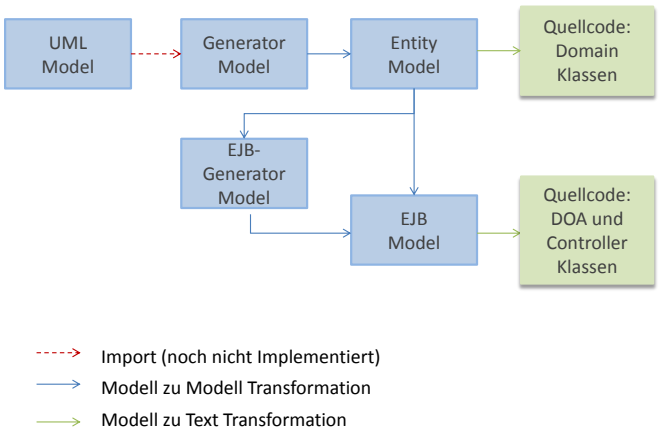


Abbildung 3.2: Generierung von EJB Quellcode mit dem Gargoyle Codegenerators basierend auf [Via11]

In Abbildung 3.2 ist der Ablauf einer Codegenerierung für die Geschäftslogik eines Informationssystems inklusive aller beteiligten Modelle zu sehen. Die einzelnen Meta-Modelle haben sich seit ihrem Entwurf in der vorangegangen Diplomarbeit von Tobias Löwenthal [Löw11] erheblich weiterentwickelt.
Zunächst wird ein Informationssystem in UML modelliert. Dieses Modell wird dann in das GeneratorModel aufgenommen. Mithilfe von Modell zu Modell Transformation wird das GeneratorModel anschließend in ein EntityModel überführt. Dies enthält die Grundlage für die Datenbank der Anwendung. Aus dem EntityModel entsteht anschließend das EJB-GeneratorModel, das generatorspezifische Eigenschaften enthält. EntityModel und EJB-GeneratorModel sind dann Grundlage für das EJBModel. Das EJB Model beschreibt die Methoden, die durch die EJBs bereitgestellt werden. Der eigentliche Quellcode wird mittels Modell zu Text Transformation aus dem EntityModel und dem EJBModel erzeugt. Aus dem EntityModel entstehen die Domain Klassen, die die Entitäten der Datenbank beschreiben und aus dem EJBModel entstehen die EJBs, die die Methoden auf den Entitäten der Datenbank zur Verfügung stellen. Im Weiteren werden die einzelnen Modelle genauer erläutert.

3.2.1 GeneratorModel

Das GeneratorModel dient im Gargoyle Codegenerator zur Eingabe des UML-Modells, aus dem der EJB-Quellcode generiert werden soll. Das Meta-Modell ist in Abbildung 3.3 zu sehen. Zunächst fällt die Ähnlichkeit zum UML-Meta-

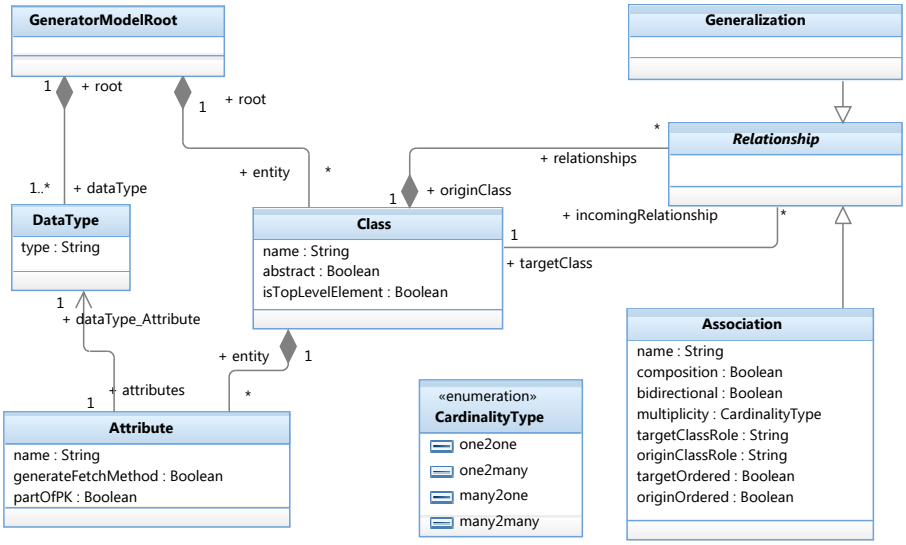


Abbildung 3.3: GeneratorModel Meta-Modell des Gargoyle Codegenerators

Modell für Klassendiagramme auf, da diese als Eingabe dienen. Es enthält Klassen mit Attributen und Beziehungen(Relationship). Jedem Attribut ist ein Datentyp zugeordnet, der einen primitiven oder komplexen Datentyp verkörpert. Beziehungen werden in Abschnitt 6.1 erläutert.

Zusätzlich enthält das GeneratorModel generatorspezifische Merkmale: die Klasse GeneratorModelRoot und die Attribute isTopLevelElement innerhalb der Klasse Class, sowie generateFetchMethod und partOfPK in der Klasse Attribute.
Die Klasse GeneratorModelRoot ist die Wurzel des Modells und somit Einstiegspunkt in eine Instanz des Meta-Modells. Das heißt von der Wurzel aus sind alle Elemente direkt oder indirekt navigierbar. Das ist für die Darstellung im Baumeditor, den EMF erstellt notwendig, da Baumeditoren eine Wurzel benötigen, an der alle weiteren Objekte erzeugt werden.
Das Attribut isTopLevelElement gibt an, ob die Entität, die für die Klasse erzeugt wird, direkt navigierbar ist. Diese Entitäten werden auf der Indexseite eines Informationssystems aufgelistet.
Ist generateFetchMethod für ein Attribut gesetzt, erstellt der Generator eine Datenbankabfrage für das Attribut. Durch diese werden dann alle Entitäten, die ein Attribut des Typs mit dem angegeben Wert besitzen, abgerufen.
Mit partOfPK werden die Attribute einer Klasse markiert, durch die die Klasse eindeutig identifizierbar ist.

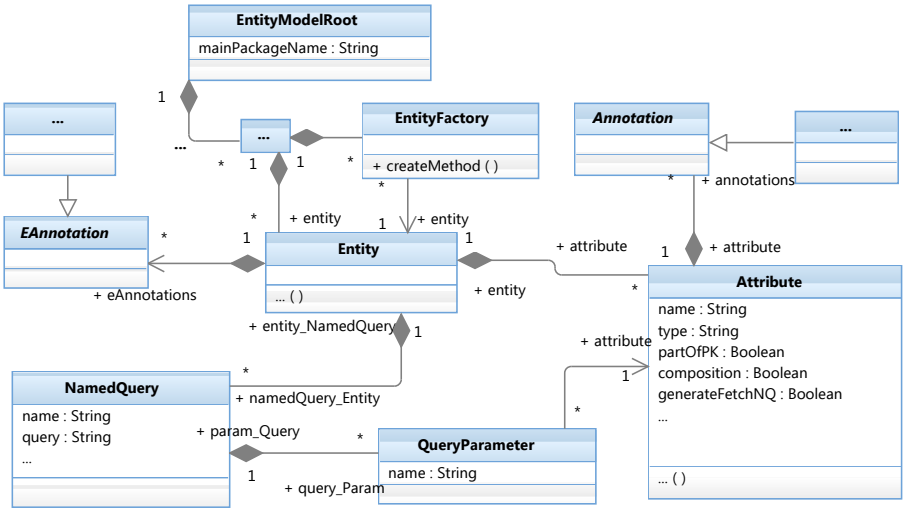


Abbildung 3.4: Vereinfachte Darstellung des Entitymodel Meta-Modells des Gargoyle Codegenerators

3.2.2 EntityModel

Das EntityModel ist die Grundlage für die sogenannten persistenten Entitäten. Dabei handelt es sich um Java Objekte, die durch die JPA in relationale Datenbanken überführt werden sollen. Dazu werden die Klassen aus dem GeneratorModel im EntityModel um einige Informationen erweitert, die die JPA benötigt, um die Entitäten in eine Datenbank zu speichern und wieder abzurufen. Dazu gehören ein Attribut „ID“, dass für jede Entität automatisch erstellt wird und zur Identifizierung der Entität in der Datenbank benötigt wird, sowie Datenbankanfragen(Queries) und einer Create Methode in der EntityFactory, mit der eine Instanz der Entität erzeugt werden kann.

Abbildung 3.4 zeigt eine vereinfachte Darstellung des EntityModel Meta-Modells des Gargoyle Codegenerators. Auch hier existiert wieder ein Wurzelelement, das für die Generierung des Editors durch EMF benötigt wird. Darunter befinden sich zum einen die EntityFactory, die die Methoden zum Erstellen der Entitätinstanzen enthält und zum anderen die Entitäten selbst. Diese besitzen wiederum Attribute. Entitäten entstehen aus den Klassen des GeneratorModels, während Attribute sowohl die Attribute des Generatormodels, als auch Assoziationen des Generatomodels enthalten. Die Attribute erhalten je nach Typ eventuell zusätzlich eine Annotation, um von der JPA richtig interpretiert zu werden. Das Attribut ID erhält zum Beispiel eine Annotation, die es als identifizierendes Attribut markiert. Genauere Informationen wie Beziehungen umgesetzt werden sind in Kapitel 6.2 zu finden.

Datenbankabfragen(NamedQueries) werden je nach Bedarf erstellt. Zunächst erhält jede Entität eine Abfrage für ihren Primärschlüssel, der aus einer Kom-

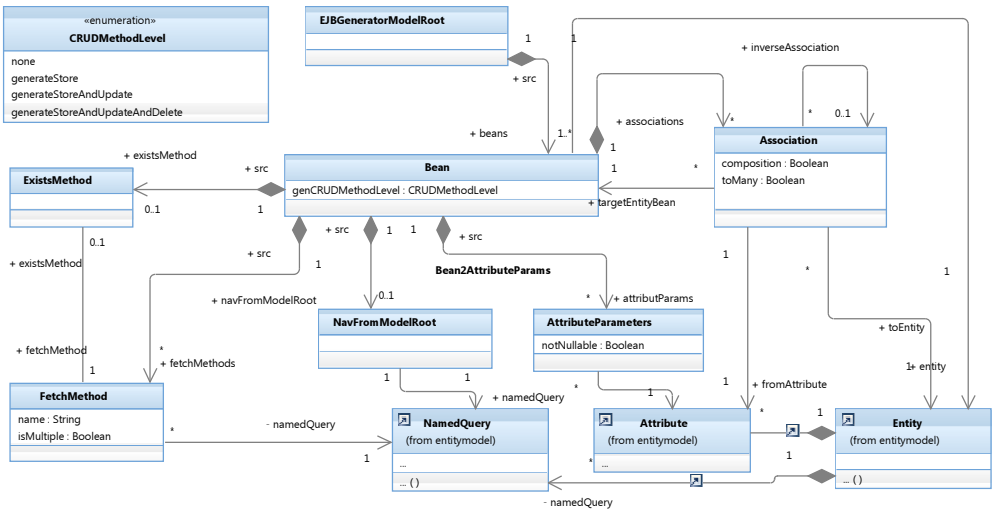


Abbildung 3.5: Vereinfachte Darstellung des EJBGeneratormodel Meta-Modells des Gargoyle Codegenerators

bination aller „partOfPK“-Attibute besteht. Zusätzlich erhalten alle Entitäten, deren Klasse im GeneratorModel als „isTopLevelElement“ markiert wurden, eine „getAll...“-Abfrage, die alle Entitäten des jeweiligen Typs aus der Datenbank abruft. Schlussendlich wird eine Datenbankabfrage für alle Attribute die mit „generateFetchMethod“ im GeneratorModel markiert wurden erstellt, die die entsprechende Entität anhand dieses Attributs aus der Datenbank abruft.

3.2.3 EJB-GeneratorModel

Das EJB-Generatormodel beinhaltet generatorspezifische Informationen für die Erstellung der EJB Methoden im EJBModel. In Abbildung 3.5 ist eine vereinfachte Darstellung des EJBGeneratormodel Meta-Modells des Gargoyle Codegenerators zu sehen. Unter der aus den anderen Modellen bekannten Wurzel wird für jede Entität des EntityModels eine Instanz der Klasse „Bean“ erstellt. Diese Klasse entspricht allerdings keinem Bean an sich, sondern enthält Objekte, die den Typ der späteren EJB Methoden beschreiben. Somit enthält das Modell eine Übersicht aller im Generierungsprozess erstellten Methoden.

In der Bean Klasse kann die Variable CRUDMethodLevel festgelegt werden, die bestimmt ob create-, update- und delete-Methoden für die Verwaltung der jeweiligen Entität selbst erstellt werden. Für normale Entitäten werden standardmäßig alle Methoden erstellt. Abstrakte Entitäten bilden eine Ausnahme und erhalten später unabhängig des CRUDMethodLevels keine create-Methode, da diese Entitäten nicht instanziiert werden sollen.

