Projektergebnis

QBench-Systemmetamodell

Identifikation

Projektidentifikation:	QBench
Projektergebnis:	Systemmetamodell
Vertraulichkeitsstufe:	Öffentlich
Arbeitspaket(e):	AP 4
Geplante Auslieferung:	31. Dezember 2005
Auslieferung:	16. Dezember 2005
Letzte Änderung:	20. Dezember 2005
Autor(en):	M. Trifu, P. Szulman, V. Kuttruff
Koordinator:	Forschungszentrum Informatik

Zusammenfassung

Dieses Dokument gibt einen Überblick über das Metamodell, welches in QBench zum Einsatz kommt. Der Fokus liegt dabei auf den quellcodenahen Schichten, welche die Analyse und Transformation bestehender Softwaresysteme unterstützen.

Schlüsselwörter

Systemmetamodell, Analysemodell, Transformationsmodell, Programmanalyse, Reverse Engineering, Qualitätsanalyse, Softwarequalität, Transformation

Vertraulichkeitsstufe

Klassifikation	Vertraulichkeit
Öffentlich	Der Öffentlichkeit zugängliche Informationen

Haftungsausschluss

Die Autoren übernehmen keinerlei Gewähr für die Aktualität, Korrektheit, Vollständigkeit oder Qualität der in diesem Dokument bereitgestellten Informationen. Haftungsansprüche gegen die Autoren, welche sich auf Schäden materieller oder ideeller Art beziehen, die durch die Nutzung oder Nichtnutzung der dargebotenen Informationen bzw. durch die Nutzung fehlerhafter und unvollständiger Informationen verursacht wurden, sind grundsätzlich ausgeschlossen.

Inhaltsverzeichnis

Kapitel 1

Einführung

Ziel des Verbundprojektes QBench ist Entwicklung und Einsatz eines ganzheitlichen Ansatzes zur konstruktions- und evolutionsbegleitenden Sicherung der inneren Qualität von objektorientierter Software, um den Aufwand der Softwareentwicklung und -evolution deutlich senken zu können. Um diesen Prozess zu erleichtern, kommen dabei Werkzeuge vor allem in den folgenden Bereichen zum Einsatz:

- Softwareanalysen
- Softwaretransformation

Diese Werkzeuge haben gemeinsam, dass sie mit einer geeigneten Repräsentation des Quelltextes umzugehen haben. Der Quelltext selbst ist nicht ausreichend abstrakt, es sind höherwertige Beschreibungen, Modelle, notwendig.

Um alle Zielsetzungen von QBench verwirklichen zu können, müssen zahlreiche Werkzeuge zusammenarbeiten. Das setzt eine einheitliche Modellierung des Quelltextes voraus. Die Syntax und Semantik der gültigen Modelle wird durch ein sogenanntes Systemmetamodell definiert.

Dieses Dokument stellt unser Systemmetamodell vor, welches eine geeignete Repräsentation eines Softwaresystems erlaubt. Diese Repräsentation findet innerhalb einer Reihe von den im Rahmen von QBench zu entwickelnden und zu erweiternden Werkzeugen - wie z.B. InjectJ (automatisierte Softwaretransformationen), Goose (automatisierte Softwareanalysen) - Verwendung. Dieses Metamodell baut auf dem METAMOD-Metamodell, das in dem CompoBench-Ergebnis [Kut03] beschrieben wurde, auf.

Das Dokument gliedert sich in vier Teile. Im folgenden Kapitel werden zunächst einige Grundbegriffe und Technologien beschrieben, die im Rahmen des Systemmetamodells von Bedeutung sind. In Kapitel 3 wird das Systemmetamodell vorgestellt und spezifiziert. Daran anschließend werden in Kapitel 5 Möglichkeiten zur persistenten Speicherung von Modellen aufgezeigt sowie der in QBench verfolgte Ansatz beschrieben. In Kapitel 6 wird schließlich noch eine kurze Zusammenfassung des Dokuments gegeben.

Kapitel 2

Begriffe und Grundlagen

In diesem Kapitel werden die im Folgenden benutzten Begriffe und Technologien kurz vorgestellt. Sie bilden die Grundlage für das im nächsten Kapitel vorgestellte Metamodell. Die meisten Begriffe und Technologien wurden bereits im QBench-Projektergebnis Stand der Technik [AMS+04] ausführlich vorgestellt. Daher werden sie an dieser Stelle nur noch einmal kurz aufgegriffen und nicht in aller Ausführlichkeit beschrieben.

2.1 Begriffe

Modell

Ein zentraler Begriff in diesem Dokument ist das *Modell*. Die genaue Definition eines Modells ist in [AMS+04] zu finden. Modelle beschreiben die Elemente der Wirklichkeit und deren Beziehungen. Die Elemente werden auch als *Objekte* bezeichnet. Ein konkretes Gebilde aus Elementen und deren Beziehungen wird vom Standpunkt eines Modells aus als Modell*instanz* bezeichnet. Im Rahmen dieses Dokuments ist insbesondere der Zusammenhang zwischen Operationen f_W in der Wirklichkeit (bzw. dem Modell, welches bzgl. einer weiteren Modellbildung die Wirklichkeit darstellt) und der entsprechenden Operation f_M auf dem Modell wichtig. Sei i die bei der Modellbildung benutzte Abstraktion und $\circ$ der Operator *nach*, so muss für die benutzten Modelle $f_M \circ i = i \circ f_W$ gelten (vgl. Abbildung 2.1). Für unseren Anwendungszweck spezifisch ausgedrückt bedeutet dies, dass eine Analyse bzw. Transformation auf einem Modell zum gleichen Ergebnis führt wie das Durchführen der Analyse bzw. der Transformation in der Wirklichkeit mit anschließender Modellbildung.

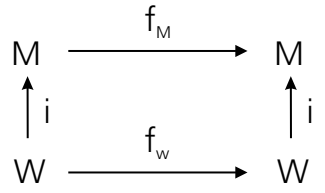

Abbildung 2.1: Beziehung zwischen Modell und Wirklichkeit

Metamodell, Meta-Metamodell

Ein Metamodell beschreibt, wie eine Klasse von Modellen aussieht. Ein Metamodell stellt somit einen Bauplan für konkrete Modelle dar. Die Instanzen eines Metamodells sind somit Mo-

delle.

Ein Meta-Metamodell wiederum ist ein Metamodell für Metamodelle, die Instanzen eines Meta-Metamodells sind somit Metamodelle. Im Prinzip kann eine beliebig große Modellschichthierarchie aufgestellt werden, wobei jede Schicht ihre gültigen Ausprägungen auf der darunterliegenden Schicht beschreibt (siehe linke Seite der Abbildung 2.2). Da ein Meta-Metamodell selbst nur ein Metamodell ist, lässt sich im Allgemeinen ein Meta-Metamodell mit sich selbst beschreiben. Die in QBench verwendete Modellhierarchie besteht aus 5 Schichten. Die unterste Schicht bildet der Quelltext. Bei der Analyse wird ein sogenanntes Strukturmodell aufgebaut, das den Aufbau vom Quelltext modelliert. Die Syntax und Semantik der Strukturmodelle wird durch ein Metamodell (METAMOD) beschrieben. Da METAMOD mit der Hilfe der UML modelliert wurde, ist es daher eine Instanz des UML-Metamodells. Das UML-Metamodell ist als Instanz des "Metaobject Facility"-Modells (kurz: MOF-Modells) definiert. In diesem Dokument beschreiben wir nur METAMOD, die Beschreibung des UML-Metamodells und des MOF-Modells findet man in [Obj01] bzw. [Obj02].

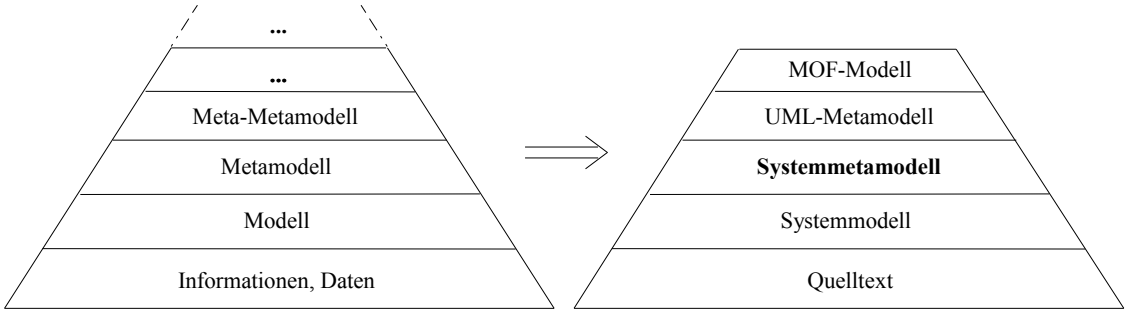

Abbildung 2.2: Modellhierarchie

Fakten, Faktenextraktion, Designdatenbank

Im Bereich der Qualitätsanalysen stellen *Fakten* die Initial über die Wirklichkeit zur Verfügung stehenden Informationen dar. Fakten ändern sich im Betrachtungszeitraum im Allgemeinen nicht. Auf Basis dieser Fakten lassen sich entsprechende Modelle des zu analysierenden Softwaresystems instanziieren.

Die Gewinnung dieser Fakten aus der Wirklichkeit (in unserem Fall aus dem Quelltext oder einer anderen Repräsentation eines Softwaresystems) geschieht mit Hilfe des sogenannten *Faktenextraktors*. Er trägt, unter Umständen begleitet von einer Vorverarbeitung, die zur Berechnung der Fakten benötigten Daten zusammen und legt diese anschließend in einer *Faktendatenbank* ab. Im Falle von Softwaresystemen spricht man in diesem Zusammenhang auch oftmals von einer *Designdatenbank*, falls die Fakten das Design des Systems repräsentieren.

Kapitel 3

Systemmetamodell

3.1 Architektur des QBench-Systemmodells

Das im Rahmen von QBench benutzte Systemmodell ist bei genauerer Betrachtung eine Menge verschiedener Modelle mit unterschiedlichen Abstraktionsniveaus.

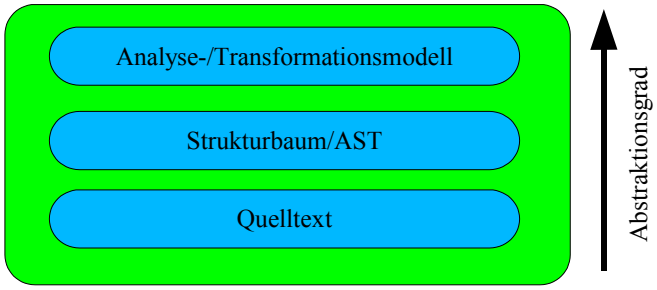

Abbildung 3.1: Architektur des QBench-Systemmodells

Ein Überblick über alle Schichten des QBench-Systemmodells ist in Abbildung 3.1 zu finden. Eine Schicht baut dabei stets auf der darunterliegenden Schicht auf, wobei bei jeder neuen Schicht der Abstraktionsgrad erhöht wird. Das QBench-Systemmodell ist somit eine Menge einzelner Modelle, die durch entsprechende Abbildungsvorschriften (*vertikale Transformationen*) miteinander verbunden sind. Je nach Anwendungsfall wird in der Praxis die geeignetste Schicht verwendet, wobei im Allgemeinen möglichst die abstrakteste Schicht verwendet werden sollte.