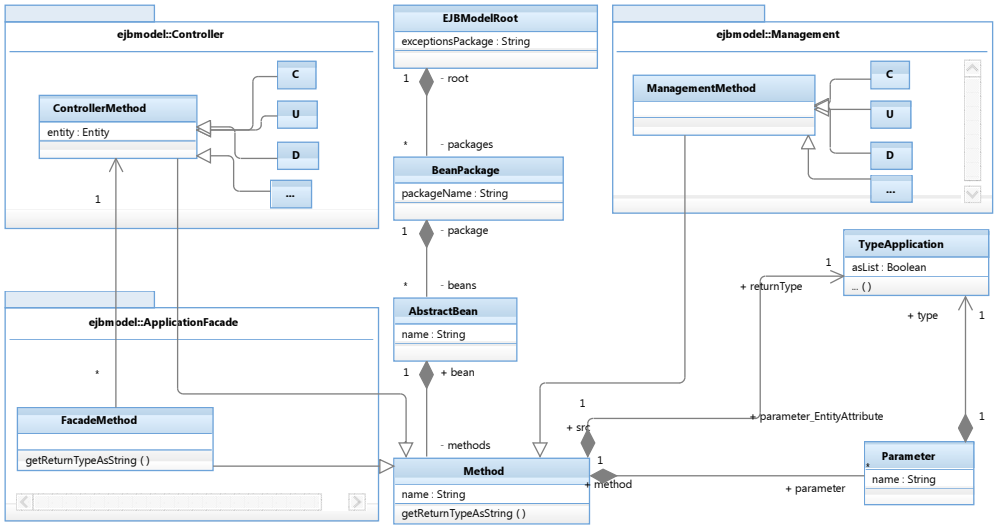


Abbildung 3.6: Vereinfachte Darstellung des EJBGeneratormodel Meta-Modells des Gargoyle Codegenerators. Die Pakete Controller und ApplicationFacade entsprechen den gleichnamigen Komponenten in der Darstellung der Anwendungsarchitektur in Abschnitt 3.1. Das Management-Paket entspricht der DAO Komponente.

Für alle Datenbankabfragen werden fetch-Methoden erstellt, um die Abfragen nutzbar zu machen. Außerdem werden für alle nicht-identifizierenden Attribute Objekte des Typs AttributeParameter mit der Information, ob dieses Attribut den Wert „null“ annehmen darf, angelegt. Auf das Assoziationskonzept wird im Detail in Abschnitt 6.3 eingegangen.

3.2.4 EJBModel

Das EJBModel des Gargoyle Codegenerators enthält alle Informationen die für die Generierung der EJB-Methoden in der Geschäftslogik notwendig sind. Das EJBModell Meta-Modell besteht hauptsächlich aus drei Komponenten: Management, Controller und ApplicationFacade. Management und Controller beinhalten Funktionalität für Entitäten und werden daher für jede Entität instanziiert. Die Application Facade hingegen kapselt lediglich Aufrufe für alle Controller Methoden und wird daher nur einmal erzeugt.

Abbildung 3.6 zeigt eine stark vereinfachte Darstellung des EJBGeneratormodel Meta-Modells des Gargoyle Codegenerators. Zu sehen ist zunächst die aus den vorhergehenden Modellen bekannte Wurzel, die zur Darstellung im Editor dient. Diese Wurzel enthält Pakete, die durch den Generator jeweils zu einem Management-, Controller- und ApplicationFacade-Paket instanziiert werden. Diese Pakete wiederum enthalten abstrakte Beans mit Methoden, die ih-

rerseits Parameter enthalten. Methoden sowie Parameter haben einen Typ, der durch eine TypeApplication gehandhabt wird. Die Art der Methode hängt dabei davon ab, in welcher Art von Paket sie erzeugt wird. Zu sehen sind die drei Pakettypen Management, Controller und ApplicationFacade.

Das Management-Paket enthält Management Methoden, die DAO-Objekte verkörpern. Diese beinhalten JPA-Funktionalität und können nicht direkt vom Client ausgeführt werden. Die Abbildung 3.6 zeigt beispielhaft CUD (create, update, delete). Nebenher existieren aber noch weitere Methodentypen. Dazu gehören Persistieren, Aktualisieren und Löschen der zur Entität gehörigen Tabellen in der Datenbank sowie die Abfrage einer bzw. aller Entitäten eines bestimmten Typs und die Überprüfung, ob die Tabelle einer bestimmten Entität bereits in der Datenbank vorhanden ist. Welche Methoden erstellt werden ist von der CRUD-Method Einstellung im EJBGeneratormodel abhängig.

Das Controller-Paket enthält Controller Methoden. Dabei handelt es sich um diejenigen Methoden, die Funktionalität für den Client bereitstellen und auf Entitätebene arbeiten. Dabei werden die Management Methoden für Datenbankzugriffe verwendet. Die Abbildung 3.6 zeigt Beispielhaft create-, update- und delete-Methode. Vom Gargoyle Codegenerator werden Controller-Methoden für alle Management-Funktionalitäten und Assoziationen erstellt. Enthält das Management-Paket also zum Beispiel eine persist-Methode für eine Entität, wird im Controller eine create-Methode erstellt. Diese führt Notnull-Checks für die angegeben Parameter durch und überprüft, ob eine solche Entität schon existiert. Anschließend ruft sie die Entity-Factory auf, die eine Entität des entsprechenden Typs erstellt und persistiert diese Entität mithilfe der Management Methode als Tabelle in der Datenbank. Controller-Methoden für Assoziationen werden in Abschnitt 6.4 genauer erläutert.

Das ApplicationFacade-Paket wird am Ende der Transformation zum EJBModel erstellt. Es enthält jeweils eine Facade-Methode für jede Controller-Methode und delegiert ihren Aufruf lediglich an die assoziierte Controller-Methode. Die Facade ist daher ein einfaches Interface, das die Nutzbarkeit der Anwendung verbessern soll.

4 Verwandte Arbeiten

Inhalt

Im vorangehenden Kapitel wurden Aufbau und Funktion des Gargoyle Codegenerators erläutert. In diesem Kapitel werden nun verwandte Arbeiten im Hinblick darauf untersucht, wie Komposition und Generalisierung im Rahmen von verteilten Systemen in bereits existierenden Generatoren realisiert werden. Auf Grundlage dieser Beobachtungen soll anschließend ein Konzept für die Implementierung von Beziehungen im Gargoyle Codegenerator entwickelt werden. Es existieren eine Vielzahl verschiedener Web-Generatoren. Da EJB 3.0 aber eine relativ junge Technologie ist, unterstützen die meisten Generatoren nur Quellcode für EJB 2.0.

4.1 AndroMDA

Bei AndroMDA handelt es ich um ein Open-Source-Projekt zur modellgetriebenen Softwareentwicklung. Das Programm ist in der Lage aus Modellen in UML Quellcode in einer beliebigen Sprache zu generieren. Dazu kann es mit neue Plug-Ins erweitert werden, die nicht vorhandene Sprachen unterstützen oder vorhandene Plug-Ins angepasst werden.
AndroMDA wird jedoch hauptsächlich zu Entwicklung von JEE Anwendungen und zugehörigem EJB 3.0 Quellcode verwendet.
Der Generator wird von seiner Community weiterentwickelt. Einige Entwickler überprüfen Inhalte regelmäßig. AndroMDA liegt zum Zeitpunkt dieser Arbeit in Version 3.3 vor und basiert somit auf einem längeren Entwicklungszeitraum, der einen hohen Reifegrad der Software zur Folge hat. AndroMDA verfügt über eine einfache Generator Architektur. Zwischenergebnisse können nicht ausgegeben oder verändert werden. [And]

4.1.1 Beziehungen

AndroMDA verfügt über ein einfaches Assoziationskonzept. Für das grundlegende UML-Modell werden nötige Attribute inklusive Annotationen sowie die

zugehörigen set-und get-Methoden erzeugt. Die Namen für Attribute und Methoden basieren auf den Rollen der Klassen in der Beziehung, wenn vorhanden. Sonst wählt das Programm einen passenden Namen basierend auf dem Klassennamen aus.
Zusätzlich bietet AndroMDA einige weitere Merkmale für Assoziationen.

- **Kaskadierung:** Kaskadierung bietet die Möglichkeit verwaltende Funktionen, wie das Aktualisieren und Löschen von Elementen, in Zusammenhang mit einer Assoziation automatisch für beide Assoziationsenden durchzuführen.

- **Fetch Type:** Wird eine Entität geladen, kann durch den Fetch-Type eingestellt werden, ob Objekte, die durch eine Assoziation mit der Entität verknüpft sind, unmittelbar (eager) oder nach Bedarf (lazy) geladen werden.

- **Sortierte Collections:** Assoziationen zu Collections, also einer Sammlung von mehreren Objekten desselben Typs, können standardmäßig nach Primärschlüssel sortiert oder manuell spezifiziert, einem der anderen Attribute, zurückgegeben werden.

Generalisierung AndroMDA unterstützt Vererbung inklusive der drei Strategien „SingleTable“, „Table Per Class“ und „Table Join“(siehe Abschnitt 5.3 für Details zu den Strategien).
Es ist möglich Entitäten als „MappedSuperclass“ zu markieren. Dadurch wird für diese Klasse keine Tabelle in der Datenbank angelegt. Klassen, die als Entitäten annotiert wurden, können trotzdem Attribute und Methoden einer solchen Klasse erben.

Komposition Komposition und Aggregation werden in AndroMDA benutzt um das Erstellen einer Join-Tabelle zu modellieren. Die Annotation der Join-Tabelle wird dabei auf der mit der Raute markierten Seite der Beziehung im Quellcode spezifiziert. Join-Tabellen beinhalten alle Informationen der Assoziation, so dass die Tabellen der Entitäten keine assoziationsspezifischen Informationen mehr enthalten müssen. Die in Abschnitt 2.3.3 beschriebene Semantik von Kompositionen wird nicht umgesetzt.

4.2 TLGen

TLGen ist ein kostenpflichtiger Generator für JEE Anwendungen, der zwar EJB 3.0 unterstützt, aber ursprünglich auf Basis von EJB 2.0 entwickelt wurde. TLGen generiert aus einem Domainmodel in UML eine umfangreiche Projektarchitektur, die von Web-Services auf Präsentationsebene, zugehörige Test-Klassen,

Geschäftslogik und Datenbankzugriffe umfasst. TLGen wird primär von der Firma StarData entwickelt und vertrieben. Zum Zeitpunkt der Arbeit liegt der Generator in der Version 2.5 vor und bietet somit wie AndroMDA ein ausgereiftes Generierungskonzept. [TLG]

4.2.1 Beziehungen

Wie AndroMDA generiert TLGen Attribute, Annotationen und set- sowie get-Methoden für Assoziationen. Semantische Assoziationen werden allerdings nicht unterstützt.

4.3 Sculptor

Sculptor ist ein durch seine Community vorangetriebenes Open-Source-Codegeneriungswerkzeug. Sculptor basiert auf Fornax, einer Entwicklungsplattform, die ihrerseits wiederum auf Eclipse und dem oAW-Rahmenwerk aufbaut. Der generierte Quellcode unterstützt viele bekannte Technologien, unter anderem auch JEE, JSF und EJB, die für diese Arbeit von Bedeutung sind.
Zur Eingabe benötigt Sculptor Modelle, die in einer eigenen domänenspezifischen Sprache spezifiziert wurden. Als Zieltechnologie stehen EJB und Spring zur Verfügung. [Scu]

4.3.1 Beziehungen

Sculptor unterstützt ein eigenes Beziehungskonzept. Assoziationen können in der domänenspezifischen Sprache spezifiziert werden. Alle Entitäten sind dabei standardmäßig Aggregat-Wurzeln und bilden somit ein eigenes Aggregat. Für alle Aggregate wiederum werden im Verlauf der Codegenerierung Repositories erzeugt, die Datenbankabfragen sowie das Persistieren und Löschen der Daten einer Entität in der Datenbank verwalten. Werden Entitäten explizit als nicht-Aggregat-Wurzeln gekennzeichnet, so werden sie Teil des Aggregats der Entität, mit der sie assoziiert sind und erhalten somit kein eigenes Repository. Zusätzlich generiert Sculptor für Entitäten im selben Aggregat den Kaskadierungstypen „all-delete-orphan“, der Teil der Hibernate Technologie[1] ist. Dieser Kaskadierungstyp überträgt persist/update/delete Aufrufe auf assoziierte Objekte und löscht ein assoziiertes Objekt, wenn die Assoziation aufgelöst wird und es nicht mit einem anderen Objekt assoziiert ist.

Damit implementiert Sculptor eigentlich eine Mischung aus Aggregation und Komposition. Es ist möglich, dass Entitäten, die Teil einer Aggregation sind ohne ihr Aggregat existieren, solange sie nicht mit einer Entität assoziiert werden.

[1] Hibernate dient ähnlich der JPA dazu Java Objekte in relationale Datenbanken abzubilden.

Sobald die Assoziation zugewiesen wurde, hängt ihr Lebenszyklus jedoch von dem des Aggregats ab.
Zusätzlich unterstützt Sculptor Vererbung inklusive Vererbungsstrategien. Standardmäßig wird die Strategie „Joined“ verwendet (siehe Abschnitt 5.3 für Details zu Vererbungsstrategien).

4.4 Fazit

Die untersuchten Codegeneratoren enthalten alle ein einfaches Assoziationskonzept, das zugehörige Attribute, Annotationen und get- sowie set-Methoden erzeugt. Zusätzlich sind je nach Generator weitere Einstellungen möglich. Bemerkenswert ist jedoch, dass keine Implementierung für das Zuweisen der Assoziation erstellt wird. Die Funktionalität muss weiterhin von Hand in einer eigenen Methode oder nach Bedarf erstellt werden, obwohl es sich dabei um Standardfunktionalität handelt. Somit werden auch besondere semantische Eigenschaften der Komposition nicht ausreichend beachtet.
Die Implementierung der Generalisierung ist ebenfalls selbst in ausgereiften Generatoren nicht selbstverständlich. Da Vererbung erst mit EJB 3.0 leicht umsetzbar wurde, fehlt diese Funktionalität in vielen Generatoren, die auf EJB 2.0 basieren. Die eigentliche Codegenerierung wirkt sich hauptsächlich auf den Quellcode der Entitäten aus und wird durch die Verwendung von EJB vereinfacht, da das Vererbungsprinzip von Java verwendet werden kann.

5 Konzept der Generatorerweiterung

Inhalt

Nachdem bereits existierende Software für das Generieren von Beziehungen für JEE Anwendungen betrachtet wurden, wird nun ein neues Assoziationskonzept sowie ein Konzept für die Einführung semantischer Beziehungen für den Gargoyle Codegenerator entwickelt. Dabei muss zunächst spezifiziert werden, wie Assoziation, Komposition und Generalisierung im Quellcode und dessen Methoden umgesetzt werden sollen. Anschließen soll, unter Berücksichtigung der besonderen Generatorarchitektur des Gargoyle Codegenerators, ein Konzept für die Umsetzung der Beziehungen für jeden Schritt des Generierungsprozesses erarbeitet werden.

5.1 Assoziation

Da der Gargoyle Codegenerator ein UML-Diagramm als Eingabe erhält, wird die Umsetzung von Beziehungen im GeneratorModel des Gargoyle Codegenerators ähnlich dem UML-Meta-Modell sein. Das soll die erste Eingabe des Diagramms vereinfachen. Es wäre sowohl eine Umsetzung als Tupel aus Referenzen, als auch als gesamte Assoziation denkbar. Referenzen hätten den Vorteil, dass sie dem UML-Meta-Modell am nächsten sind und zusammengehörige Informationen der Assoziationsenden (Rollenname der Klasse, Multiplizität und Navigierbarkeit) sinnvoll Kapseln.

Die Umsetzung als gesamtes Assoziationsobjekt bietet hingegen eine kompaktere Alternative und erleichtert das Erstellen von Transformationsskripten, da das Navigieren durch Objektebenen mit Xtend umständlich ist. Außerdem unterscheidet EJB nur vier Typen von Assoziationen bezüglich ihrer Multiplizität, eins zu eins, eins zu viele, viele zu eins und viele zu viele. Das macht eine genauere Unterscheidung im Generator unnötig. Daher wird eine Lösung angestrebt, deren zugrunde liegendes Modell Beziehungen, und davon abgeleitet Assoziationen, als ganzes Objekt umsetzt. Die Informationskapselung von Referenzen wird in den Editor des Generators übernommen, um die Eingabe für den Benutzer zu erleichtern.

Für die Transformation in das EntityModel des Generators müssen alle nötigen Annotationen innerhalb der Entitäten als Objekte im EntityModel erstellt und im Quellcode der Entitäten generiert werden. Um die Arbeit mit Assoziationen zu vereinfachen werden Kaskadierungstypen implementiert. Zunächst wird jedoch nur der Merge-Typ verwendet, bei Aktualisierung einer Tabelle in der Datenbank durch eine geänderte Entität ebenfalls die Tabellen ihrer assoziierten Entitäten aktualisiert. Es folgt eine Aufzählung aller Kaskadierungstypen nach [IHHK07]:

- Persist: Der Kaskadierungstyp Persist bewirkt, dass beim Persistieren einer Entität in die Datenbank alle assoziierten Objekte ebenfalls persistiert werden. Das ist nützlich, wenn in einer Methode mehrere Objekte erzeugt und assoziiert werden.
- Merge: Durch den Kaskadierungstyp Merge werden bei der Synchronisation der Datenbank durch die JPA nicht nur die Tabelle der übergebene Entität synchronisiert, sonder ebenfalls assoziierte Entitäten mit ihrem Stand in der Datenbank abgeglichen und persistiert falls noch keine entsprechende Tabelle vorhanden ist.
- Refresh: Wird eine Entität aufgrund geänderter Werte in der Datenbank aktualisiert, so gilt dies auch für assoziierte Entitäten, wenn die Assoziation den Kaskadierungstyp Refresh besitzt.
- Remove: Wird eine Entität gelöscht, so werden alle Entitäten, die mit einer mit dem Kaskadierungstyp Remove gekennzeichneten Assoziation mit der gelöschten Entität verbunden waren, ebenfalls gelöscht.
- All: Der Kaskadierungstypen bezeichnet die Kurzform für alle oben aufgeführten Kaskadierungstypen.

Das Zuweisen und Auflösen der Assoziationen ist Standardfunktionalität und soll daher nicht von Hand programmiert werden müssen. Die zugehörigen Methoden werden im Controller des EJBModels und im Quellcode der entsprechenden EJBs erzeugt. Alle dafür nötigen Informationen muss das EJBGeneratorModel bereitstellen. Für bidirektionale Assoziationen werden allerdings zwei Methoden für die Verwaltung der Assoziation erstellt. Um die Implementierung weiterer Funktionalität beim Zuweisen bzw. Auflösen von Assoziationen für den Entwickler, der die generierte Software weiterentwickelt, nicht unnötig kompliziert zu machen, wird eine der beiden Methoden ihren Aufruf an die andere Methode delegieren. Dadurch bleibt die Verwaltung der Assoziation bei einer Methode, obwohl zwei verschiedene Methoden existieren. Der Methodenname wird durch die Rollen der beteiligen Klassen im GeneratorModel erstellt. Daher darf eine Klasse in verschiedenen Assoziationen nicht dieselbe Rolle haben, wenn die Assoziation zur Klasse navigierbar ist. Vor dem Löschen einer Entität werden alle Assoziationen (auch Kompositionen) aufgelöst.

Des Weiteren muss das Parameterkonzept der erstellten Methoden berücksichtigen, dass Parameter zweier Klassen gegebenenfalls denselben Namen besitzen können (zum Beispiel wenn eine Klasse sich selbst assoziiert).

5.2 Komposition

Die Komposition soll im Gargoyle Codegenerator die in Abschnitt 2.3.3 beschriebene Semantik erfüllen. Die Semantik besagt, dass Entitäten, die als Komponente modelliert werden, nicht unabhängig ihres Kompositums existieren können. Daher wird die Komponente im EntityModel des Gargoyle Codegenerators als schwache Entität umgesetzt. Das heißt, dass Entitäten von Komponenten nur im Zusammenhang mit der Entität ihres Kompositums eindeutig identifizierbar sind. Der Primärschlüssel des Kompositums wird beim Erstellen der Komponente zum Fremdschlüssel für die Komponente und ist damit Teil deren identifizierender Attribute. Die zusätzlichen Primärschlüsselarttibute der Komponente müssen folglich auch nur im Zusammenhang mit dem Fremdschlüssel eindeutig sein. Abbildung 5.1 zeigt Beispielhaft das Fremdschlüsselkonzept in einer Kompositionshierarchie der Klassen Forum, Thread und Post, unter der Voraussetzung, dass alle dort aufgeführten Attribute identifizierende Attribute sind.

Die Abfrage der Fremdschlüssel kann für bidirektionale Kompositionen direkt über das Assoziierte Kompositum erfolgen. Bei unidirektionalen Kompositionen ist das Kompositum für Komponenten jedoch nicht direkt navigierbar. Daher wird in diesem Fall ein Attribut erzeugt, dass den Wert des Fremdschlüssels speichert. Dieses Attribut muss beim Erstellen der Entität gesetzt werden und bei Aktualisierung der Kompositum Entität durch die update Methode ebenfalls aktualisiert werden.

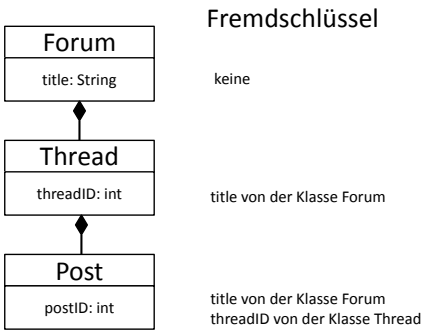


Abbildung 5.1: Fremdschlüsselbeispiel einer Kompositionshierarchie

Da Komponenten bei ihrer Erzeugung mit ihrem Kompositum assoziiert werden, muss im Gegensatz zu normalen Assoziation lediglich eine Methode für das Auflösen der Komposition erstellt werden. Das Auflösen einer Komposition führt zur Löschung der Komponente. Um keine Endlosschleife durch das Auflösen aller Assoziationen in der delete-Methode zu erzeugen, muss das eigentliche Auflösen in der delete-Methode geschehen, wäh-

rend die Methode für das Auflösen der Komposition nur die delete-Methode der Komponente aufruft.

Dieses Konzept bringt einige Einschränkungen mit sich, die frühzeitig durch Check-Fehlermeldungen überprüft werden müssen. Zunächst darf jede Komponente nur ein Kompositum besitzen und Kompositionsketten dürfen keine Zyklen enthalten. Um Konflikte mit Attributnamen und Fremdschlüsseln zu vermeiden, müssen alle identifizierenden Attribute innerhalb einer Kompositionskette einen eindeutigen Namen besitzen.

5.3 Generalisierung

Die Einführung der Generalisierung im Gargoyle Codegenerator wird durch EJB 3.0 stark vereinfacht, da das Vererbungsprinzip von Java genutzt werden kann. Daher wirkt sich dieser Prozess Hauptsächlich auf das EntityModel und den Quellcode der entsprechenden Entitäten aus. Um alle nötigen Annotationen zu generieren, muss das EntityModel entsprechend erweitert werden. Es sollen die gängigen Vererbungsstrategien „Single Table“, „Table Per Class“ und „Joined“ unterstützt werden. Es folgt eine Aufzählung der Strategien nach [IHHK07]:

Single Table bildet die gesamte Vererbungshierarchie in eine einzige Tabelle ab. Zur Unterscheidung der einzelnen Klassen wird eine neue Spalte in der Tabelle angelegt. Das ermöglicht schnelle Zugriffe und polymorphe Abfragen, verbraucht jedoch am meisten Speicherplatz, da alle Attribute als Spalten übernommen werden. Für Klassen, die das jeweilige Attribut jedoch nicht besitzen, bleibt der Wert leer. Abbildung 5.2 veranschaulicht, wie die Oberentitäten Thread und die beiden Unterentitäten Poll und ThreadWithIcon durch die SingleTable Vererbungsstrategie in relationale Datenbanken überführt würden.

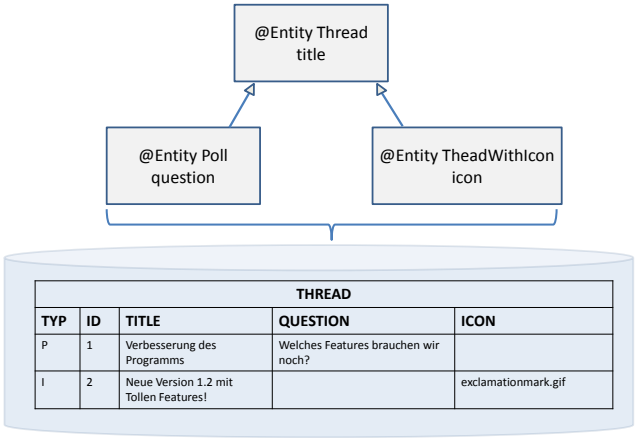


Abbildung 5.2: Veranschaulichung der Single Table Vererbungsstrategie

Table Per Class bildet jede Klasse der Vererbungshierarchie auf eine eigene Tabelle ab. Dabei werden geerbte Attribute als eigene Attribute übernommen. Dadurch sind zwar schnelle Zugriffe auf alle Klassen möglich, polymorphe Abfragen werden aber nicht unterstützt. Abbildung 5.3 veranschaulicht, wie die Oberentitäten Thread und die beiden Unterentitäten Poll und ThreadWithIcon durch die Table Per Class Vererbungsstrategie in relationale Datenbanken überführt würden.