Insgesamt unterscheiden wir drei verschiedene Schichten:

- Der *Quelltext* stellt das feingranularste und umfassendste Modell eines Softwaresystems zur Verfügung. Der Quelltext-Schicht wird die Schicht Nr. 0 zugewiesen.
- Aus dem Quelltext lässt sich der abstrakte *Strukturbaum* gewinnen. Er stellt im Wesentlichen eine für Werkzeuge geeignete Repräsentation des Quelltexts dar.
- Darauf aufbauend bietet die Schicht Nr. 2 ein Analyse- und Transformationsmodell, welches den Strukturbaum weiter abstrahiert.

In den folgenden Abschnitten werden die Schichten des QBench-Systemmodells (siehe Abbildung 3.1 bzw. 2.2) genauer betrachtet, wobei dem Analyse- und Transformationsmodell auf Schicht Nr. 2 die meiste Aufmerksamkeit gewidmet wird, da dieses im Rahmen von QBench weiterentwickelt und in der vorliegenden Form noch nicht dokumentiert wurde.

Die drei Schichten werden nach folgender Vorgehensweise beschrieben: Nach einer Motivation, welche die Notwendigkeit der einzelnen Schichten im QBench-Systemmodell begründet, werden die Anforderungen angegeben, die von einem konkreten Modell dieser Abstraktionsschicht erfüllt werden müssen. Schließlich wird das Metamodell des Modells spezifiziert, welches die angegebenen Anforderungen erfüllt und im Rahmen von QBench zum Einsatz kommt.

3.2 Quelltext

3.2.1 Motivation

Der Quelltext ist für zahlreiche gewachsene Systeme die einzig verfügbare und konsistente Beschreibung des Systems. Neben der Struktur definiert er auch das Verhalten der Software. Beim heutigen Stand der Technik ist er allerdings auch die einzige universelle und daher am meisten benutzte Möglichkeit zur Spezifikation dieses Verhaltens.

In nahezu allen Fällen ist der Quelltext die einzige Möglichkeit, mit Hilfe eines Übersetzers ausführbaren Code zu erhalten. Da letztlich alle Spezifikationen im Allgemeinen in einem letzten Schritt auf Quelltext abgebildet bzw. übersetzt werden, nimmt dieser eine zentrale Rolle bei der Beschreibung eines Softwaresystems ein.

3.2.2 Anforderungen

Je nach Anwendungsfall muss der Quelltext eines Systems verschiedenen Anforderungen genügen. Damit der Quelltext erfolgreich übersetzt werden kann, muss er bezüglich der verwendeten Programmiersprache syntaktisch und semantisch korrekt sein.

Teilweise noch restriktivere Anforderungen müssen an den Quelltext gestellt werden, falls er den Ausgangspunkt für werkzeuggestützte Transformationen bzw. Restrukturierungen bildet. In diesem Fall muss der Quelltext ebenfalls syntaktisch und semantisch korrekt sein. Außerdem muss es sich um eine geschlossene Welt handeln, das heißt alle benutzten Quellen bzw. Bibliotheken müssen zur Verfügung stehen.

Stellt der Quelltext den Ausgangspunkt für (Qualitäts-)Analysen dar, so können die Anforderungen an den Quelltext unter Umständen gelockert werden. Je nach geforderter Genauigkeit beziehungsweise gefordertem Detaillierungsgrad des Analyseergebnisses ist auch unvollständiger beziehungsweise mit kleinen syntaktischen und semantischen Fehlern behafteter Quelltext akzeptierbar. In solch einem Fall muss das verwendete Analysewerkzeug nach einem Fehler wieder aufsetzen können.

3.2.3 Spezifikation des Metamodells

Das Metamodell von Quelltext ist auf den ersten Blick sehr einfach: es handelt sich um eine Sequenz einzelner Zeichen in einer durch das Betriebsystem oder die Sprachbeschreibung vorgegebenen Codierung (z.B. ASCII, UNICODE). Dieses einfache Metamodell spiegelt sich auch in den Werkzeugen wider, welche die direkte Manipulation von Quelltext erlauben. Es handelt sich bei diesen Werkzeugen ausschließlich um Manipulatoren für allgemeine Textdateien, wie zum Beispiel `awk` und `sed`.

Auch wenn dieses Metamodell des Quelltexts sehr einfach ist, so ist es für die Beschreibung von Quelltext einer Programmiersprache nicht ausreichend. Mathematisch ausgedrückt ist es notwendig, aber nicht hinreichend. Dieses einfache Metamodell würde jede beliebige (Text-) Datei als Quelltext zulassen. Dies ist im Hinblick auf den Zweck des Quelltexts, der Beschreibung eines Softwaresystems, nicht sinnvoll. Der Quelltext muss mehr oder weniger der Spezifikation der benutzten Programmiersprache, das heißt dem damit festgelegten Sprachmetamodell, folgen. Um den Quelltext zu übersetzen, muss er dieser Sprachspezifikation bis ins Detail folgen.

Wird er für einfachere Analysezwecke benutzt, darf er auch kleine Abweichungen bezüglich dieser Sprachspezifikation haben. Der Grad der erlaubten Abweichung lässt sich allerdings nicht pauschal festlegen.

Die Spezifikation der Metamodelle der einzelnen in QBench berücksichtigten Programmiersprachen Java, Delphi, C# und C/C++ ist in den einzelnen Sprachbeschreibungen zu finden. Für Java ist die ausführliche Sprachspezifikation in [GJSB00] zu finden. Die Sprache C++ wird in [Str97] beschrieben, die Spezifikation von Delphi bzw. Object Pascal in [Bor03] und in [Lis00]. Die Programmiersprache C# ist durch die ECMA standardisiert worden. Das Resultat mit der Kennung ECMA-334 ist in [Int02] zu finden.

3.3 Strukturbaum

3.3.1 Motivation

Der Quelltext ist für zahlreiche Anwendungsfälle der maschinellen Analyse und Manipulation eines Softwaresystems keine geeignete Darstellung. Für einen Rechner ist der Quelltext zunächst nur eine Zeichenfolge ohne Struktur. Erst eine syntaktische Analyse ermittelt aus der Zeichenkette die einzelnen Strukturen der benutzten Programmiersprache. Da die Strukturen heutiger Programmiersprachen hierarchisch zusammengesetzt sind, lassen sich diese in einer baumartigen Datenstruktur, dem sogenannten Strukturbaum, darstellen. Neben dem Erkennen der Strukturen müssen während einer anschließenden semantischen Analyse die Verbindungen zwischen den einzelnen Strukturen hergestellt werden, zum Beispiel die Verbindung zwischen der Benutzung einer Variable und ihrer Deklaration. Ein Programmierer bringt beim Lesen des Quelltexts implizit die Leistung, Struktur und Semantik der Zeichenfolge zu erkennen und zu interpretieren. Analysen und Transformationen werden von einem Programmierer konzeptionell also mindestens auf diesem Abstraktionsniveau durchgeführt, insbesondere also nicht auf dem Niveau von Zeichenketten.

3.3.2 Anforderungen

Das wichtigste Merkmal des Strukturbaums muss sein, wie der Name bereits sagt, dass die einzelnen Strukturen eines Programms explizit in einer Baumstruktur repräsentiert werden. Neben den Strukturen muss der Strukturbaum auch noch zusätzliche semantische Informationen bereitstellen. So müssen die Ergebnisse der Namens- und Typanalyse in Form entsprechender Referenzen zur Verfügung stehen. Neben den durch die Namensanalyse bereitgestellten Benutzungs-Definitions-Ketten sind auch die Definitions-Benutzungsketten (das heißt Querverweis-Informationen (engl. Crossreferencing)) wünschenswert.

Wird der Strukturbaum nur für Analysezwecke verwendet, so ist ein lesender Zugriff darauf ausreichend. Falls der Strukturbaum die Grundlage für darauf aufbauende Transformationen ist, so muss dieser natürlich manipulierbar, das heißt schreibbar sein. Sowohl im Analysefall, als auch im Transformationsfall muss eine entsprechende Schnittstelle (API) für Modellanfragen beziehungsweise Modelltransformationen zur Verfügung gestellt werden.

Im Fall von Modelltransformationen ist es notwendig, aus dem transformierten Strukturbaum wieder Quelltext generieren zu können. Hierbei müssen Kommentare und Formatierungsinformationen erhalten bleiben, das heißt der generierte Quelltext sollte abzüglich der durchgeführten Änderungen exakt dem ursprünglichen Quelltext entsprechen.

3.3.3 Spezifikation des Metamodells

Das Metamodell des Strukturbaums ist implizit durch die Sprachspezifikation der zugrundeliegenden Programmiersprache gegeben. Der Aufbau eines Strukturbaums erfordert dabei ei-

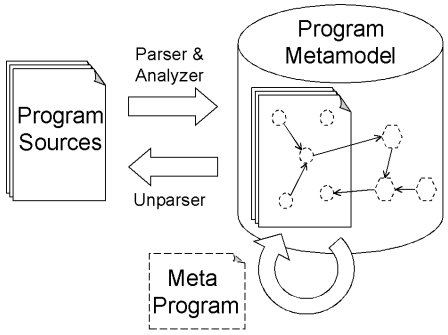

Abbildung 3.2: Vorgehen bei der Programmrepräsentation und -verarbeitung mit Recoder

ne syntaktische Zerlegung des Quelltexts. Die gefundenen Strukturen werden durch Knoten im Strukturbaum repräsentiert.

Um einen maximalen Nutzen aus dem Strukturbaum zu ziehen, ist es vorteilhaft, wenn die Knoten des Strukturbaums auf irgendeine Art typisiert sind. Im einfachsten Fall benutzt man für die Typisierung der Knoten das Typsystem der Sprache, in welcher der Strukturbaum implementiert ist. Die Knotentypen lassen sich aus der Grammatikspezifikation wie folgt ableiten: Die linken Seiten einer Produktion der Grammatik beschreiben einen neuen Knotentyp des Strukturbaums, die jeweils rechten Seiten die Zusammensetzung der Kinder des neuen Knotentyps.

Da es den Umfang sowohl dieses Dokuments als auch den des Projektes QBench sprengen würde, eine vollständige Spezifikation und Implementierung für alle bei den QBench-Partnern im Einsatz befindlichen Programmiersprachen (Java, C/C++, C#, Delphi) zu erstellen, konzentrieren wir uns im Folgenden auf das Java-Metamodell. Dies geschieht nicht zuletzt deswegen, weil es neben C# am besten durch eine allgemein zugängliche, konsistente Sprachspezifikation beschrieben ist (siehe [GJSB00]). Ein Reverse Engineering der Sprachspezifikation ist somit nicht notwendig.

Im folgenden Abschnitt wird eine mögliche Implementierung des Java-Metamodells vorgestellt, welche die zu untersuchenden und zu transformierenden Programme in Form eines Strukturbaums darstellt.

3.3.3.1 Recoder

Die Bibliothek Recoder [LN02] ist ein Metaprogrammiersystem für Java. Es bietet die Möglichkeit, Java-Dateien (Quellcode und Bytecode) einzulesen, eine Repräsentation in Form eines Strukturbaums aufzubauen (inklusive der Ergebnisse einer Namen- und Typanalyse), Analysen und Transformationen auf dieser Programmrepräsentation durchzuführen, und anschließend wieder Quelltext unter Beibehaltung der ursprünglichen Formatierung inklusive Kommentaren zu generieren. Abbildung 3.2 zeigt dieses Vorgehen noch einmal. Die Transformation des Strukturbaums ist dabei nur dann möglich, falls die entsprechenden Elemente im Quelltext vorliegen. Liegen sie nur im Bytecode vor, so ist nur eine Analyse bis auf Signaturebene möglich, aber keine Transformation.

Die Programmrepräsentation erfolgt wie bereits erwähnt mit Hilfe eines Strukturbaums der Java-Quellen. Der Strukturbaum erfasst dabei alle Elemente der Sprache Java, das heißt insbesondere auch alle Arten von Ausdrücken. Der Strukturbaum folgt dabei dem in [GJSB00] vorgestellten Metamodell. Da die vollständige Spezifikation des Recoder-Metamodells kein Ziel dieses Dokuments ist, sondern vielmehr die Spezifikation des im Rahmen von QBench entwickelten Analyse- und Transformationsmetamodells in Abschnitt 3.4, soll an dieser Stelle nur ein kurzer Überblick über Recoder gegeben werden. Für eine detaillierte Beschreibung sei auf [LN02] sowie auf die dort zu findende API-Dokumentation verwiesen.

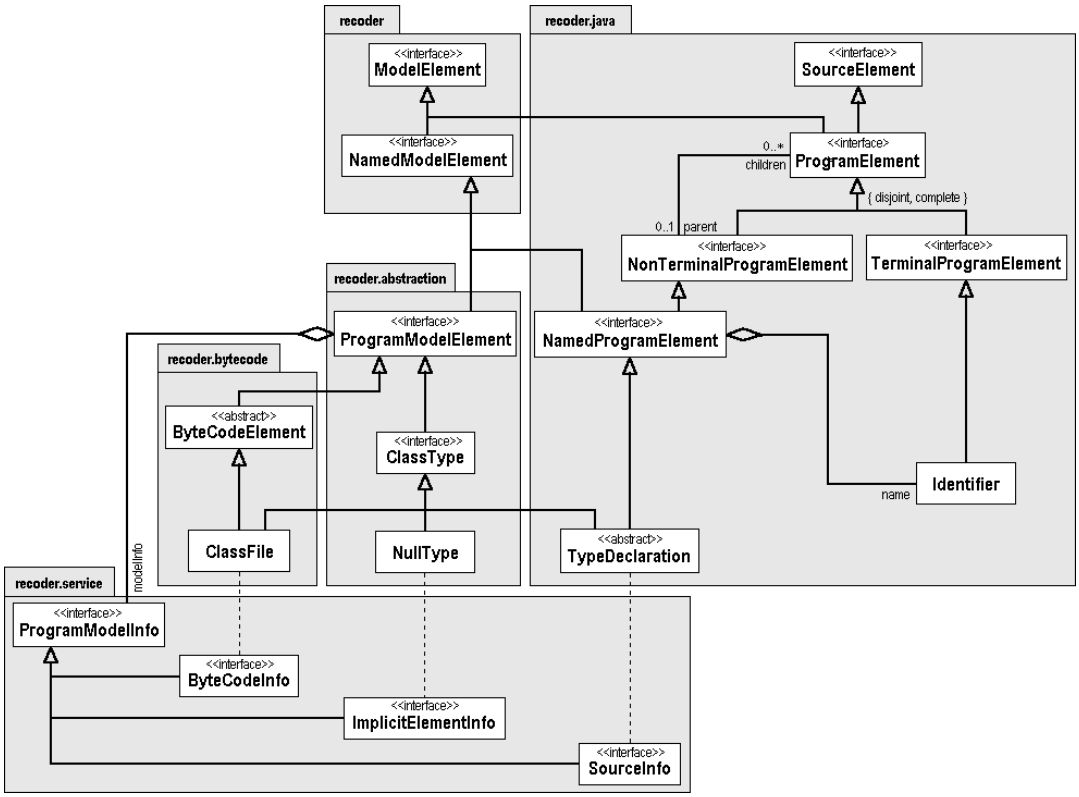

Abbildung 3.3: Strukturierungselemente des Recoder-Metamodells

Abbildung 3.3 zeigt eine Auswahl der zur Strukturierung und Klassifikation der Modellelemente benutzten Schnittstellen. Insbesondere ist in dieser Abbildung das in Abschnitt 3.3.3 skizzierte Verfahren zur Ableitung der Knotentypen des Strukturbaums aus der Sprachspezifikation (genauer: aus der Grammatikspezifikation) zu erkennen: Die linke Seite einer Grammatikproduktion (in der Abbildung: *ProgramElement*) beschreibt einen neuen Knotentyp, die rechte Seite definiert die Zusammensetzung der Kinder (in der Abbildung durch die Verbindung zwischen *NonTerminalProgramElement* und *ProgramElement* zu sehen).

Um einen Eindruck von den einzelnen Knotentypen des Strukturbaums von Recoder zu vermitteln, sind in Abbildung 3.4 beispielhaft die grobgranularen Sprachelemente dargestellt. Die dargestellten Elemente dienen zur Beschreibung der Schnittstelle einer Klasse.

Der Einstieg in den Strukturbaum und eine schnelle Navigation in diesem erfolgt in Recoder mit Hilfe sogenannter *Services*. Die generelle Struktur dieser Services ist in Abbildung 3.5 dargestellt. So bietet Recoder die Möglichkeit, ausgehend von einem Namen entsprechende Knoten des Strukturbaums zu suchen (*NameInfo*), oder allgemeine Informationen über bestimmte Konstrukte zu berechnen (*ProgramModelInfo*), wie zum Beispiel Typinformationen für Ausdrücke oder die Menge aller Ober- und Unterklassen einer Klasse. Zusätzlich bietet Recoder auch Querverweisinformationen an, das heißt ausgehend von einer Deklaration lassen sich alle Benutzungsstellen ausgeben (*CrossReferencer*). Mit Hilfe des Service *ChangeHistory* protokolliert Recoder alle Änderungen am Strukturbaum mit und kann sie somit gegebenenfalls wieder rückgängig machen, falls dies notwendig sein sollte.

Wie bereits am Anfang dieses Abschnitts erwähnt, existiert eine vollständige Implementierung des Recoder-Systems bisher nur für Java 1.4.x. Es existieren allerdings Bemühungen, das Recoder-System an die Sprache C# anzupassen (siehe [Tea02]). Aufgrund der Ähnlichkeiten von

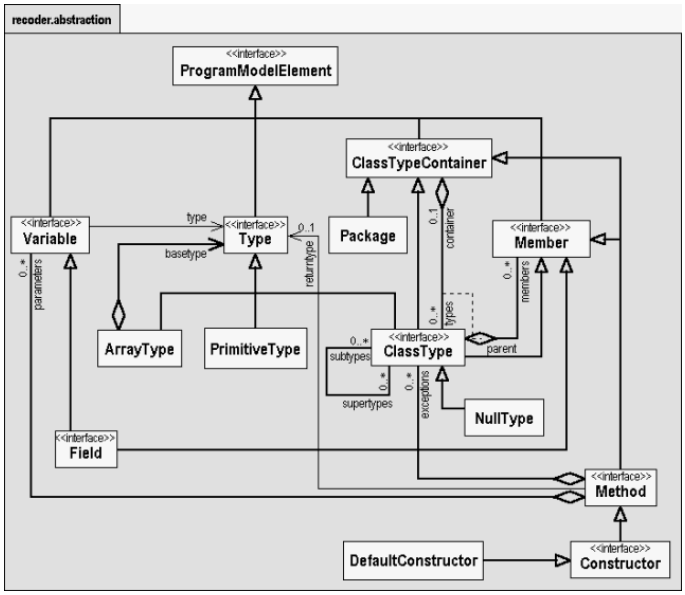

Abbildung 3.4: Modellierung der grobgranularen Sprachelemente im Recoder-Metamodell

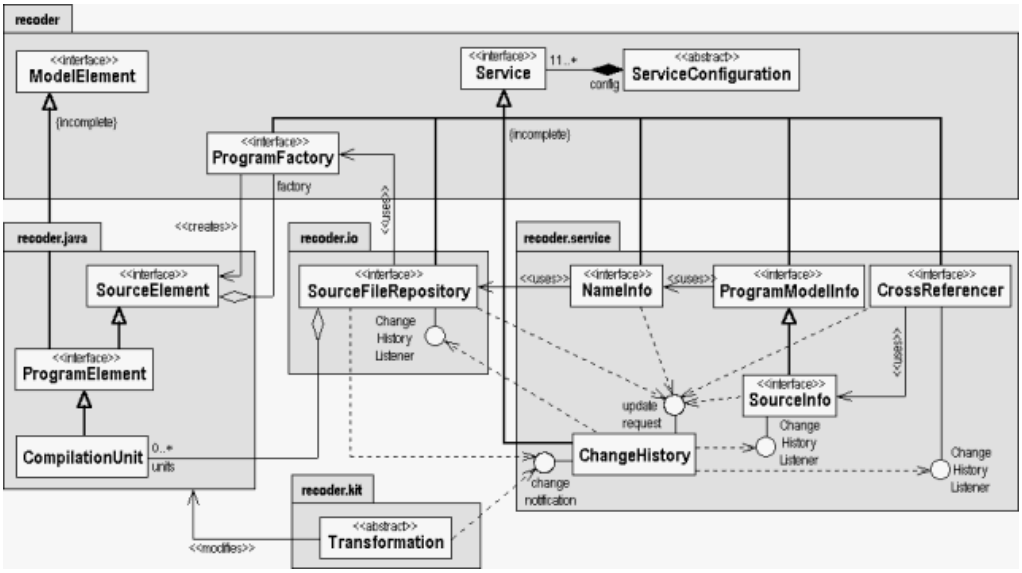

Abbildung 3.5: Service-Infrastruktur von Recoder

Java und C# kann das bisherige Recoder-Metamodell als Startpunkt dienen. Größere Änderungen erfordern allerdings die in C# vorhandenen Präprozessoranweisungen, da hiermit letzten Endes verschiedene Konfigurationen beschrieben werden können (siehe [KOP+03]). Dies erfordert eine Erweiterung des Metamodells hinsichtlich der gleichzeitigen Verwaltung mehrerer Konfigurationen.