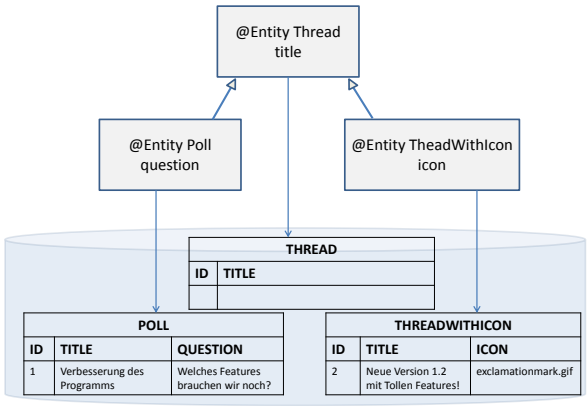


Abbildung 5.3: Veranschaulichung der Table Per Class Vererbungsstrategie

Joined bildet eine Tabelle für jede Klasse, geerbte Attribute werden jedoch nicht in Unterklassen übernommen. Das bedeutet, dass die Tabelle der Oberklassen auch Attribute der Unterklassen enthält. Die Tabelle der Oberklasse wird durch Fremdschlüsselbeziehungen mit den Tabellen der Unterklassen verknüpft. Dadurch müssen bei jeder Abfrage Joins über den Tabellen durchgeführt werden, was bei großen Hierarchien viel Rechenleistung beansprucht. Die Strategie ist dafür jedoch effizient bezüglich des Speicherverbrauchs und unterstützt polymorphe Abfragen. Abbildung 5.4 veranschaulicht, wie die Oberentitäten Thread und die beiden Unterentitäten Poll und ThreadWithIcon durch die Joined Vererbungsstrategie in relationale Datenbanken überführt würden.

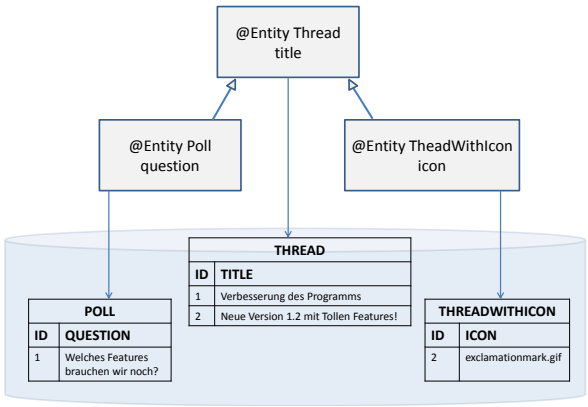


Abbildung 5.4: Veranschaulichung der Joined Vererbungsstrategie

Vom Generator werden für Generalisierungen standardmäßig Annotationen für die Single Table Strategie erzeugt, da diese Strategie polymorphe abfragen unterstützt und mehr Speicherplatz als Rechenleistung beansprucht, wobei Speicherplatz in modernen Computern in ausreichender Menge zur Verfügung steht.

Unterentitäten erben durch diese Annotationen alle Attribute und somit auch Assoziationen der Oberentitäten. Daher erhalten sie im EntityModel kein eigenes ID Attribut, sondern erben dies ebenfalls von ihrer Oberentität. Die Funktionalitäten des Controllers werden nicht vererbt. Da die JPA polymorphe Abfragen für zwei der drei Vererbungsstrategien unterstützt, können Controller-Methoden die zum Beispiel das Zuweisen von Assoziationen der Oberentität verwalten auch in Zusammenhang mit ihren Unterentitäten genutzt werden. Die Vererbungsstrategie Table Per Class sollte nur eingesetzt werden, wenn keine polymorphen Abfragen benötigt werden. Bei allen Vererbungsstrategien erhalten Sub-Entitäten zusätzlich eigene typspezifische create-, update- und delete-Methoden.

Im Rahmen der Generalisierung sollen zudem abstrakte Klassen eingeführt werden. Dabei handelt es sich um Klassen, die selber nicht instanziiert werden sollen. Daher werden für die zugehörigen Entitäten keine create-Methoden im Controller erstellt. Andere Verwaltungsmethoden, für zum Beispiel Assoziationen, werden normal erstellt, da die JPA polymorphe Abfragen auch für abstrakte Entitäten unterstützt.

6 Technische Umsetzung

Inhalt

Im Folgenden wird die technische Umsetzung der Generierung von Beziehungen im Gargoyle Codegenerator erläutert. Dabei werden vereinfachte Versionen der Meta-Modelle des Generators benutzt, die sich überwiegend auf die für die Beziehungen wichtigen Teile beschränken.

6.1 GeneratorModel

Wie in Abschnitt 3.2.1 beschrieben, stellt das GeneratorModel das erste Modell des Gargoyle Codegenerators dar und dient zum erstmaligen Einlesen des UML-Modells in den Generator.
Da zwar eine EMF-basierte Implementierung des UML-Meta-Modells [Ecl] existiert, deren Integration im Rahmen dieser Arbeit jedoch zu zeitaufwändig gewesen wäre, wurde zunächst eine der UML sehr nahe Erweiterung des GeneratorModels entwickelt. Abbildung 6.1 zeigt den für die Beziehungen relevanten Teil des weiterentwickelten GeneratorModels. Eine Beziehung wird hier durch die abstrakte Klasse „Relationship“ verkörpert, die eine Ursprungsklasse (originClass), von der die Beziehung ausgeht, und eine Zielklasse (targetClass), zu der die Beziehung hin führt, besitzt. Davon abgeleitet werden verschiedenen Arten von Beziehungen, zum einen die Generalisierung (Generalization), die außer der Ursprungs- und Zielklasse keine weiteren Attribute besitzt und zum anderen die Assoziation (Association).

Die Assoziation besitzt neben Ursprungs- und Zielklasse weitere Attribute. Der Name dient zur Anzeige im EMF-basierten Editor des Modells. Besitzt eine Assoziation keinen Namen, so wird lediglich die Zielklasse angezeigt. Wie in Abschnitt 2.3.3 erläutert, ist die Komposition eine spezielle Form der Assoziation. Da neben der Komposition aber keine weitere spezielle Assoziation eingeführt wurde, ist diese Eigenschaft durch ein einfaches Attribut realisiert worden, um unnötige Komplexität des Meta-Modells zu verhindern. Das Attribut *bidirectional* gibt an, ob die Assoziation bidirektional verlaufen soll, also ob sie nur von

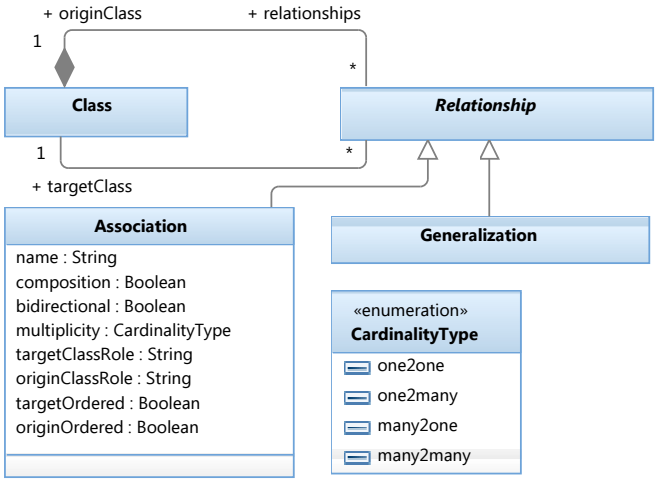


Abbildung 6.1: Meta-Modell für Beziehungen im GeneratorModel

der Ursprungs- zur Zielklasse oder auch in die entgegengesetzte Richtung navigierbar sein soll. Das Attribut *multiplicity* gibt die Multiplizität der gesamten Assoziation an. Hier wird nur zwischen eins (one) und viele (many) entschieden, da sich auch bei genauerer Unterscheidung keinen Unterschied für den weiteren Generierungsprozess ergeben würde. Schlussendlich muss der Benutzer noch Namen für Rollen der Ursprungs- und Zielklassen angeben und einstellen, ob die Klassen geordnet als Liste oder ungeordnet als Set zugreifbar sind. Während für unidirektionale Assoziationen nur die Zielklasse eine Rolle in der Assoziation benötigt, muss für bidirektionale Assoziationen auch eine Rolle für die Ursprungsklasse definiert werden, da daraus im folgenden Generierungsprozess die Namen der entsprechenden Attribute und Methoden für die Klassen generiert werden.

6.2 EntityModel

In Abschnitt 3.2.2 wurde beschrieben, dass das EntityModel im Gargoyle Codegenerator die persistenten Entitäten und deren Attribute sowie Queries für Datenbankabfragen enthält. Um semantische Beziehungen zu realisieren wurde das Modell um alle dafür wichtigen Annotationen erweitert und neue Objektklassen zur besseren Übersicht für den Benutzer des Generators und zur Vereinfachung des weiteren Generierungsprozesses erstellt.

In den Abbildungen 6.2 und 6.3 ist der für Beziehungen relevante Teil des EntityModels zu sehen. Abgebildet sind zunächst die Klassen Entity, deren Objekte aus Objekten der Klasse Class des EntityModels entstehen und die Klasse Attribute, deren Objekte aus den Klassen Attribute und Assoziation des GeneratorModels entstehen.
Beide Klassen können verschiedene Annotationen haben. Zudem wurden die

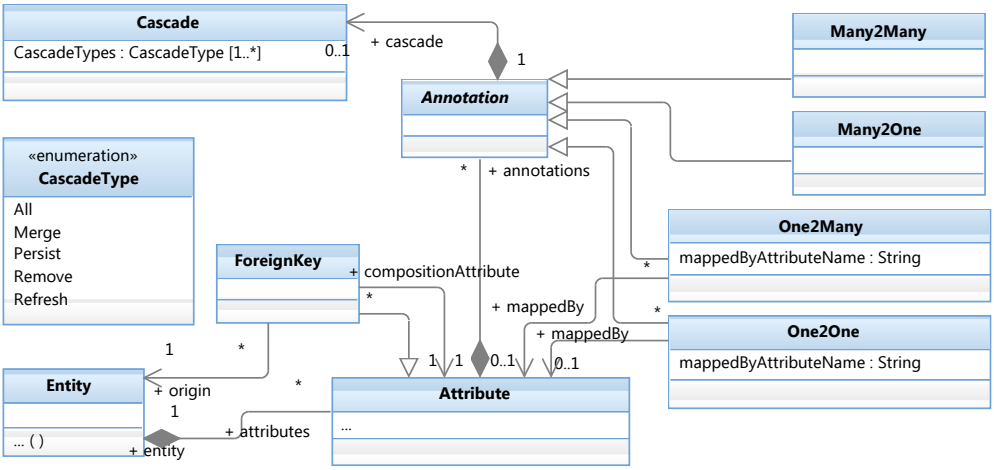


Abbildung 6.2: Meta-Modell für Assoziationen im EntityModel

Objektklassen „ForeignKey“ und „InheritedAttribute“ eingeführt, die Spezialisierungen der Klasse Attribute sind und Informationen für den Benutzer und den Generator enthalten. Sie werden im Editor grau angezeigt, da sie nicht manuell angepasst werden sollten, solange sich das assoziierte Attribut nicht verändert.

6.2.1 Assoziation und Komposition

Assoziationen und folglich auch Kompositionen werden im EntityModel durch Attribute mit entsprechenden Annotationen realisiert, zu sehen in Abbildung 6.2. Die vier Klassen One2One, One2Many, Many2One und Many2Many beschreiben die gleichnamigen Annotationen im Entitäten-Quellcode, durch die die JPA die Attribute in Datenbank-Relationen überführt. Das Attribut MappedBy wird für bidirektionale Assoziationen verwendet. Der MappedBy-Wert wird innerhalb einer Assoziation spezifiziert und verweist auf die Spalte in der Tabelle der assoziierten Entität, die die Assoziation enthält. Dadurch wird in der Tabelle der mit MappedBy annotierten Klasse keine weitere Spalte erstellt und die Assoziation nur von einer Klasse aus verwaltet. Dies ist allerdings nur für bidirektionale One2One sowie One2Many und der zugehörigen Many2One Relation möglich, da die Tabellen bei Many2Many Relationen nicht eindeutig zugeordnet werden können.

Zusätzlich können für die Assoziations-Annotationen Kaskadierungstypen erstellt werden. Nur der Typ Merge wird vom Generator automatisch erzeugt, weitere Kaskadierungstypen sind optional.

In Abschnitt 5.2 wurde die Einführung eines Fremdschlüsselkonzepts für Komponenten einer Komposition erläutert. Zur Verwaltung dieser Fremdschlüssel wurde die Objektklasse „ForeignKey“ eingeführt. ForeignKey-Instanzen werden

innerhalb der Entität-Instanzen aller Komponenten erzeugt. Dabei werden alle Primärschlüsselattribute des Kompositums, die in der Kompositionshierarchie über der Komponente liegen als ForeignKey für die Komponente erzeugt. Das ForeignKey Objekt kopiert alle Werte des Primärschlüsselattributes, auf dem es basiert und referenziert zusätzlich das Attribut sowie das Kompositionsattribut.

Durch das Fremdschlüsselkonzept ändern sich auch die Queries, die im EntityModel für jede Klasse erstellt werden. Dadurch, dass Komponenten nur innerhalb ihres Kompositums eindeutig sind, muss dies in die Datenbankabfrage der Komponente mit einbezogen werden. Das führt zu einem Join über die gesamte Kompositionshierarchie. Codebeispiel 6.1 zeigt beispielhaft die Abfrage für die Klasse Post aus Abbildung 5.1.