3.4 Analyse- und Transformationsmodell für objektorientierte Sprachen

3.4.1 Motivation

Zahlreiche Analysen des Reverse Engineerings und der Qualitätssicherung, aber auch zahlreiche Transformationen, sind nicht an derart feingranulare Modelle gebunden, wie dies Modelle von Programmiersprachen (Strukturbäume) sind. Gröbergranulare Modelle sind für diese Aktivitäten nicht nur ausreichend, sondern auch dringend notwendig, da insbesondere für große Systeme die Informationsflut des abstrakten Strukturbaums (AST) für viele Aktivitäten erdrückend ist. Die Vielzahl der durch ein Sprachmetamodell bereitgestellten Informationen wirkt sich sowohl negativ auf den Speicherbedarf der Modelle (Metamodellinstanzen), als auch auf den Aufwand für die konzeptionelle Planung von Analysen und Transformationen durch einen Softwareingenieur aus. Entsprechend abstraktere Modelle bieten hier Vorteile, da sie den Teil der Komplexität des Systems verbergen, der aus der Anweisungs- und Ausdrucksebene resultiert. Das prominenteste Metamodell, welches nur Teile eines Sprachmetamodells zugunsten besserer Analysierbarkeit und Handhabbarkeit durch den Softwareingenieur betrachtet, ist die Unified Modeling Language (UML).

Neben besserer Handhabbarkeit abstrahiert ein entsprechendes Modell bis zu einem gewissen Grad auch von speziellen Sprachkonstrukten verschiedener konkreter Programmiersprachen. So ist es möglich, mehrere Sprachen einer Sprachfamilie durch dieses Modell zu erfassen. Die UML ist hierfür wiederum das verbreitetste Beispiel. Im Rahmen von QBench konzentrieren wir uns hierbei auf die bei den Projektpartnern eingesetzten Sprachen Java, C++, C# und Delphi, oder allgemeiner ausgedrückt auf objektorientierte, ausdrucksbasierte Sprachen. Diesen Sprachen ist gemein, dass sie ähnliche Entitäten auf der Designebene besitzen (z.B. Klassen, Methoden und Attribute, teilweise auch globale Funktionen und Variablen), aber auch ähnliche Konzepte für allgemeinere Sprachkonstrukte wie Zugriffe auf Designelemente oder Zuweisungen verfolgen. Daher ist es mit Hilfe eines entsprechenden Modells möglich, zahlreiche Analysen des Reverse Engineerings und der Qualitätssicherung, zum Teil aber auch Transformationen, sprachunabhängig bezüglich dieser Sprachen zu beschreiben und durchzuführen. Auch wenn Analysen oder Transformationen an bestimmten Punkten von sprachabhängigen Informationen oder sprachabhängigen Teiltransformationen abhängen, so bringt das Ausfaktorisieren des sprachunabhängigen Teils der Analyse oder Transformation Vorteile bei der Portierung auf eine andere Sprache, da dieser übernommen werden kann. Ausserdem kann das sprachübergreifende Metamodell als Grundlage für gröber granulare Abstraktionen dienen.

3.4.2 Anforderungen

Das im Folgenden vorgestellte Modell muss einigen im Vorfeld der Modellbildung definierten Anforderungen und Randbedingungen genügen. Das Modell sollte möglichst *sprachübergreifend* sein, um Softwaresysteme repräsentieren zu können, welche in objektorientierten, ausdrucksbasierten Sprachen geschrieben sind. Hierbei sollte sich das Modell nicht nur auf „moderne“ objektorientierte Sprachen wie Java und C# konzentrieren, sondern auch die aus prozeduralen Sprachen „gewachsenen“ Sprachen wie Delphi und C++ unterstützen.

Das Modell muss für die Analyse und Transformation von Softwaresystemen geeignet sein. Ein entsprechender *Abstraktionsgrad* gegenüber dem Quelltext beziehungsweise dem daraus abgeleiteten AST stellt dabei eine wichtige Eigenschaft des Modells dar. Das Modell sollte sich *persistent speichern* lassen, das heißt es kann zu einem späteren Zeitpunkt *ohne Zugriff auf den Quelltext* des beschriebenen Systems wieder aufgebaut werden. Die dazu notwendigen Daten sollten möglichst *redundanzarm* sein, um einerseits den Speicherbedarf einer Modell (Metamodellinstanz) gering zu halten (und somit die Analyse großer Systeme zu ermöglichen), und andererseits die Dateigröße für die persistente Speicherung und die dazu notwendige Verarbeitungszeit bei ge-

gebener Codierung zu minimieren. Nur sehr aufwändig berechnete Informationen sollten redundant gespeichert werden.

3.4.3 Spezifikation

Die folgenden Abschnitte spezifizieren den Teil des QBench-Systemmodells, welcher den in Abschnitt 3.4.2 angegebenen Anforderungen genügt. Zunächst wird dazu auf den generellen Aufbau des Modells eingegangen. Anschließend wird die Notation, in welcher das Metamodell spezifiziert wird, vorgestellt. Aufgrund des Umfangs des Modells wird dieses inkrementell mit Hilfe verschiedener Sichten realisiert. Das Gesamtmodell ergibt sich anschließend aus der Vereinigung dieser Sichten.

3.4.3.1 Aufbau des Modells

Das im Folgenden vorgestellte Modell mit seinen Erweiterungen besitzt im Wesentlichen zwei Stoßrichtungen: es soll zum einen *sprachunabhängig* sein, und zum anderen einen auf den Anwendungsfall angepassten *Detaillierungsgrad* besitzen.

Das Modell setzt sich aus einem *Analysekern* und einer Erweiterung dieses Kerns hin zu einem Transformationsmodell zusammen. Der Analysekern stellt im Vergleich zur Transformationserweiterung eher grobgranulare Modellelemente und -informationen bereit. Da für die weitere Analyse eines Systems der Quelltext des Systems nicht mehr benötigt wird, sondern nur noch die im Modell enthaltenen grobgranularen Informationen, muss das Analysemodell persistent gespeichert werden können. Dies erlaubt dann auch die Durchführung von Analysen unabhängig von der Verfügbarkeit des Quelltexts.

Die Transformationserweiterung setzt auf dem Analysekern insofern auf, als dass durch sie nur neue Modellelemente und -informationen beziehungsweise Operationen hinzukommen, das heißt der Detaillierungsgrad des Modells wird erhöht. Der Analysekern ist somit eine echte Teilmenge des Transformationsmodells. Da die Transformationserweiterung letztlich die Transformationen auf dem Quelltext des Systems nachziehen muss, lohnt sich die persistente Speicherung dieses Modells nicht im gleichen Maße wie beim Analysekern, da neben dem Modell auch der Quelltext zur Verfügung stehen muss, was im Wesentlichen einer redundanten Speicherung entspricht.

Sowohl der Analysekern als auch die Transformationserweiterung sind an ihrer Schnittstelle sprachunabhängig, das heißt die Semantik der einzelnen Modellelemente und der Modellinformationen ist für alle unterstützten Sprachen gleich. Da dies letztendlich immer auf die Syntax und die Semantik der konkret unterstützten Programmiersprache abgebildet werden muss, müssen sowohl für den Analysekern als auch für die Transformationserweiterung für jede unterstützte Programmiersprache entsprechende Abbildungen angegeben werden. Diese Abbildungen sind aber im QBench Ergebniss [ST05] vorgestellt. In Abbildung 3.6 wird der Aufbau des Gesamtmodells noch einmal veranschaulicht.

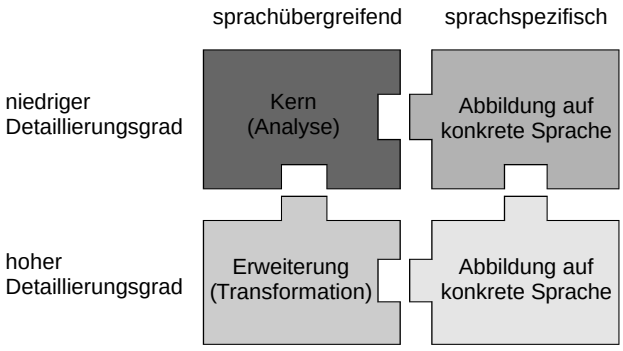

Abbildung 3.6: Aufbau des Modells

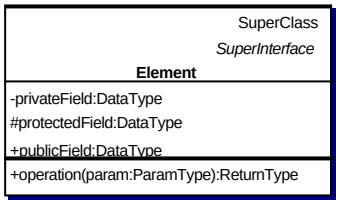

Abbildung 3.7: Benutzte UML Notation

3.4.3.2 Notation

Das im Folgenden vorgestellte Metamodell wird in diesem Dokument mit Hilfe der UML beschrieben. Dabei wird die Notation benutzt, wie sie das UML-Werkzeug *Together* der Firma Borland (ehemals der Firma TogetherSoft) bereitstellt. Die Abbildung 3.7 stellt ein Beispiel für die benutzte Notation dar. Im obersten Teil der Klasse steht der **fett** gedruckte Name der Klasse bzw. die Liste der erweiterten Typen. Die implementierten Schnittstellen sind *kursiv* gedruckt, die Oberklassen mit normalem Text. Falls es sich bei dem modellierten Element um eine Schnittstelle handelt, wird ihr Name *kursiv* geschrieben und durch den Stereotyp ≪interface≫ ergänzt. Der mittlere Teil listet die Attribute der Klasse (Bezeichner und Typ) auf, die Methoden sind im unteren Teil aufgeführt.. Die Sichtbarkeit eines Klassenmerkmals (Attribut oder Methode) wird in der benutzten Notation durch das Voranstellen eines der Zeichen '−', '#' oder '+' angegeben. Das Zeichen '−' bedeutet dabei private Sichtbarkeit (das heißt das Merkmal ist nur innerhalb der Klasse sichtbar), das Zeichen '#' spezifiziert geschützte Sichtbarkeit und das Zeichen '+' schließlich öffentliche Sichtbarkeit.

3.4.3.3 Analysemetamodell

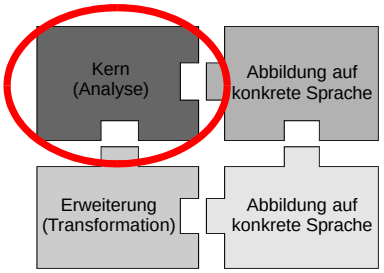

Abbildung 3.8: Struktursicht

Dieses Unterkapitel beinhaltet die strukturelle Beschreibung des Analyseteils des Metamodells. Abbildung 3.9 zeigt den oberen Teil der Klassenhierarchie der Modellelementen und die damit verbundenen Hilfsklassen.

Kern des Metamodells

An der Spitze dieser Hierarchie befindet sich die Schnittstelle *ModelElement*. Hauptzweck dieser Schnittstelle ist es, die einheitliche Behandlung aller Modellelemente zu ermöglichen und gleichzeitig die üblichen Dienstleistungen anzubieten, wie zum Beispiel ein Modellelement zu annotieren, einen Besucher entgegenzunehmen ([GHJV95]) oder das Modellelement abzufragen, das dieses enthält. Siehe dazu auch die Diskussion über die Abbildungen 3.10 und 3.11.

ModelElement: Das *ModelElement* bringt zum Ausdruck, dass es sich bei den Metamodell-Instanzen um einen Graphen handelt, welcher eine entsprechende Sicht auf das zugrundeliegende System bietet. Die Instanzen der einzelnen Metamodell Klassen bilden dabei die Knoten dieses Graphen, welcher sich zum Beispiel mit entsprechenden Werkzeugen visualisieren lässt. Weiterhin ermöglicht diese gemeinsame Oberklasse die Anwendung von Graphalgorithmen, sei es zur Berechnung entsprechender Graphwerte und -pfade oder zur Berechnung geeigneter Visualisierungs-Layouts, da diese Algorithmen im Allgemeinen als Eingabe nur eine Menge

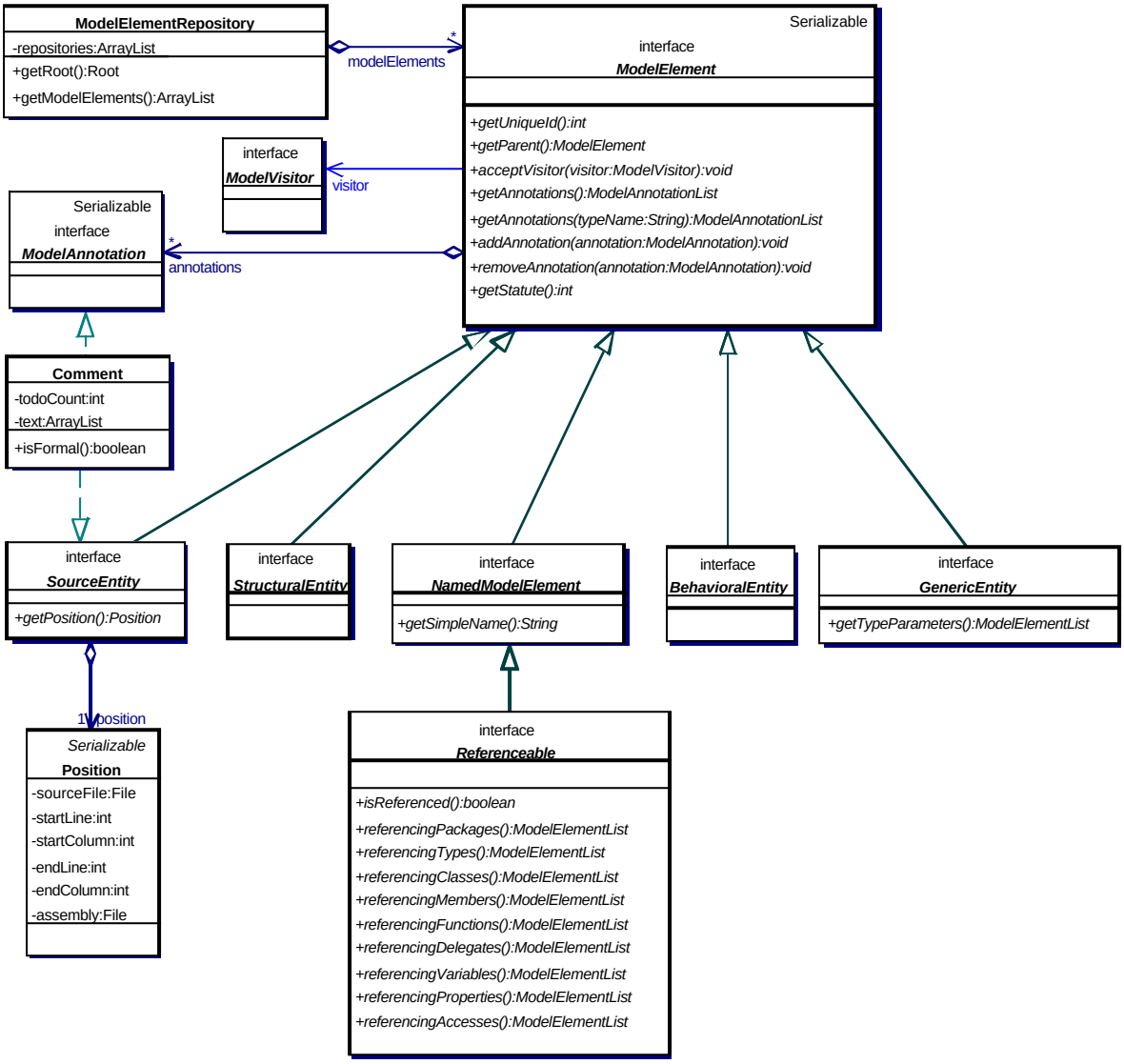

Abbildung 3.9: Überblick über das Analysemodell

von Knoten und Kanten benötigen. Die Modellannotationen erlauben dem Anwender des Metamodells dieses durch eigene Informationen zu erweitern, wie zum Beispiel durch Stereotypes.

Die Schnittstelle bietet die folgenden Operationen an:

ModelElement	
Methoden	
getUniqueId()	Gibt den eindeutigen Bezeichner dieses Knotens zurück, der während der Serialisierung des Modells verwendet wird.
getParent()	Gibt das Modellelement zurück, das dieses in sich kapselt.

acceptVisitor(visitor)	Diese Methode unterstützt die Implementierung des Besucher-Entwurfmusters, wie es in[GHJV95]beschrieben ist.
getAnnotations(String)	Gibt die Liste der Modellannotationen eines vorgegebenen Typs dieses Modellelements zurück.
getAnnotations()	Gibt die Liste der Modellannotationen zu diesem Modellelement zurück.
addAnnotation(ModelAnnotation)	Fügt diesem Modellelement eine Annotation hinzu.
removeAnnotation(ModelAnnotation)	Entfernt eine Annotation dieses Modellelements.
getStatute()	Gibt den Zustand dieses Modellelements zurück. Der Zustand kann eine von den folgenden Werten haben: *normal* für Modellelemente die vom Quellcode extrahiert werden, *library* für Modellelemente die von einer Bibliothek extrahiert werden, *implicit* für Modellelemente die implizit im Quellcode spezifiziert werden und *failed dependency* für Modellelemente die keine Definition im Quellcode oder in einer Bibliothek haben.

Die *ModelElement* Schnittstelle ist durch eine Vielzahl von Schnittstellen, die jeweils zusätzliche Rollen hinzufügen, verfeinert.

SourceEntity: Die Schnittstelle *SourceEntity* bietet im Wesentlichen Informationen, wo das Quellcodeartefakt im System definiert ist. Sie dient zur Unterscheidung der Modellelemente, welche explizit durch Quelltext oder eine ähnliche Spezifikation (zum Beispiel Bytecode) gegeben sind, und denen, die nur implizit durch letztere definiert werden, zum Beispiel durch Zusammenschluss mehrerer dieser expliziten Elemente. Diese implizit definierten Elemente haben somit keine Entsprechung im Quelltext des Systems, auch wenn sie primäre Elemente des Systems sind. Ein Beispiel für solch ein Element sind Paketen in Java. Es sind zwar explizite Elemente des Java-Sprachmetamodells, doch existiert für sie keine explizite Definition im Quelltext. Stattdessen werden sie implizit durch die im Paket enthaltenen Klassen definiert. Die Schnittstelle erweitert das *ModelElement* um eine *Position* Klasse und sorgt für den Zugriff auf das entsprechende *Position* Objekt durch get- und set-Methoden. *SourceEntity* definiert die folgenden Methoden:

SourceEntity	
Methoden	
getPosition()	Gibt die Position im Quellcode zurück.

Position: Die *Position* Klasse zeigt die Position eines Modellelementes im Quelltext an. Es werden dabei die folgenden Informationen bereitgestellt:

Position	
Attribute	
sourceFile	Verweis auf das Objekt, welches die Datei repräsentiert, in welcher die Deklaration des Elements zu finden ist.

startLine	Startzeile des Elements im Quelltext.
startColumn	Startspalte des Elements im Quelltext.
endLine	Letzte Zeile des Elements im Quelltext.
endColumn	Letzte Spalte des Elements im Quelltext.
assembly	Gibt die Lage (Datei) dieser *SourceEntity* im kompilierten Projekt wieder.

StructuralEntity: Die konkreten Unterklassen der Schnittstelle *StructuralEntity* bilden die Strukturierungsmöglichkeiten der benutzten Programmiersprachen ab. Für objektorientierte Systeme sind dies im Wesentlichen die aus der UML bekannten Entitäten zur Beschreibung der statischen Struktur eines Softwaresystems.

NamedModelElement: Die Schnittstelle *NamedModelElement* ist die gemeinsame Oberklasse für alle Modellentitäten, die über einen deklarierten Namen bzw. Identifizierer verfügen. Diese Entitäten sind somit über ihren Namen bzw. Identifizierer referenzierbar.

NamedModelElement	
Methoden	
getSimpleName()	Gibt den einfachen, unqualifizierten Namen dieses benannten Modelelements wieder.

BehaviouralEntity: Instanzen der Schnittstelle *BehaviouralEntity* sind die Träger des Kontrollflusses eines Systems, das heißt sie definieren das Laufzeitverhalten dieses Systems.

GenericEntity: Die Schnittstelle *GenericEntity* wird als gemeinsamer Basistyp für *GenericFunction* und *GenericClass* verwendet. Sie bietet die folgenden Methoden an:

GenericEntity	
Methoden	
getTypeParameters()	Liefert die Liste der Typparameter dieser generischen Entität.