Quelltext 6.1: Query für die Entität der Klasse Post aus Abbildung 5.1

```
1 SELECT post
2 FROM Forum forum JOIN forum.threads thread JOIN thread.posts post
3 WHERE post.postID = :postID and thread.threadID = :threadID and forum.title = :title
```

Alternativ wäre auch eine Einführung von weiteren Attributen möglich gewesen, die die Fremdschlüssel der Kompositionshierarchie speichern. Ein Attribut, das alle Fremdschlüssel speichert hätte den Vorteil sehr schneller Datenbankabfragen und geringen Speicherbedarfs. Allerdings wäre die Nutzung der Attribute in Methoden weit weniger intuitiv für die Weiterentwicklung der generierten Software, da mehrere Attribute zu einem zusammengefasst würden.
Des Weiteren hätte jeder Fremdschlüssel als eigenes Attribut in den Entitäten von Komponenten gespeichert werden können. Das hätte schnelle Datenbankabfragen, aber einen hohen Speicherbedarf zur Folge gehabt. Außerdem müssten bei beiden Alternativen die erstellten Attribute stets mit den eigentlichen Fremdschlüsseln synchron gehalten werden.
Die Einführung von Kompositionsstrategien ähnlich der Strategien für Vererbung wäre denkbar. Für diese Arbeit wurde jedoch das Anfangs vorgestellte Konzept eingeführt, da der dazu generierte Quellcode am gebräuchlichsten für die Weiterentwicklung der Software ist.

6.2.2 Generalisierung

Die Generalisierung wird im EntityModel ebenfalls durch Annotationen realisiert. Das entsprechende Meta-Modell ist in Abbildung 6.3 zu sehen. Die Klasse EAnnotaion beschreibt Annotationen für Entitäten. Eine der Annotationen ist die @Inheritance Annotation, die durch die Klasse Inheritance beschrieben wird. Darin wird auch die Vererbungsstrategie festgelegt. Für die Strategie „SingleTable“ wird zudem das Attribut DiscriminatorColumn in der Klasse Inheritance und die Klasse DiscirminatorValue verwendet. Beides zusammen dient zum Erstellen der Spalte, die zur Unterscheidung der Unterklassen verwendet wird.

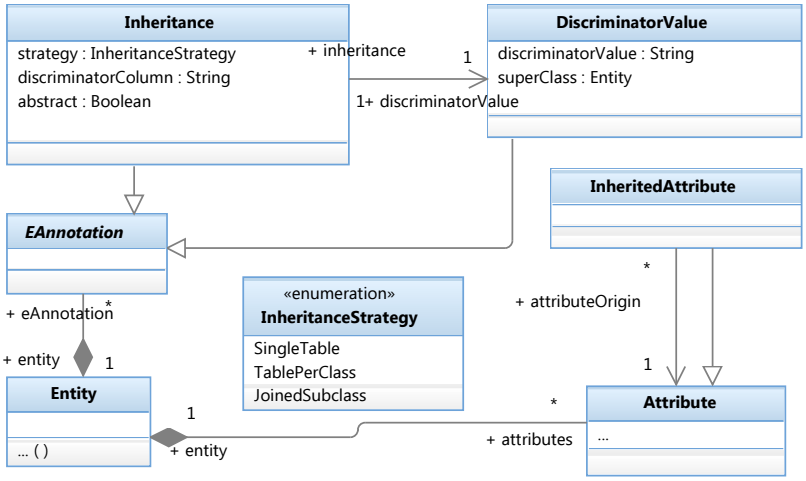


Abbildung 6.3: Meta-Modell für Vererbung im EntityModel

Die Objektklasse „InheritedAttribute“ wurde zusätzlich eingeführt, um geerbte Attribute zu verwalten. Dazu übernehmen InheritedAttributes alle Eigenschaften des originalen Attributs und referenziert es zusätzlich.

6.2.3 Transformation des GeneratorModels zum EntityModel

Die Transformation vom GeneratorModel ins EntityModel gestaltet sich im Vergleich zu den restlichen Transformationen relativ einfach, da größten Teils Objekte erzeugt werden, die die Attributwerte der Klasse des GeneratorModels übernehmen, auf den sie basieren. Die wichtigste Ergänzung im EntityModel sind die Annotationen. Quellcode 9.1 im Anhang zeigt die Extension, die die Annotationen der Assoziationsattribute inklusive des Kaskadierungstypen Merge erzeugt. Die Extension erhält die Assoziation des GeneratorModels als Eingabe und ruft die create-Extension der Annotationen entsprechend der Multiplizität der Assoziation auf.

Das Erzeugen der InheritedAttribute- und ForeignKey-Instanzen geschieht rekursiv, um Fremdschlüssel- bzw. geerbte Attribute auch für Kompositions- bzw. Vererbungshierarchien richtig zu erstellen. Dazu werden für beide Objekte je zwei Extensions erstellt. Die erste Extension wird nach dem Erstellen aller Entitäten und Attribute mit den jeweils obersten Elementen der Hierarchien aufgerufen. Für Kompositionen sind dies Komposita, die selber keine Komponenten sind und für Generalisierungen sind dies Oberklassen, die selber keine Unterklassen sind. In dieser Extension werden dann alle relevanten Attribute, die zu InheritedAttribute bzw. ForeignKey Objekten werden sollen und die direkten Nachfolger in der Hierarchie an die zweite Extension übergeben. Diese Extension erstellt dann die neuen Objekte und ruft sich rekursiv mit den zusätzlichen Attributen der Hierarchienachfolger und deren relevanten Attributen auf. Für das Erstellen der Fremdschlüssel müssen zudem Generalisierungen beachtet werden,

da diese nicht direkt über Kompositionen erreichbar sind, aber dennoch ebenfalls Fremdschlüssel benötigen. Fremdschlüssel dürfen nicht über InheritedAttribute Objekte verwaltet werden, da sie dann nicht mehr als Fremdschlüssel-Objekte identifizierbar wären. Der Quellcode für die gerade beschriebene Erzeugen der Fremdschlüssel-Attribute findet sich im Anhang 9.2.

6.2.4 Generierung der Domainen Klassen aus dem EntityModel

Für die Generierung der Domain Klassen werden alle Objekte im EntityModel in entsprechenden Quellcode umgesetzt. Quellcode 6.2 zeigt Beispielhaft, wie eine One2One Annotation durch ein Xpand Template in entsprechenden Quellcode mit Kaskadierungstyp und MappedBy Annotation umgesetzt wird.

Quelltext 6.2: XPand Template für die Generiung von One2One Annotationen im Quellcode

```
1 «DEFINE annotation FOR entitymodel::One2One»
2         @OneToOne«IF this.cascade != null»(cascade={
3         «EXPAND cascadeType FOREACH this.cascade.cascadeTypes SEPARATOR ","»}
4         «IF this.mappedBy != null», mappedBy="«this.mappedBy.name»")
5         «ELSE»)«ENDIF»«ELSEIF this.mappedBy != null»(mappedBy="«this.mappedBy.name»")
6         «ENDIF»
7 «ENDDEFINE»
```

Da für geerbte Attribute auch die Zugriffsmethoden (Setter und Getter) geerbte werden, bleibt das zugehörige Template leer. Quellcode 9.3 im Anhang zeigt das Template zur Generierung der Zugriffsmethoden für Fremdschlüssel. Der Quellcode wird nur für nicht geerbte Fremdschlüssel erstellt, da die Zugriffsmethoden auch für geerbte Fremdschlüssel vererbt werden. Ist die direkte Komposition zur Entität die den Fremdschlüssel erhält eine unidirektionale Komposition, so kann die Entität der Komponente nicht zur Entität des Kompositums navigieren. In diesem Fall werden set- und get-Methode für das Attriubt generiert, das den Wert des Fremdschlüssels für die Komponente speichert. Ist die Komposition navigierbar, wird nur eine get-Methode erzeugt.

6.3 EJB-GeneratorModel

In Abschnitt 3.2.4 wurde beschrieben, dass das EJB-GeneratorModel lediglich dazu dient Informationen für die Generierung des EJB-Models zu parametrieren. Dementsprechend fällt auch die Umsetzung der Assoziationen kompakt aus. Das EJB-GeneratorModel wurde für Assoziationen um die Klasse Association ergänzt, die alle nötigen Informationen für die Erzeugung der Methoden im EJBModel enthält.

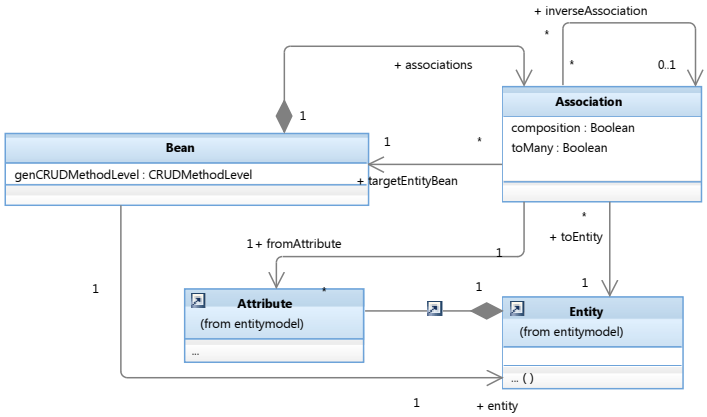


Abbildung 6.4: Meta-Modell für Beziehungen im EJB-GeneratorModel

Abbildung 6.4 zeigt das Meta-Modell für Assoziationen im EJB-Generatormodel. Die Klasse Association verknüpft die Attribute einer Entität, die eine Assoziation darstellen mit der entsprechenden Zielentität. Außerdem wird der Assoziationstyp durch das Attribut composition festgehalten und das Attribut toMany gibt Auskunft darüber, ob es sich beim Ziel der Assoziation um ein einzelnes oder mehrere Elemente handelt. Für bidirektionale Assoziaitonen wird das Association-Objekt der Gegenseite referenziert.
Für Kompositionen wird ein Attribut innerhalb der Klasse Association gesetzt. Die Generalisierung benötigt keine zusätzlichen Informationen.

6.4 EJBModel

In Abschnitt 3.2.4 wurde beschrieben, dass das EJBModel ein Modell zur Beschreibung der EJBs ist, die für ein Informationssystem auf Geschäftsebene generiert werden sollen. Die Funktionalität der Assoziationsmethoden wird dabei ausschließlich im Controller-Paket des EJBModels realisiert.

6.4.1 Veränderungen im Controller des EJBModels

Das Zuweisen bzw Aufheben von Assoziationen sowie die Semantik von Kompositionen wird in Controller-Methoden im Controller-Paket des EJBModels umgesetzt. Im Rahmen dieser Arbeit wurde das Controller-Paket des EJBModels komplett überarbeitet. Zuvor besaß der Controller ein typisiertes Methodenkonzept. Das heißt fetch-, create-, update- und delete-Methoden besaßen eigene Methodentypen. Diese wiederum besaßen für jede Aktion innerhalb der Methode (wie zum Beispiel Notnull-Checks der Parameter) eine weitere Klasse. Methodenaufrufe innerhalb der Controller-Methoden wurden über Application-Klassen abgewickelt, die den Aufruf an die entsprechende Methode weiterleite-

ten. Dieses typisierte Methodenkonzept sollte durch ein allgemeineres Konzept abgelöst werden. Ziel dabei war es neue Controller-Methoden, wie die Assoziationsmethoden, die in dieser Arbeit entwickelt wurden, unabhängig ihres Typs im Controller modellierbar zu machen, um das Controller-Paket nicht mit Methodentypen zu überladen. Dies sollte zudem die Generierung des EJB Quellcodes in der Modell zu Text Transformation erleichtern, da dort ebenfalls zu jedem Methodentyp ein eigenes Template existierte und nach der Überarbeitung auch diese Templates durch ein allgemeines Template abgelöst würden.

Abbildung 6.5 zeigt das überarbeitete Controller-Paket des EJBModels. Zu sehen ist zunächst die Klasse *Method* aus dem EJBModel. Methoden können mehrere Parameter besitzen. Davon abgeleitet werden die ControllerMethod, die zusätzlich auf die zugehörige Entität verweist und die AssociationMethod, die zu der Entität der ControllerMethod das zugehörige Assoziations-Attribut assoziiert.

Contoller-Methoden werden nicht mehr weiter in Typen unterteilt. Stattdessen erhält jede Controller-Methode eine Liste von Commands. Zurzeit existieren vier Sorten von Commands: Notnull-Checks für Parameter, das setzen von Attributen auf den Wert eines Parameters sowie das Aufrufen einer create- oder anderen Methode. Create-Methoden-Aufrufe werden separat behandelt, da diese die entsprechende Methode zum Erstellen der Entität aus dem EntityModel benötigen. Methodenaufrufe benutzen ParameterMappings, um die Werte eigenen Parameter auf die Parameter der aufgerufenen Methode abzubilden.