Referenceable: Die Schnittstelle *Referenceable* drückt aus, ob eine Entität des zu analysierenden Systems durch einen Identifizierer referenziert werden kann. Daher ist die Schnittstelle *Referenceable* eng mit der Schnittstelle *NamedModelElement* verbunden, da der Name einer Entität einen solchen Identifizierer darstellt. Da die Schnittstelle *Referenceable* neben dieser Eigenschaft auch noch Operationen zur Querverweisbestimmung (*crossreferencing*) bereitstellt, wird sie nicht mit *NamedModelElement* zusammengelegt, sondern als eigene Schnittstelle modelliert. Dabei sind folgende Anfragemethoden von Bedeutung:

Referenceable
Methoden

isReferenced()	Zeigt an, ob das entsprechende Modellelement von einem anderen Modellelement referenziert wird.
referencingX()	Liefert eine Liste mit Elementen vom Typ X zurück, in deren Gültigkeitsbereich das *Referenceable* Objekt referenziert wird. Wird zum Beispiel eine Klasse X im Rahmen von Variablendeklarationen i und j innerhalb zweier Klassen Y und Z referenziert, so werden diese beiden Klassen Y und Z zurückgegeben, falls X die Anfrage *referencingClasses()* erhält. Erhält die Klasse X die Anfrage *referencingVariables()*, so werden die Variablendeklarationen i und j zurückgegeben.

ModelElementRepository: Die *ModelElementRepository* Klasse bietet Persistenzdienstleistungen, wie auch die Behandlung mehrerer Modellinstanzen. Der statische Teil der Klasse kümmert sich um die verschiedenen Modellinstanzen, während der Instanzenteil der Klasse die Modellelemente verwaltet, die zu einem vorgegebenen Repository gehören.

ModelElementRepository	
Attribute	
repositories	Ein statisches Feld, das die Liste der Modellinstanzen verwaltet, die sich gerade im Speicher befinden.
Methoden	
getRoot()	Gibt das *Root*-Element dieser Modellinstanz zurück. Siehe die weiteren Erklärungen unten.
getModelElements()	Gibt die Liste der Modellelementen in dieser Modellinstanz zurück.

ModelVisitor: Die ModelVisitor Schnittstelle muss durch einen beliebigen Nutzer implementiert werden. Weitere Informationen dazu sind im Visitor Entwurfsmuster unter [GHJV95] nachzulesen.

ModelAnnotation: Die *ModelAnnotation* Schnittstelle ist eine Markierungsschnittstelle, die durch jede Klasse, die ein Modelelement kommentiert, implementiert werden muss.

Comment: Die *Comment* Klasse modelliert Code-Kommentare. Sie implementiert *ModelAnnotation*, so dass damit jedes beliebige Modellelement annotiert werden kann. Die *Comment* Klasse bietet die folgenden Merkmalen an:

Comment	
Attribute	
text	Liste der Kommentarzeilen.
todoCount	Anzahl der *TODO*-, *HACK*- oder *FIXME*-Tags in diesem Kommentar.
Methoden	
isFormal()	Liefert *wahr*, falls dieser Kommentar ein formaler Kommentar ist.

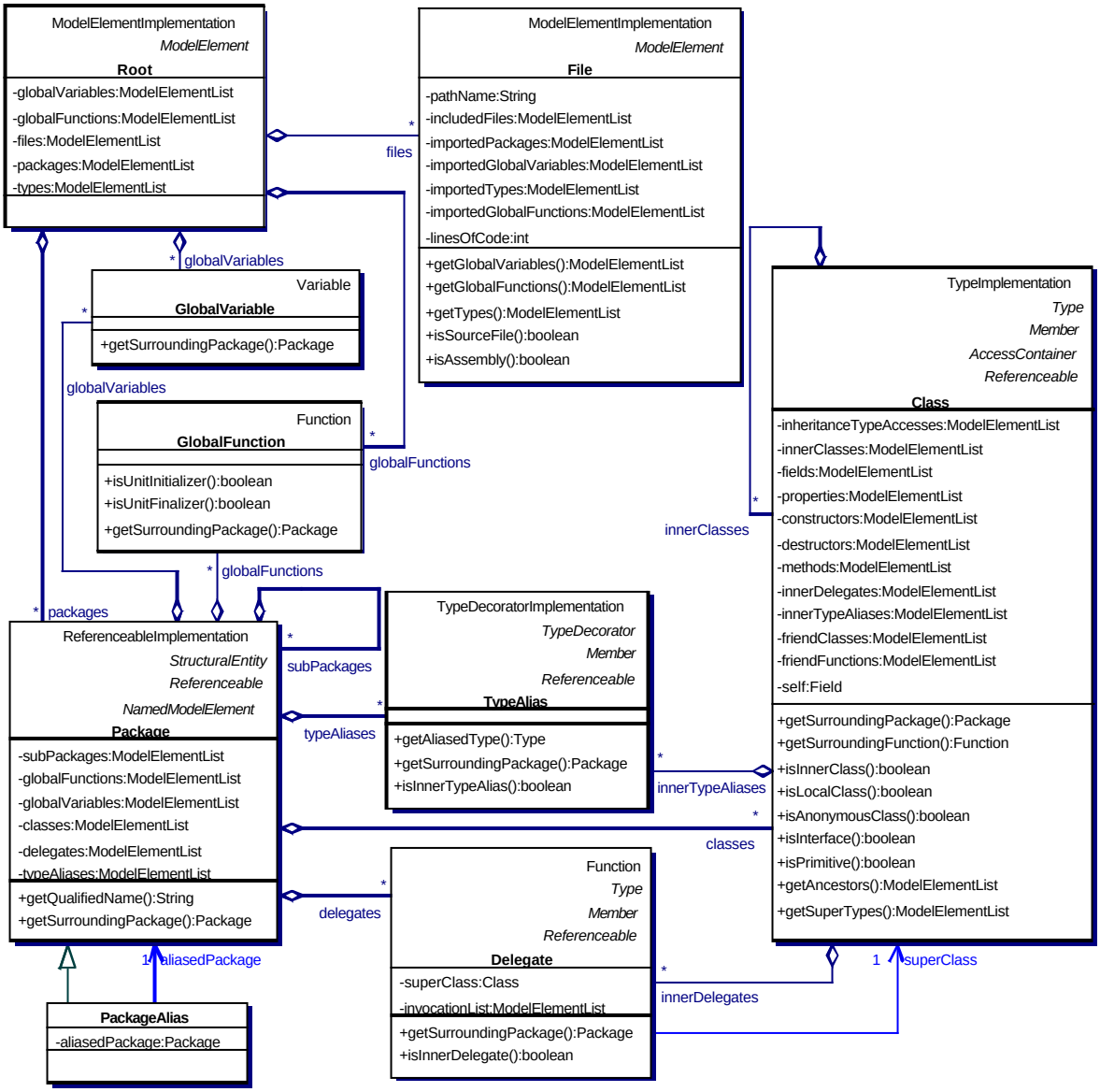

Abbildung 3.10: Der Kern der Modellelemente 1

Strukturierungselemente

Wie Abbildung 3.10 zeigt, sind die Modellelemente in einem Aggregationsbaum angeordnet, dessen Wurzel die *Root* Klasse ist. Die pro Metamodellinstanz einzige Instanz dieser Klasse fungiert als Einstiegspunkt bezüglich der Navigation in diesem Modell. Sie bündelt in Form eines Containers alle freien Strukturierungsentitäten. Freie Strukturierungsentitäten sind dabei Entitäten, die nicht innerhalb einer anderen Strukturierungsentität angegeben sind. Beispiele hierfür sind Pakete sowie globale Funktionen und Variablen. Da die Klasse *Root* im Wesentlichen nur dazu benötigt wird, alle freien Strukturierungselemente zu bündeln, besitzt sie keine persistenten In-

formationen, da die entsprechenden Mengen während des Modellaufbaus automatisch bestimmt werden können. Anfragemethoden der Klasse *Root* sind die folgenden:

Root	
Attribute	
globalVariables	Alle im System vorkommenden globalen Variablen.
globalFunctions	Analog dazu alle vorkommenden globalen Funktionen.
packages	Liste aller Pakete im System. Die in anderen Paketen geschachtelten Pakete werden hier auch aufgelistet.
files	Liste aller Quelldateien im System.
types	Liste aller im System definierten oder verwendeten Typen.

Package: Ein *Package* ist in erster Linie ein Gruppierungsmechanismus für Klassen und weitere darin enthaltenen Paketen ohne weitere Semantik bezüglich Sichtbarkeit und Namensräumen. Sollte ein Paket eine darüber hinausgehende Semantik in einer konkreten Programmiersprache besitzen, so ist dies Teil der sprachabhängigen Erweiterung des Modells. Paketen können geschachtelt sein, das heißt ein Paket kann beliebig viele weitere Paketen enthalten. Bezüglich Namensräumen solch geschachtelter Paketen gilt das vorher gesagte.

Da ein *Package* ein Gruppierungsmechanismus für Klassen und weitere darin enthaltene Paketen ist, müssen diese Beziehungen auch in den persistenten Attributen zum Ausdruck kommen. Daher besitzt diese Klasse folgende Merkmale:

Package	
Attribute	
subPackages	Die Menge aller der in diesem Paket enthaltenen Unterpaketen.
classes	Die Menge aller der in diesem Paket enthaltenen Klassen. Beinhaltet nicht die transitiv über Unterpaketen enthaltenen Klassen.
delegates	Die Menge aller in diesem Paket direkt definierten *Delegates*.
typeAliases	Die Menge aller in diesem Paket direkt definierten *TypeAliases*.
globalVariables	Die Menge aller in diesem Paket direkt definierten globalen Variablen.
globalFunctions	Die Menge aller in diesem Paket direkt definierten globalen Funktionen.
Methoden	
getQualifiedName()	Gibt den vollqualifizierten Namen dieses Pakets wieder.
getSurroundingPackage()	Gibt das umliegende Ausgangspaket dieses Pakets wieder.

PackageAlias: In einigen OO Sprachen, wie zum Beispiel C# kann man einen kurzen Alias für einen langen Namespace-Namen bilden, um somit durch diesen Alias auf den Namespace zu verweisen. Die beiden Namen entsprechen einander und die *PackageAlias* Konstruktion verdeutlicht diese Entsprechung ausdrücklich.

PackageAlias	
Attribute	
aliasedPackage	Das tatsächliche Paket auf das sich der Alias bezieht.

File: Da im Rahmen von Qualitätsanalysen auch der Aufbau des zu untersuchenden Systems auf Dateiebene von Interesse sein kann, stellt das hier vorgestellte Metamodell ein entsprechendes Element zur Verfügung. Es handelt sich dabei um das in Abbildung 3.10 dargestellte Modellelement *File*, in dessen Kontext sowohl Klassendeklarationen, als auch globale Variablen und globale Funktionen stehen können. Weiterhin kann eine Datei weitere Dateien importieren, was ebenfalls in Abbildung 3.10 durch die *imports*-Assoziationen dargestellt ist. Eine Instanz der Klasse *File* stellt im Allgemeinen eine Übersetzungseinheit dar.