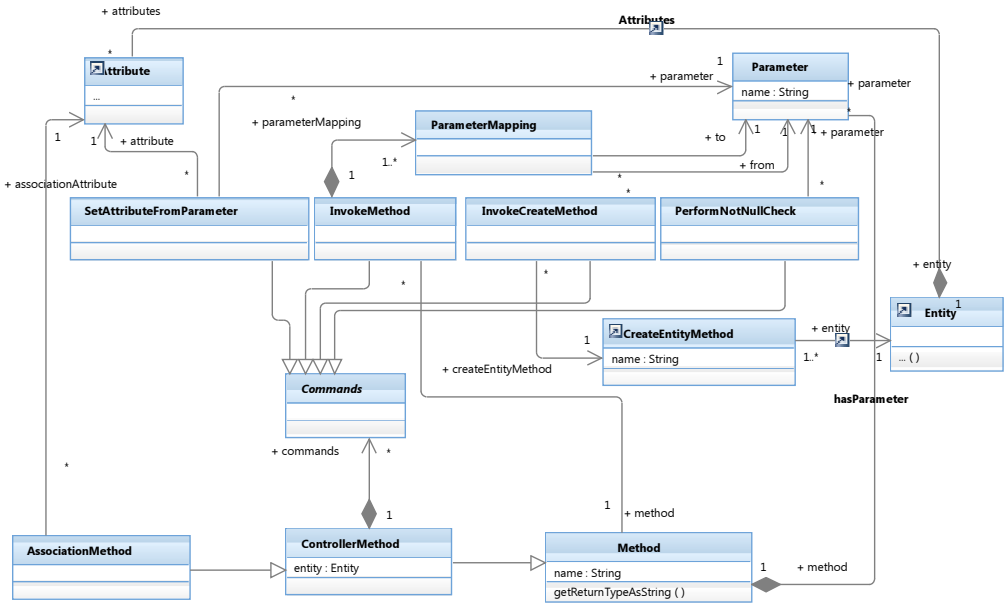


Abbildung 6.5: Vereinfachte Darstellung des Controller Pakets im EJBModel

Insgesamt kommt das EJBModel seinem Zweck, frei Methoden für die Generierung als EJB-Quellcode gestalten zu können, ein Stück näher. Dennoch fehlen immer noch einige Aspekte, wie ein Variablenkonzept und setzten von return-Werten für Methoden, um Methoden wirklich unabhängig ihres Typs gestalten zu können. Die Entwicklung hin zu völliger Unabhängigkeit vom Typ der Methode hätte jedoch den Rahmen dieser Arbeit überschritten und bleibt zunächst offen. Daher können die Templates der Modell zu Text Transformation nicht völlig allgemein gestaltet werden. Viele Commands der Methoden werden immer noch im semantischen Zusammenhang interpretiert und entsprechend transformiert.

6.4.2 Transformation des EJB-GeneratorModels zum EJBModel

Die Transformation des EJB-GeneratorModels zum EJBModel ist sehr komplex. Das Erzeugen der Assoziationsmethoden unterscheidet sich grundlegend von denen der Komposition. Wechselwirkungen mit der Generalisierung müssen beachtet werden, allgemein macht die Generalisierung jedoch nur einen geringen Teil der Transformation aus.

Assoziation

Für jede einfache Assoziation wird im Controller der zugehörigen Entität eine Methode zum Zuweisen (assign) und Auflösen (unassign) der Assoziation erstellt. Diese Methoden werden gleich modelliert und nur durch ihren Namen in den Templates der drauf folgenden Modell zu Text Transformation unterschiedlich interpretiert. Damit die Namen der generierten Methoden eindeutig und später für Entwickler leicht zu benutzen sind, setzt sich dieser aus „assign“ bzw. „unassign“, dem Name der Zielentität, „To“, Name der Ursprungsentität und Rolle der Zielentität in der Assoziation zusammen. Für die Assoziation von der Klasse Forum zur Klasse Thread, in der Thread die Rolle „threads“ hat (siehe Abbildung 6.6), würde zum Beispiel die Zuweisungsmethode „assignThreadToForumThreads(...)“ generiert.

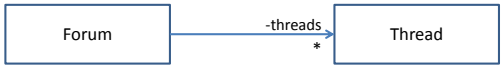


Abbildung 6.6: Beispiel einer Forum zu Thread Assoziation

Anschließend werden die identifizierenden Attribute der Ursprungsklasse und Zielklasse als Parameter hinzugefügt. Die Parameter für die Zielklasse erhalten das Präfix „target“, um sie von den Parametern der Ursprungsklasse unterscheiden zu können. Dieses Präfix wird für den Gargoyle Codegenerator reserviert.

Benutzer werden jedoch bereits im GeneratorModel durch einen Fehler darauf hingewiesen, sollten sie ein reserviertes Präfix verwendet haben. Für alle Parameter wird ein Notnull-Check angelegt.
Alternativ dazu hätten auch Methoden erzeugt werden können, die statt den identifizierenden Attributen direkt die Entitäten als Objekte erhalten. Um diese Methoden zu nutzen müsste ein Entwickler jedoch vor jedem Aufruf einer der Methoden selbst die entsprechenden Objekte aus den Tabellen der Datenbank erzeugen. Das würde sich wiederholenden Programmieraufwand bedeuten, der durch den Generator eigentlich vermieden werden sollte. Darum wurden zunächst Methoden für identifizierende Attribute generiert. Eine zukünftige Erweiterung des Generators um Assoziationsmethoden, die direkt konkrete Objekte als Parameter erhalten, wird jedoch nicht ausgeschlossen.

Ist die Assoziation bidirektional und es wurden bereits assign/unassign-Methoden für die andere Assoziationsrichtung erstellt, so wird lediglich ein Aufruf der bereits erstellten Methode mit allen Parametern erzeugt. Handelt es sich um eine unidirektionale Assoziation oder wurden noch keine Methoden für die Assoziation erstellt, so erhalten die Methoden Methodenaufrufe für die fetch-Methoden der Ursprungs- und Zielentität mit den zugehörigen ParameterMappings, die die benötigten Parameter der assign/unassign Methode auf die Parameter der fetch-Methode abbildet. Abschließend wird abhängig vom Assoziationstyp ein update-Aufruf für eine der beteiligten Entitäten erzeugt. Durch den Kaskadierungstyp Merge aus dem EntityModel wird durch den Aufruf auch die Tabelle der assoziierten Entität in der Datenbank aktualisiert.

In den delete-Methoden eines Controllers werden Aufrufe für alle unassign-Methoden einfacher Assoziationen in diesem Controller erzeugt, um die Assoziationen der Entitäten vor dem Löschen ordnungsgemäß aufzulösen.

Da die Erzeugung der assign- und unassign-Methoden sehr komplex ist, zeigt Abbildung 6.7 den Kontrollflussgraphen für die zuvor beschriebene Generierung der Unassign-Methode. Die Generierung der Assign-Methode geschieht größten Teils analog, lediglich die Delegation an die Delete-Methode der Komponente im Falle einer Komposition wird ausgelassen.

Zunächst werden einige Informationen des EJB-GeneratorModels übernommen und der Name für die zukünftige unassign-Methode generiert. Anschließend werden je nach Art der Assoziation Parameter für die Signatur der Methode erzeugt. Einfache Assoziationen erhalten die identifizierenden Attribute von Ursprungs- und Zielentität, während bei Kompositionen einzelne Parameter der Zielentität eventuell schon durch Fremdschlüssel der Ursprungsentität als Parameter vorhanden sind. Alle Parameter erhalten Notnull-Checks, da Entitäten eindeutig anhand der Parameter identifizierbar sein müssen. Anschließend werden Methodenaufrufe innerhalb der unassign-Methode generiert. Deren Typ hängt ebenfalls vom Typ der Assoziation ab. Kompositionen erhalten einen Aufruf der delete-Methode der Komponente, Assoziationen erhalten fetch- und update-Methoden der Entitäten oder, falls existent, einen Aufruf der bereits existieren-

den Methode der Rückrichtung. Abschließend wird die zu der unassign-Methode gehörige Facade-Methode im Facade-Paket erzeugt.

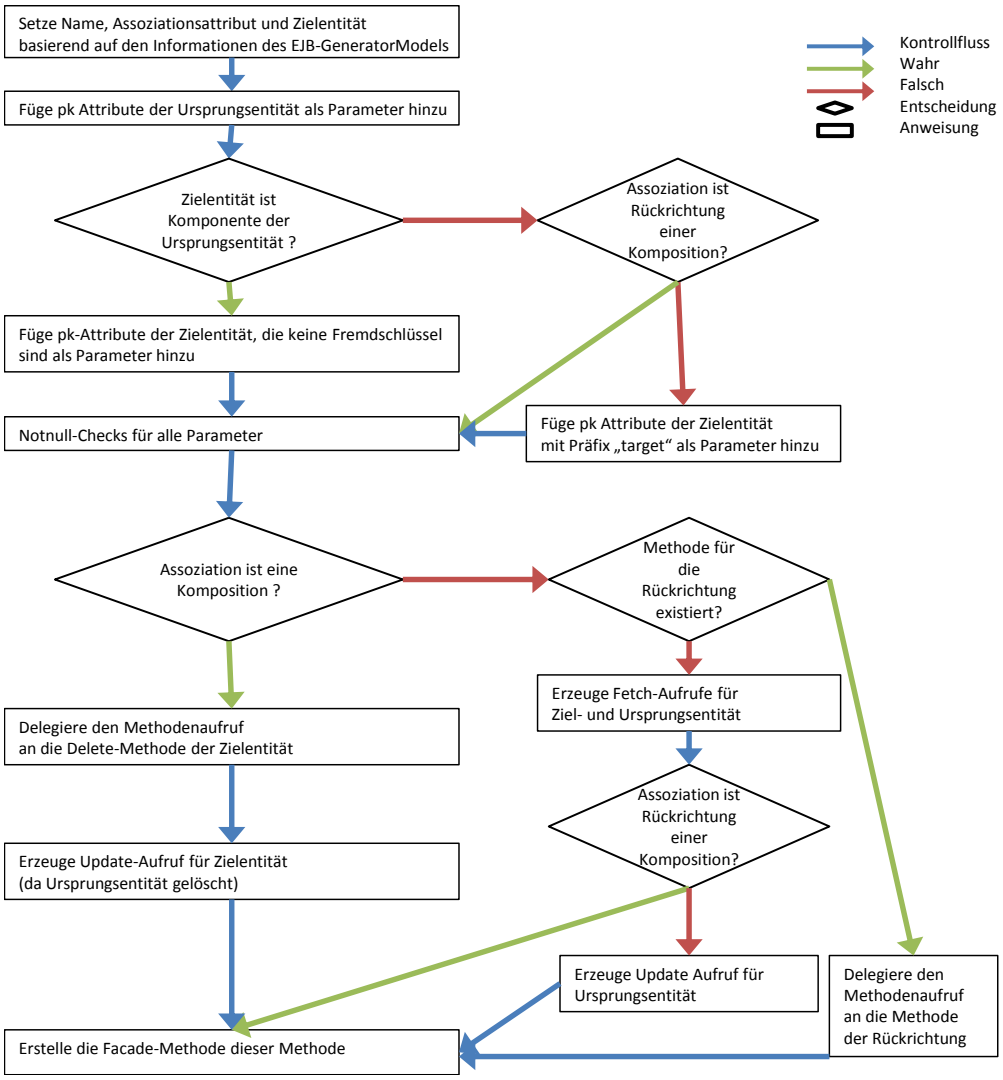


Abbildung 6.7: Kontrollflussgraph für das Erstellen der Unassing-Methoden im EJBModel Controller

Komposition

Da Kompositionen eine andere Semantik als einfache Assoziationen haben, werden für diesen speziellen Assoziationstyp andere Methoden generiert. Für jede Komposition wird lediglich eine neue unassign-Methode zum Auflösen der Komposition erzeugt. Diese unassing-Methode erhält alle identifizierenden Attribute der Komponente als Parameter. Da die Komponente ohne ihr Kompositum nicht existieren kann, wird innerhalb der Methode nur ein Aufruf der delete-Methode der Komponente generiert. Dieser Aufruf wird immer in der unassign-Methode

des Kompositums erzeugt, da diese Methode sowohl für unidirektionale, als auch für bidirektionale Kompositionen existiert. Folglich delegiert die eventuell zusätzlich generierte unassign-Methode im Controller der Komponente ihren Aufruf an die unassign-Methode des Kompositums.

Zusätzlich wird die create-Methode der Komponente um einen fetch- und einen update-Aufruf für das zugehörige Kompositum erweitert. Dies dient dazu die Komponente beim Erstellen mit ihrem Kompositum zu assoziieren. Das eigentliche Zuweisen wird wieder durch Templates der Modell zu Text Transformation interpretiert.

Anders als bei einfachen Assoziationen wird die unassign-Methode von Kompositionen nicht beim Löschen der Komponente aufgerufen, da dies eine Endlosschleife verursachen würde. Stattdessen wird die Komposition innerhalb der delete-Methode aufgelöst bevor die Komponente gelöscht wird.

Generalisierung

Da Unterklassen auch Kompositionen von Oberklassen erben, werden Erweiterungen der create- und delete-Methode für Kompositionen in die Controller der Entitäten von Unterklassen übernommen. Das heißt, dass der Generator nach der Generierung der Methodenaufrufe die gleichen Methodenaufrufe ebenfalls für alle Unterklassen generiert.

6.4.3 Generierung des EJB-Quellcodes aus dem EJBModel

Quellcode 9.4 im Anhang zeigt beispielhaft wie assign-Methoden aus dem Controller des EJB-Models in Quellcode übersetzt werden. Methodenaufrufe innerhalb der assign-Methode werden durch eigene Templates umgewandelt. Der Quellcode, der die Entitäten einander zuweist wird direkt im Template umgesetzt.
Die Erweiterungen der create- und delete-Methoden für Kompositionen werden durch Aufrufe der entsprechenden Templates und dem Code für die Zuweisung der Komposition realisiert, wie beispielhaft in Quellcode 9.5 im Anhang für die Generierung der fetch-Methoden Aufrufe zu sehen ist.

7 Evaluation

Inhalt

Ziel dieser Arbeit war die Erweiterung des Gargoyle Codegenerators um semantische Beziehungen. Dazu wurde ein Konzept für die Generatorerweiterung entwickelt und in den bestehenden Gargoyle Codegenerator integriert. Zur Evaluation der Codegenerierung wurde der Gargoyle Codegenerator in der Entwicklung mehrerer Informationssysteme am LuFGi3 verwendet. Diese Informationssysteme besitzen eine Vielzahl von Kompositionen. Wichtige Punkte für die Evaluation des Programms waren Korrektheit des generierten Quellcodes sowie die Zeitersparnis durch die Generierung des Quellcodes, insbesondere des Kompositions- und Vererbungsquellcodes.

7.1 Vorgehen

Im Rahmen dieser Arbeit wurde der Gargoyle Codegenerator in einer ersten Version um semantische Beziehungen erweitert. Diese Version wurde für die Generierung des Grundgerüsts einer Business Goal Management Umgebung für die Masterarbeit „Development of an EJB-Based Business Goal Management Environment“ von Bogdan Kyryliuk [Kyr12] verwendet. Basierend auf dem Ergebnis dieser ersten Anwendung wurden einige Verbesserungen des Generators vorgenommen.
Anschließend wurde der Generator mit dem Reference Model Integration System von Simona Jeners [PL11] getestet. Es wurden manuelle Test im Rahmen der Entwicklung des Informationssystems sowie generierte Integrationstests im Rahmen der Arbeit von Steffen Conrad [Con12] durchgeführt. Dabei traten einige Fehler im generierten Quellcode auf. Daraufhin wurden insbesondere das Zuweisen und Löschen von Komponenten überarbeitet.

Abschließend wurde das Grundgerüst des Metric Definition Integraton and Configuration (MeDIC) Wiki der Bachelorarbeit „Konzeption einer Internetplattform zur Dokumentation von Metriken“ von Michael Schlimnat [Sch12] und das Grundgerüst des MeDIC Dashboards der Masterarbeit „Konzeptionelle Erweiterung von Projektdashboards für unerfahrene Anwender“ von Frederic Evers [Eve12] mit Hilfe des Gargoyle Codegenerators generiert. Das MeDIC Wiki wurde mit

generierten Integrationstests im Rahmen der Arbeit von Steffen Conrad [Con12] getestet. Das MeDIC Dashboard hingegen befand sich zur Zeit der Arbeit noch in einem unausgereiften Entwicklungsstadium und wurde darum nur manuell durch den Entwickler getestet.

7.2 Evaluation der Codegenerierung

Für alle in Abschnitt 7.1 vorgestellten Programme wurden jeweils die fünf Pakete Controller, DOA, Domain, Exceptions und Facade generiert. Die Anzahl generierter Lines of Code(LOC) und Klassen sowie die Anzahl der jeweils im Modell der Informationssysteme genutzten Beziehungstypen findet sich in Tabelle 7.1.

Informationssystem	**GenLOC**	**Klassen**	**Assoz.**	**Kompos.**	**General.**
Business Goal Management (BGM)	9.700	88	9	11	0
Reference Model Integration (RMI)	10.539	73	15	8	0
MeDIC Wiki	4.268	53	6	3	0
MeDIC Dashboard	8.329	99	14	3	4

Tabelle 7.1: Generierte Lines of Code (GenLoC), Anzahl der generierten Klassen und Anzahl der jeweils genutzten Beziehungstypen der für die Evaluation generierten Codegerüste

Die Tabelle zeigt, dass die drei Systeme BGM, RMI und MeDIC Dashboard mit 70 bis 100 Klassen über ein großen Modell verfügen. Für das MeDIC Wiki wurden nur 53 Klassen generiert, da das zugrunde liegende Modell einen geringen Umfang besitzt. Obwohl BGM und RMI über weniger Klassen als das MeDIC Dashboard verfügen, wurden für beide Systeme ca. 2000 LOC mehr generiert als für das MeDIC Dashboard. Das ist mit der hohen Anzahl an Assoziationen und insbesondere Kompositionen in BGM und RMI zu erklären. Zusätzlich benutzt das MeDIC Dashboard Generalisierungen. Da für vererbte Attribute und Assoziationen kein zusätzlicher Quellcode generiert wird, trägt dies zur geringeren Anzahl LOC des MeDIC Dashboards bei.

7.2.1 Testen des generierten Quellcodes

Im generierten Quellcode des Informationssystems von Bogdan Kyryliuk und Simona Jeners traten anfangs einige Fehler auf. Auf Basis von manuellen Tests für das Reference Model Integration System wurde der Generierungsprozess verbessert, bis schließlich ein Codegerüst generiert werden konnte, das in manuellen Tests fehlerfrei betrieben werden konnte.

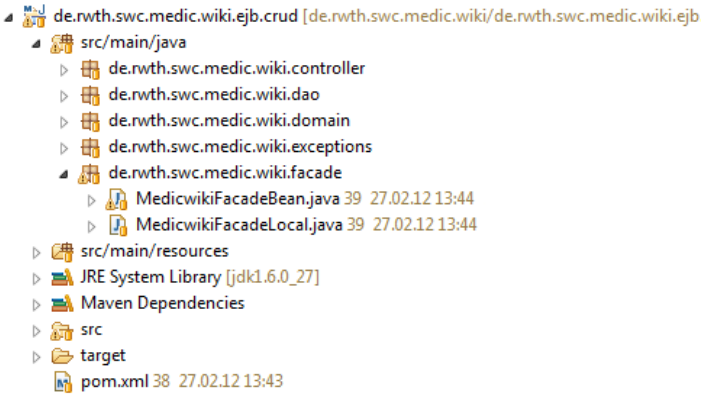
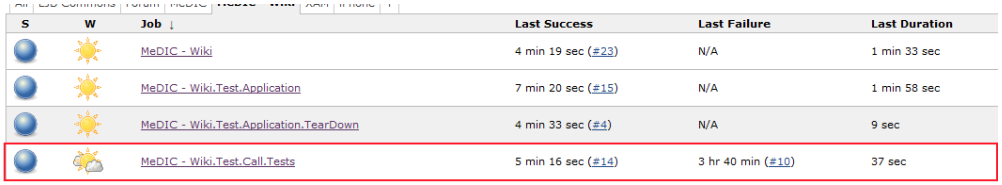


Abbildung 7.1: Kompilierbar generierte Pakete des MeDIC Wiki



S	W	Job ↓	Last Success	Last Failure	Last Duration
		MeDIC - Wiki	4 min 19 sec (#23)	N/A	1 min 33 sec
		MeDIC - Wiki.Test.Application	7 min 20 sec (#15)	N/A	1 min 58 sec
		MeDIC - Wiki.Test.Application.TearDown	4 min 33 sec (#4)	N/A	9 sec
		MeDIC - Wiki.Test.Call.Tests	5 min 16 sec (#14)	3 hr 40 min (#10)	37 sec

Abbildung 7.2: Erfolgreiche MeDIC Wiki Integrationstests

Um die Implementierung weiterhin auf mögliche Fehler zu prüfen, wurden Integrationstests für das Reference Model Integration System generiert. Die Testgenerierung wurde parallel in der Diplomarbeit „Ein Ansatz zum modellgetriebenen Test von EJB-basierten Informationssystemen“ von Steffen Conrad entwickelt [Con12]. Nachdem auch hier keine weiteren Fehler auftraten, wurde die überarbeitete Version des Generators mit den Systemen von Michael Schlimnat und Frederic Evers getestet.
Abbildung 7.1 ist ein Screenshot der generierten Pakete des MeDIC Wiki. Alle Pakete kompilieren ohne Fehler. Warnungen werden durch überflüssige Imports erzeugt, die aber im Betrieb kein Problem darstellen. Des Weiteren zeigt Abbildung 7.2 das Ergebnis eines Integrationstest des MeDIC Wiki. Darin ist zu sehen, dass das MeDIC Wiki die vom Integrationstest überprüften Anforderungen erfüllt. Eine detaillierte Beschreibung der Integrationstests findet sich in der Diplomarbeit von Steffen Conrad [Con12].

7.2.2 Zeitersparnis

Abschließend wurde eine Befragung der Entwickler der Informationssysteme durchgeführt. Ziel dabei war es zu erfahren wie lange die Entwicklung, insbesondere semantischer Beziehungen, ohne den Generator gedauert hätte und ob die generierte Implementierung den Erwartungen entsprach.

Generell hängt die Zeitersparnis vom Umfang und der Funktionalität des Informationssystems ab. Verfügt das Informationssystem nur über einfache CRUD-Funktionalität, erspart das Generieren fast die gesamte Entwicklungszeit. Je komplexer das Modell, desto mehr Zeit wird eingespart, da Modelle zu Beginn der Entwicklung auch ohne Nutzung des Generators erstellt würden. Der folgende Generierungsprozess würde nur einige Minuten im Vergleich zu mehreren Stunden Entwicklungszeit im Falle der manuellen Implementierung in Anspruch nehmen.
Eine Befragung der Entwickler der zur Evaluation generierten Systeme ergab eine geschätzte Entwicklungszeit von 40 bis 60 Stunden für das Grundgerüst älterer Versionen vergleichbarer Anwendungen. Die Implementierung der Kompositionen war sehr aufwändig und machte dabei bis zu 50% der Entwicklungszeit aus. Generalisierungen wurden weniger häufig verwendet und deren Implementierung ist etwas weniger umfangreich.

Dem gegenüber steht eine geringe Entwicklungszeit von ca. 30 Minuten für die Eingabe des Modells in den Gargoyle Codegenerator. Die Generierung dauert anschließend nur einige Minuten, selbst wenn Änderungen an einzelnen Modellen durchgeführt werden. Der Aufwand für die Weiterentwicklung des generierten Codegerüsts um spezielle Funktionalität bleibt gleich oder wird durch das einheitlich generierte Codegerüst weiter vereinfacht.

7.2.3 Konzept der Erweiterung

Es zeigte sich, dass das GeneratorModel vor der Transformation ins EntityModel durch Einschränkungen des Generators bezüglich Attribut- und Rollennamen oft angepasst werden musste. Viele Klassen besaßen oft gleiche identifizierende Attributnamen wie „name“ oder „titel“. In einem Forumsystem zum Beispiel haben meist sowohl das Forum selbst, als auch Komponenten des Forums wie Threads und Posts einen Titel, der als identifizierendes Attribut genutzt werden könnte.

Für tiefe Kompositionshierarchien erlaubt die Aufführung aller Attribute und Fremdschlüssel als Parameter eine eindeutige Zuordnung aller benötigten Werte und liefert somit eine gute Übersicht. Würden hingegen alle Werte als Objekt übergeben, wäre die Methodensignatur wesentlich kleiner.

Der generierte Quellcode ließ sich in alle zugehörigen Systeme integrieren. Die Evaluation ergab zudem, dass die generierten Methodennamen eine eindeutige Zuordnung von Methode zu Assoziation erlauben und die Entwicklungsarbeit mit dem generierten Quellcode unterstützen.

8 Zusammenfassung und Ausblick

Inhalt

Nachdem die Generatorerweiterung und deren Evaluation beschrieben wurde, soll dieses Kapitel die Arbeit zusammenfassen und einen Ausblick auf mögliche weitere Entwicklungen am Gargoyle Codegenerator geben.

8.1 Zusammenfassung

Diese Arbeit beschreibt die Erweiterung des Gargoyle Codegenerator um Komposition und Generalisierung. Die Erweiterung basierte auf dem zuvor in der Arbeit „Generierung von Web-basierten Prototypen für Geschäftsanwendungen“ entwickelten Codegenerator.

Dazu wurden zunächst bestehende ähnliche Generatoren untersucht und aufbauend darauf ein eigenes Konzept für die Einführung semantischer Beziehungen im Gargoyle Codegenerator entwickelt. Im Rahmen der Einführung wurde ein erweitertes Konzept für einfache Assoziationen erarbeitet. Für einfache Assoziationen werden Methoden zur Zuweisung und Auflösung der Assoziation generiert. Um Assoziationen einheitlich zu verwalten, wurden Assoziationsmethoden für bidirektionale Assoziationen so entwickelt, dass sie ihren Aufruf an andere Methoden delegieren.

Komposition und Generalisierung besitzen spezielle Semantiken, die durch die Erweiterung bzw. Abänderung bisher generierter Methoden umgesetzt wurden. Komponenten von Kompositionen wurden als schwache Entitäten umgesetzt, die nur in Zusammenhang mit ihrem Kompositum eindeutig sind. Die Komposition wird beim Erzeugen der Komponente zugewiesen. Das Auflösen der Komposition resultiert in der Löschung der Komponente.
Die Generalisierung wurde hauptsächlich durch Annotationen in den Klassen der Entitäten realisiert. Zudem wurde ein Konzept für abstrakte Klassen und vererbte Attribute entwickelt.

Die Konzepte wurden anschließend unter Berücksichtigung der speziellen Architektur des Generators umgesetzt und durch die Verwendung des Generators

in der Entwicklung mehrerer Informationssysteme am LuFGi3 evaluiert. Dabei zeigte sich durch Testen des generierten Quellcodes und Befragung der Entwickler der Informationssystem, dass die Umsetzung des Konzeptes erfolgreich war und insbesondere die Generierung des Kompositionsquellcodes ein großer Teil der Entwicklungszeit einsparen ließ.