Die Metamodellklasse *File* bietet die folgenden persistenten Attribute:

File	
Attribute	
pathName	Der Dateiname der Datei im Dateisystem, falls möglich inklusive Pfad.
includedFiles	Die Menge der Dateien, welche von dieser Datei direkt importiert werden.
importedPackages	Die Menge der Pakete, welche von dieser Datei direkt importiert werden.
importedGlobalVariables	Die Menge der im Rahmen der Quellcodedatei deklarierten globalen Variablen. Enthält auch Deklarationen, die mit dem Schlüsselwort "extern" bezeichnet sind.
importedGlobalFunctions	Die Menge der im Rahmen der Quellcodedatei deklarierten globalen Funktionen. Enthält auch Prototypdeklarationen.
importedTypes	Liste der importierten Typen.
linesOfCode	Anzahl der Codezeilen.
Methoden	
getGlobalVariables()	Gibt die Liste der in dieser Datei direkt definierten globalen Variablen wieder.
getGlobalFunctions()	Gibt die Liste der in dieser Datei direkt definierten globalen Funktionen wieder.
getTypes()	Gibt die Liste der in dieser Datei direkt definierten Typen wieder.
isSourceFile()	Liefert *wahr*, falls es sich dabei um eine Quelldatei handelt.
isAssembly()	Liefert *wahr*, falls es sich dabei um eine Entwicklungseinheit handelt (kompilierte Assembly, JAR-Datei oder Delphi-Package).

Class: Dieses Modellelement entspricht dem gleichnamigen Element aus der UML. Es definiert einen neuen Typ, kann mit anderen Klassen in einer Vererbungsbeziehung stehen und beinhaltet Klassenmerkmale wie Methoden und Attribute. Primitive Typen, wie sie in vielen objektorientierten Sprachen zu finden sind, werden auch auf das Modellelement *Class* abgebildet.

Im Gegensatz zur Oberklasse *Type* bietet der Typ *Class* zahlreiche Informationen und Anfragemethoden an. Dies liegt nicht zuletzt daran, dass die Klasse ein sehr zentrales Konzept in der objektorientierten Programmierung ist. Die einzelnen bereitgestellten Merkmale sind die folgenden:

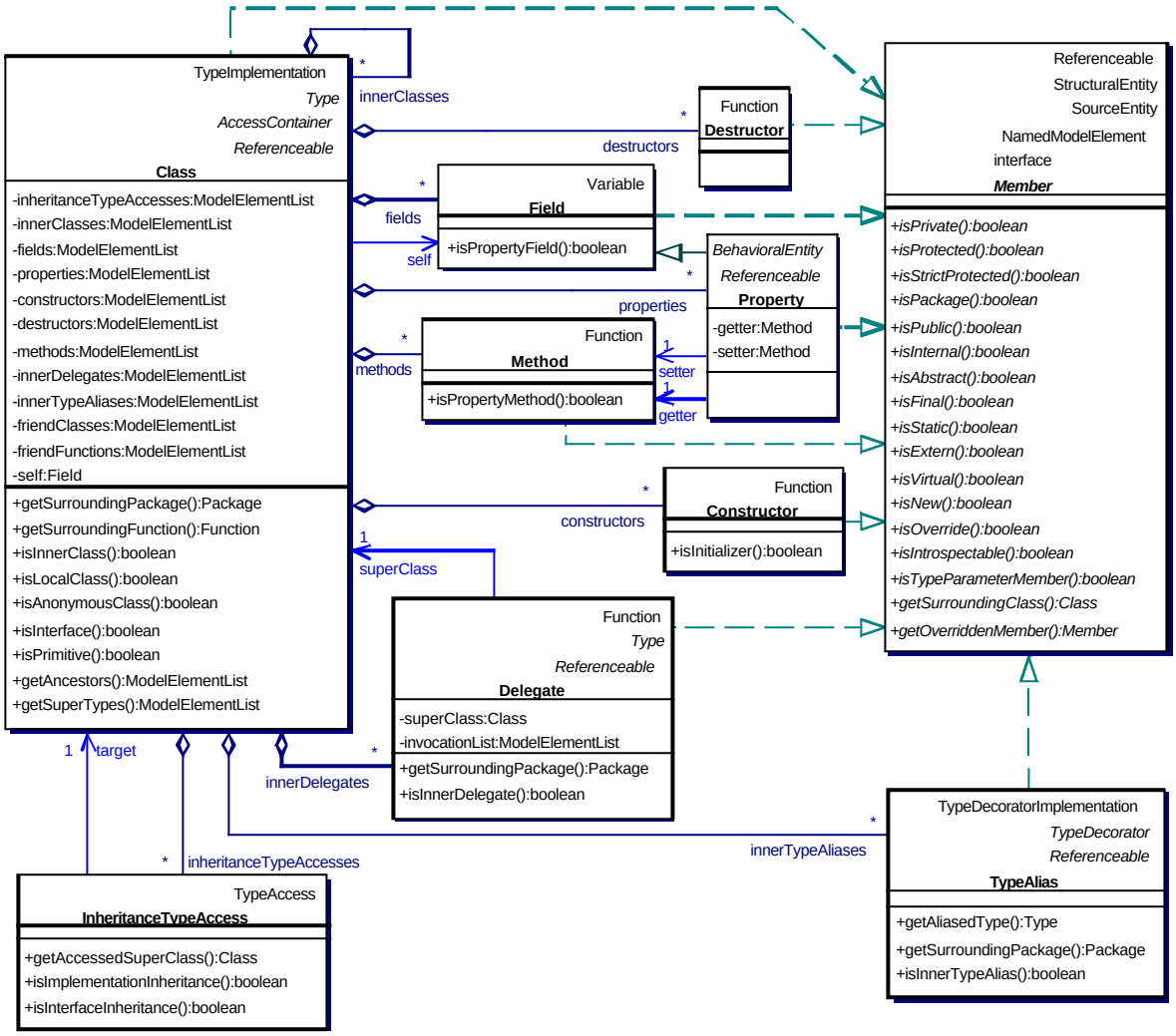

Abbildung 3.11: Der Kern der Modellelemente 2

Class	
Attribute	
inheritanceTypeAccesses	Eine Liste der direkten Vererbungsbeziehungen dieser Klasse. Siehe auch Abbildung 3.15
friendClasses	Liste deklarierter Friend Klassen.
friendFunctions	Liste deklarierter Friend-Funktionen.
members/innerMembers	Enthält mehrere Listen zum Speichern von Elementen, die zu dieser Klasse gehören. Für jeden Typ der gespeicherten Elemente wird eine eigene Liste verwaltet. Diese Typen sind: *Field*, *Method*, *Constructor*, *Destructor*, *Property*, *Class*, *Delegate* und *TypeAlias*. Alle diese Typen implementieren die Schnittstelle *Member*, die später noch beschrieben wird.
self	Ein Hilfskonstrukt, das als Ziel aller impliziten und expliziten *this*- und *super*-Zugriffe benutzt wird.

Methoden	
getSurroundingPackage()	Liefert einen Verweis auf das Paket, in welchem die Klasse enthalten ist.
getSurroundingFunction()	Liefert für lokale Klassen eine Referenz auf die sie direkt umgebende Funktion.
isInnerClass()	Prüft, ob es sich bei dieser Klasse um eine innere Klasse, das heißt im Kontext einer anderen Klasse deklarierten Klasse, handelt.
isLocalClass()	Liefert *wahr*, falls die Klasse im Rahmen einer *BehavioralEntity* deklariert wurde.
isAnonymousClass()	Zeigt an, ob es sich bei der Klasse um eine sogenannte „anonyme“ Klasse, das heißt eine Klasse ohne explizite Namensangabe bei der Deklaration, handelt.
isInterface()	Liefert *wahr*, falls es sich um eine Schnittstelle handelt.
isPrimitive()	Liefert *wahr*, falls es sich um einen primitiven Typ handelt.
getSuperTypes()	Gibt alle direkten Obertypen dieser Klasse wieder.
getAncestors()	Gibt alle direkten Vorgänger dieser Klasse wieder. Diese Liste beinhaltet zudem Oberklassen, auf die durch Implementierungsvererbung (private) verwiesen wird.

Member: Bei den konkreten Unterklassen der Schnittstelle *Member* handelt es sich um mögliche Merkmale einer Klasse. In unserem Modell ist das Modellelement *Class* selbst ein mögliches Klassenmerkmal. Dies liegt daran, dass zahlreiche objektorientierte Sprachen innere Klassen als Merkmale der äußeren Klasse betrachten.

Ein Merkmal einer Klasse ist, wie der Name bereits sagt, durch die Zugehörigkeit des Elements zu einer Klasse gekennzeichnet. Ein weiteres Kennzeichen eines Merkmals ist in vielen Sprachen die mögliche Vergabe von bestimmten Modifizierern, welche zum Beispiel eine Zugriffskontrolle auf die Merkmale ermöglichen oder weitere Eigenschaften des Merkmals beschreiben. In unserem Modell bietet die Klasse *Member* daher folgende Attribute und Anfragemethoden:

Member	
Methoden	
isPrivate()	Prüft, ob das Merkmal mit privater Sichtbarkeit deklariert wurde.
isProtected()	Prüft analog zu *isPrivate()*, ob das Merkmal mit *protected* Sichtbarkeit deklariert wurde.
isStrictProtected()	Prüft analog zu *isPrivate()*, ob das Merkmal mit *strict protected* Sichtbarkeit deklariert wurde. Diese Sichtbarkeit wird nur in Delphi getroffen.
isPackage()	Prüft analog zu *isPrivate()*, ob das Merkmal package-local deklariert wurde.
isPublic()	Zeigt an, ob das Merkmal mit öffentlicher Sichtbarkeit versehen wurde.
isInternal()	Liefert *wahr*, falls dieses Merkmal innerhalb seiner eigenen assembly zugreifbar ist.
isStatic()	Liefert *wahr*, falls für den Zugriff auf ein Merkmal nur die Klasse und nicht eine konkrete Instanz benötigt wird.

isAbstract()	Liefert *wahr*, falls das Merkmal als abstrakt deklariert wurde. Dies kann je nach konkretem Merkmaltyp eine unterschiedliche Semantik haben.
isFinal()	Zeigt an, ob das Merkmal überschrieben werden darf oder nicht.
isExtern()	Liefert *wahr*, falls dieses Merkmal die Deklaration einer externen Entität ist.
isVirtual()	Prüft ob dieses Merkmal überschrieben werden kann oder nicht.
isNew()	Liefert *wahr*, falls dieses Merkmal neu in der Klassenhierarchie ist und kein überschriebenes Merkmal korrigiert.
isOverride()	Liefert *wahr*, falls dieses Merkmal ein geerbtes Merkmal überschreibt.
isIntrospectable()	Liefert *wahr*, falls dieses Merkmal durch Introspektion sichtbar ist.
getOverridenMember()	Liefert eine Referenz auf das geerbte Merkmal, das durch dieses Merkmal überschrieben wird.
isTypeParameterMember()	Prüft ob die umgebende Klasse dieses Merkmals eine *TypeParameterClass* ist. Siehe unten die weitere Erklärung.
getSurroundingClass()	Liefert die lexikalisch umgebende Klasse, in deren Kontext das Merkmal deklariert wurde.