8.2 Ausblick

Durch die Erweiterung im Rahmen mehrerer Bachelor- und Diplomarbeiten hat sich der Gargoyle Codegenerator zu einem umfangreichen Werkzeug für die Generierung von Informationssystemen entwickelt.

Basierend auf dieser Arbeit stehen noch einige Verbesserungen aus. Zunächst sollte die Überarbeitung des EJBModels abgeschlossen werden, damit EJB-Methoden vollständig modelliert werden können. Im Zuge dessen muss ebenfalls die Generierung des EJB-Quellcodes durch XPand-Templates überarbeitet werden. Zusätzlich könnte eine Funktion für die Generierung von Assoziationsmethoden für konkrete Objekte erstellt werden, um Fremdschlüssel für erfahrene Programmierer einfacher zu verwalten.

Assoziationsmethoden delegieren für bidirektionale Assoziationen ihren Aufruf untereinander, dies könnte auch für Unterentitäten übernommen werden, die ihren Aufruf zunächst an Methoden des gleichen Typs der Oberentitäten delegieren. Dies hätte den Vorteil, dass Entwickler Ereignisse, die beim Erstellen, Updaten oder Löschen von Oberentitäten auftreten sollen auch für Unterentitäten übernommen würden.

Da Informationssysteme selbst in ihrer Funktionalität erweiterbar sind, bietet auch der Codegenerator weiteres Potential für die standardmäßige Generierung weiterer Funktionen. Bei vielen Informationssystemen besteht der Bedarf Informationen zu versionieren. Da Versionsverwaltungen zwar modellabhängig sind, aber die Funktionalität an sich immer gleich ist, wäre dies eine mögliche weitere Erweiterung für den Gargoyle Codegenerator.

9 Anhang

Inhalt

9.1 Quellcode

Quelltext 9.1: Xtend Transformation für Annotationen von Assoziationen

```
entitymodel::Annotation createAnnotation(generatormodel::Association association):
  let cascadeTypes = {"merge"} :
  switch{
    case association.multiplicity.toString() == "one2one" :
                association.createOne2One(cascadeTypes, null)
    case association.multiplicity.toString() == "many2one" :
                association.createMany2One(cascadeTypes)
    case association.multiplicity.toString() == "one2many" :
                (association.bidirectional
                  ? association.createOne2Many(cascadeTypes, association.originClassRole)
                  : association.createOne2Many(cascadeTypes, null))
    case association.multiplicity.toString() == "many2many" :
                association.createMany2Many(cascadeTypes, association)
    default : null
  };
```

Quelltext 9.2: Xtend Extensions zum Erzeugen von ForeignKey Objekten

```

/*Diese Extension dient zum erstmaligen Aufruf der Extention, die
        rekursiv die ForeignKey Attribute erzeugt.*/
Void addForeignKeysForDirectContained(entitymodel::Entity entity,
                                      List[entitymodel::Entity] allEntities) :
  allEntities.attribute.select(e|e.composition).type.exists(e|e == entity.name)
    ? null
    : allEntities.select(e|entity.attribute.select(e|e.composition
                && e.metaType != entitymodel::InheritedAttribute).type.exists(t|t == e.name))
                .addForeignKeysForContained(entity.attribute.select(e|e.partOfPK), allEntities)
;

/*Diese Extension erzeugt rekursiv ForeignKey Attribute*/
Void addForeignKeysForContained(entitymodel::Entity entity,
        List[entitymodel::Attribute] pks, List[entitymodel::Entity] allEntities) :
  entity != null
```

```
  ? (createFKs(pks, entity)
    -> allEntities.select(e|entity.attribute.select(e|e.composition
                && e.metaType != entitymodel::InheritedAttribute)
                .type.exists(t|t == e.name))
                .addForeignKeysForContained(entity.attribute
                .select(e|e.partOfPK), allEntities)
    -> allEntities.select(e|e.eAnnotations
    .typeSelect(entitymodel::DiscriminatorValue).first().superClass == entity)
 .addForeignKeysForContained(entity.attribute
.typeSelect(entitymodel::ForeignKey),allEntities))
  : null
;
```

Quelltext 9.3: XPand Template für die Generierung von Zugriffsmethoden für Fremdschlüssel

```
«DEFINE gettersAndSetters FOR entitymodel::ForeignKey»
  «IF this.entity.getContainmentAttribute().type == this.entity.name»
  «IF !this.entity.getContainmentAttribute().isBidirectional()»
    public «this.type» get«this.name.toFirstUpper()»() {
      return this.«this.name»;
    }

    public void set«this.name.toFirstUpper()»(«this.type» «this.name») {
      this.«this.name» = «this.name»;
    }
  «ELSE»
    public «this.type» get«this.name.toFirstUpper()»() {
      if(this.get«this.entity.getContainmentAttribute()
       .inverseReferenceName.toFirstUpper()»() == null) return null;
      return this.get«this.entity.getContainmentAttribute()
       .inverseReferenceName.toFirstUpper()»().get«this.name.toFirstUpper()»();
    }
  «ENDIF»«ENDIF»
«ENDDEFINE»
```

Quelltext 9.4: XPand Template für die Generierung von assign-Methoden

```
«DEFINE MethodAssign FOR ejbmodel::Controller::AssociationMethod»
  «IF this.commands.typeSelect(ejbmodel::Controller::InvokeMethod).method
    .typeSelect(ejbmodel::ApplicationFacade::FacadeMethod)
    .exists(e|e.name.startsWith("assign"))»
    //Delegate assign
    «EXPAND assign FOR this»
  «ELSE»
    // Get the entities out of the database
    «EXPAND fetchSingleForMethod FOR this»

    «IF !this.attribute.hasX2ManyAnnotation()»
        // Set the attribute if not already another one exists
        if(«this.entity.getVariableName()»
                            .get«this.attribute.name.toFirstUpper()»()!=null)
                throw new AssignException("«this.attribute.name»");
```

```
16          «this.entity.getVariableName()».set«this.attribute.name.toFirstUpper()»
17                          («EXPAND variableNameForAttribute»);
18          «ELSE»
19          // Add the attribute
20          «this.entity.getVariableName()».add«this.attribute.type.toFirstUpper()»
21                          To«this.attribute.name.toFirstUpper()»(«EXPAND variableNameForAttribute»);
22      «ENDIF»
23      «IF this.attribute.isBidirectional() && this.attribute.hasOne2XAnnotation()»
24          // Set the inverse attribute if not already another one exists
25          if(«EXPAND variableNameForAttribute».get«this.attribute
26                          .inverseReferenceName.toFirstUpper()»()!=null)
27                  throw new AssignException("«this.attribute.name»");
28          «EXPAND variableNameForAttribute».set«this.attribute.inverseReferenceName.toFirstUpper()»
29                          («this.entity.getVariableName()»);
30          «ELSEIF this.attribute.isBidirectional() && !this.attribute.hasOne2XAnnotation()»
31                  // Add the inverse attribute
32                  «EXPAND variableNameForAttribute».add«this.entity.name.toFirstUpper()»
33                          To«this.attribute.inverseReferenceName.toFirstUpper()»
34                          («this.entity.getVariableName()»);
35          «ELSEIF this.attribute.inverseReferenceName == "" && this.attribute.composition»
36                  // Set ForeignKey Attribute
37                  «FOREACH this.entity.attribute.select(e|e.partOfPK) AS FK»
38                    «EXPAND variableNameForAttribute».set«FK.name.toFirstUpper()»(«FK.name»);
39                  «ENDFOREACH»
40      «ENDIF»
41
42          // Persist the entit(y/ies) back into the database
43          «EXPAND update FOR this»
44      «ENDIF»
45  «ENDDEFINE»
```

Quelltext 9.5: Teil des XPand Templates für die Generierung von create-Methoden für Komponenten

```
1   «DEFINE controllerCreate FOR ejbmodel::Controller::ControllerMethod»
2   ...
3       «IF this.entity.isContained()»
4   // -------------------------------------------------------
5   // |     Begin Assigning Containment                     |
6   // -------------------------------------------------------
7           // Get the entity out of the database
8           «EXPAND fetchSingleForMethod FOR this»
9           ...
10      «ENDIF»
11  ...
12  «ENDDEFINE»
13
14
15  «DEFINE fetchSingleForMethod FOR ejbmodel::Controller::ControllerMethod»
16    «FOREACH this.commands.typeSelect(ejbmodel::Controller::InvokeMethod)
17           .select(e|e.method.name.startsWith("get")
18            && !e.method.returnType.asList) AS invokeFetchMethod»
19        «LET invokeFetchMethod.method AS fetchMethod»
```

```
20          «fetchMethod.returnType.getName()»
21          «IF invokeFetchMethod.parameterMapping.from
22            .exists(e|e.name.startsWith("target"))»
23            target«fetchMethod.returnType.getName().toFirstUpper()»
24          «ELSE»«fetchMethod.returnType.getName().toFirstLower()»«ENDIF» =
25            «EXPAND controllerVariable(this) FOR fetchMethod.invokedBy
26             .select(e| this.commands.contains(e)).first()»
27             .«EXPAND invokeMethod FOR invokeFetchMethod»;
28        «ENDLET»
29    «ENDFOREACH»
30  «ENDDEFINE»
```

Literaturverzeichnis

[And] *AndroMDA Webseite.* http://www.andromda.org/. Last accessed on 2012-03-10.

[BG05] Bacvanski, Vladimir und Graff, Petter: *Mastering Eclipse Modeling Framework.* EclipseCon., 2005.

[Con12] Conrad, Steffen: *Ein Ansatz zum modellgetriebenen Test von EJB-basierten Informationssystemen.* Diplomarbeit, RWTH Aachen, 2012.

[Ecl] *Eclipse Wiki MDT-UML2.* http://wiki.eclipse.org/MDT-UML2. Last accessed on 2011-12-10.

[EFH+10] Efftinge, Sven, Friese, Peter, Hase, Arno, Hübner, Dennis, Kadura, Clemens, Köhnlein, Jan, Moroff, Dieter, Thoms, Karsten, Völter, Markus und Schönbach, Patrick: *Xpand Documentation.* Meta, 2010.

[Eve12] Evers, Frederic: *Konzeptionelle Erweiterung von Projektdashboards für unerfahrene Anwender.* Masterarbeit, RWTH Aachen, 2012.

[Gla] *Glassfish Webseite.* http://glassfish.java.net/. Last accessed on 2012-02-23.

[IBM] *IBM Websphere Webseite.* www.ibm.com/software/websphere/. Last accessed on 2012-02-23.

[IHHK07] Ihns, O., D. Harbeck, S. Heldt und H. Koschek: *EJB 3 professionell.* dpunkt.Verlag GmbH, 2007.

[JBo] *JBoss Webseite.* http://www.jboss.org/jbossas. Last accessed on 2012-02-23.

[Kyr12] Kyryliuk, Bogdan: *Development of an EJB-Based Business Goal Management Environment.* Masterarbeit, RWTH Aachen, 2012.

[LCM06] Lange, C F J, Chaudron, M R V und Muskens, J: *In practice: UML software architecture and design description.* IEEE Software, 23(2), 2006.

[LL07] Ludewig, Jochen und Lichter, Horst: *Software Engineering.* dpunkt.Verlag GmbH, 2007.

[Löw11] Löwenthal, Tobias: *Generierung von web-basierten Prototypen für Geschäftsanwendungen.* Diplomarbeit, RWTH Aachen, 2011.

[ope] *openArchitectureware.* `http://www.openarchitectureware.org/`. Last accessed on 2012-02-24.

[PL11] Pricope, S. und Lichter, H.: *A Model Based Integration Approach for Reference Models.* Second Proceedings of 12th International Conference on Product Focused Software Development and Process Improvement, 2011.

[PTN+07] Pietrek, Georg, Trompeter, Jens, Niehues, Benedikt, Kamann, Thorsten, Holzer, Boris, Kloss, Michael, Thoms, Karsten, Beltran, Juan Carlos Flores und Mork, Steffen: *Modellgetriebene Softwareentwicklung. MDA und MDSD in der Praxis.* entwickler.press, 2007.

[Rat] *Rational Software Architect Wiki.* `http://www.ibm.com/developerworks/wikis/display/rsa/Home`. Last accessed on 2012-02-23.

[RJB99] Rumbaugh, J, Jacobson, I und Booch, G: *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[Rum04] Rumpe, Bernhard: *Agile Modellierung mit UML.* Springer Verlag, 2004.

[Sch12] Schlimnat, Michael: *Konzeption einer Internetplattform zur Dokumentation von Metriken.* Bachelorarbeit, RWTH Aachen, 2012.

[Scu] *Sculptor Webseite.* `http://fornax.itemis.de/confluence/display/fornax/Sculptor+(CSC)`. Last accessed on 2012-03-13.

[TLG] *TLGen Webseite.* `http://www.tlgen.com/`. Last accessed on 2012-03-10.

[Via11] Vianden, Matthias: *Introduction EJB-GenCodename: Gargoyle.* Präsentation, RWTH Aachen, 2011.

[Wel] *Welcome to Eclipse.* `http://help.eclipse.org/indigo/index.jsp`. Last accessed on 2012-02-23.