Typhierarchie

Abbildung 3.12 zeigt die Vererbungshierarchie der verschiedenen Typen. Die Schnittstelle *Type* dient der Abbildung des Typsystems einer Programmiersprache in das vorgestellte Metamodell. Sie dient dazu, einen beliebigen Typ zu fassen. *Type* bietet außer dem von *NamedModelElement* ererbten Namen die folgenden Informationen:

Type: Die Schnittstelle *Type* ist durch *Class*, die oben beschrieben wurde, und *Delegate* implementiert. Die Klasse *Class* modelliert auch die primitiven Typen, die in allen gängigen objektorientierten Sprachen zu finden sind, wie zum Beispiel *int*, *float*, *double* usw.

Type	
Methoden	
getQualifiedName()	Gibt den voll qualifizierten Namen dieses Typs zurück.
isReferenceType()	Liefert *wahr*, falls es sich um einen Referenztyp handelt.
isValueType()	Liefert *wahr*, falls es sich um einen Werttyp handelt. Für mehr Informationen über den Unterschied zwischen Referenz- und Werttyp siehe [Int02].

GenericClass: Die Klasse *Class* wird noch durch *GenericClass* erweitert, die auch die Schnittstelle *GenericEntity* implementiert. Wie bereits vom Namen angedeutet wird, modelliert *GenericClass* eine generische Klasse. Sie ergänzt *Class* durch die folgende Information:

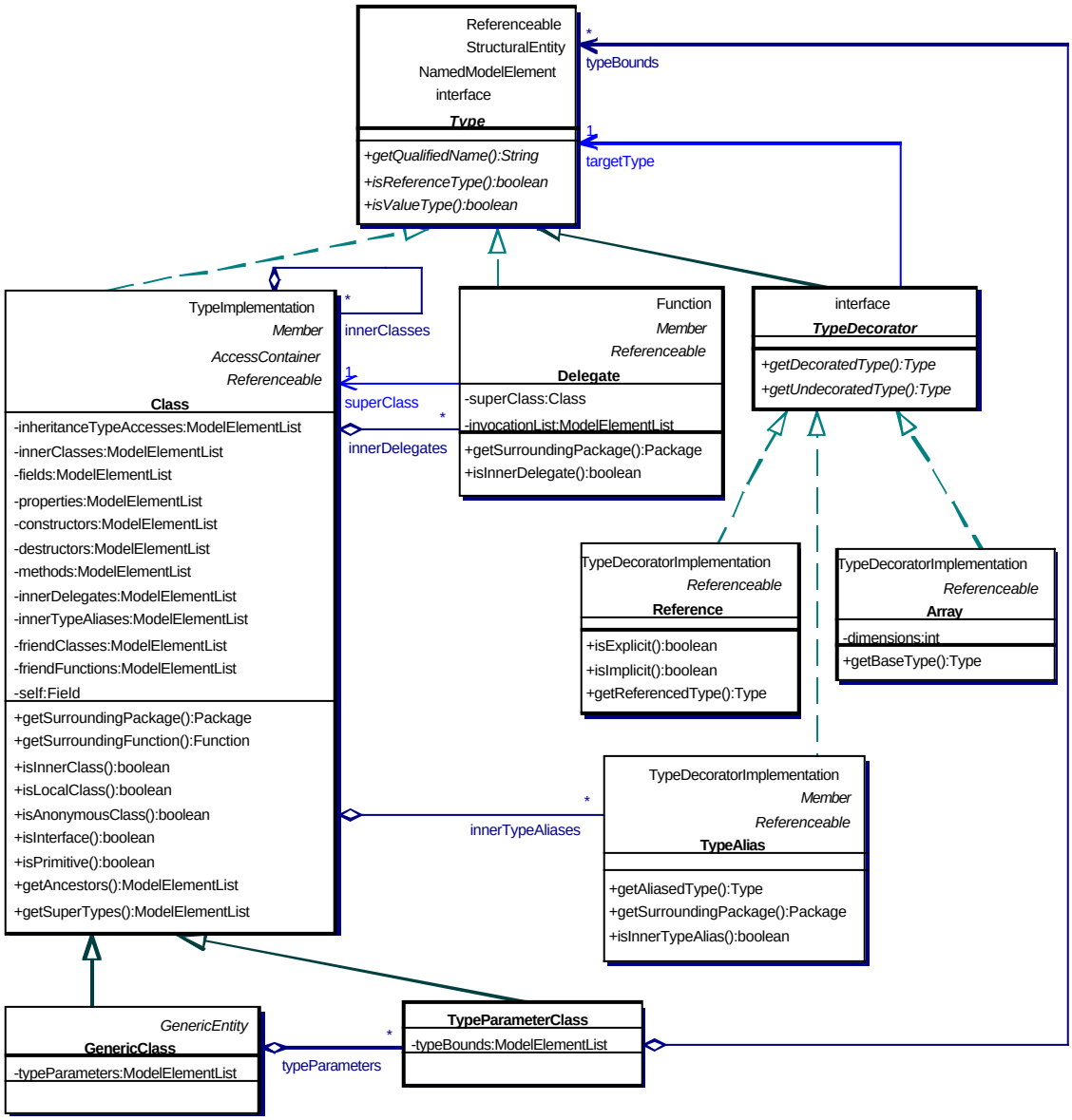

Abbildung 3.12: Die Typhierarchie

GenericClass	
Attribute	
typeParameters	Liste der Typparameter dieser generischen Klasse.

TypeParameterClass: Die Typparameter in der Liste *typeParemeters* sind Instanzen der Klasse *TypeParameterClass*. Eine *TypeParameterClass* ist eine Verfeinerung von *Class*, aus diesem Grund kann sie wie eine Klasse in sich geschachtelte Elemente (*Member*) besitzen. Jede *TypeParameterClass* hat eine Liste von konkreten Typen (type bounds), woran der jeweilige Typparameter der generischen Klasse bei der Erzeugung einer Ausprägung gebunden wird.

TypeParameterClass	
Attribute	
typeBounds	Liste der Typbindungen dieser *TypeParameterClass*.

Delegate: Delegates stellen die moderne Methode Function-Pointers einzukapseln dar. Sie sind in C# als aufrufbare Typen definiert. Siehe dazu [Int02]. Die Delegate Klasse hat folgende Merkmale:

Delegate	
Attribute	
superClass	Dies ist eine Referenz auf die gemeinsame, direkte Oberklasse aller Delegates, die *System.Delegate* Klasse, die Teil der *Common Language Infrastructure (CLI), Base Class Library (BCL)* ist.
invocationList	Ein Delegate aufzurufen kann zu Aufrufen mehrerer Methoden führen, da sich Delegates wie "multicast call forwarder" verhalten können. Dieses Feld beinhaltet die Liste aller Methoden, die möglicherweise aufgerufen werden, wenn dieses Delegate aufgerufen wird.
Methoden	
getSurroundingPackage()	Liefert einen Verweis auf das Paket, in welchem diese *Delegate* enthalten ist.
isInnerDelegate()	Prüft, ob es sich bei diesem Delegate um eine innere Klasse, das heißt im Kontext einer Klasse deklarierten Delegate, handelt.

TypeDecorator: Die Schnittstelle *TypeDecorator* wird benutzt um die abgeleiteten Typen wie Reihungen und Typaliasen zu modellieren. Sie definiert die folgenden Methoden:

TypeDecorator	
Methoden	
getDecoratedType()	Gibt den gekapselten Typ zurück, der ggf. auch ein *TypeDecorator* sein kann.
getUndecoratedType()	Gibt den innersten Typ zurück, d.h. alle *TypeDecorators* werden entfernt.

Array: Die Klasse *Array* implementiert die Schnittstelle *TypeDecorator*. *Array* bietet die folgenden Informationen:

Array	
Attribute	
dimensions	Die Anzahl der Dimensionen dieser Reihung.
Methoden	
getBaseType()	Gibt den Basistyp dieser Reihung zurück. Bei einer mehrdimensionalen Reihung würde die *getDecoratedType()*-Methode eine Reihung zurückgeben, deren Dimension um eins kleiner ist, als die von der originellen Reihung. Die Methode *getUndecoratedType()* würde den innersten Typ in der Reihung zurückgeben, der nicht unbedingt vom Typ *TypeDecorator* ist. Diese Methode liefert den dekorierten Typ der innersten, eindimensionalen Reihung.

Reference: Die Referenzklasse modelliert C++ Pointer und Referenzen. Sie bietet die folgenden Informationen:

Reference	
Methoden	
isExplicit()	Liefert *wahr*, falls es sich um eine Hinweismarke auf einen Typ handelt.
isImplicit()	Liefert *wahr*, falls es sich um eine Referenz auf einen Typ handelt.
getReferencedType()	Gibt den dereferenzierten Typ wieder.

TypeAlias: Diese Klasse modelliert Typaliasen (C++ typdef-Konstrukte). Sie bietet die folgenden Informationen:

TypeAlias	
Methoden	
getAliasedType()	Liefert eine Referenz auf den Typ des Typalias.
getSurroundingPackage()	Liefert einen Verweis auf das Paket, in welchem dieses Typalias enthalten ist.
isInnerTypeAlias()	Prüft, ob es sich bei diesem Typalias um eine innere Klasse, das heißt im Kontext einer Klasse deklarierten Typalias, handelt.

Hierarchie der Funktionen

Die Modellierung von Funktionen jeglicher Art ist in Abbildung 3.13 zu sehen. Es werden dabei die in objektorientierten Systemen anzutreffenden Typen von Funktionen modelliert. Da Funktionen stets die Träger des Kontrollflusses sind, bestimmen sie in ihrer Gesamtheit das Verhalten des Systems. Daher implementieren sie alle die Schnittstelle *BehavioralEntity*. Da sie gleichzeitig im Quelltext oder im Kompilat vorliegende Strukturierungselemente sind, ist eine Funktion stets auch eine Instanz von *SourceEntity* und *BehavioralEntity*. Da Funktionen mit Hilfe ihrer Signatur

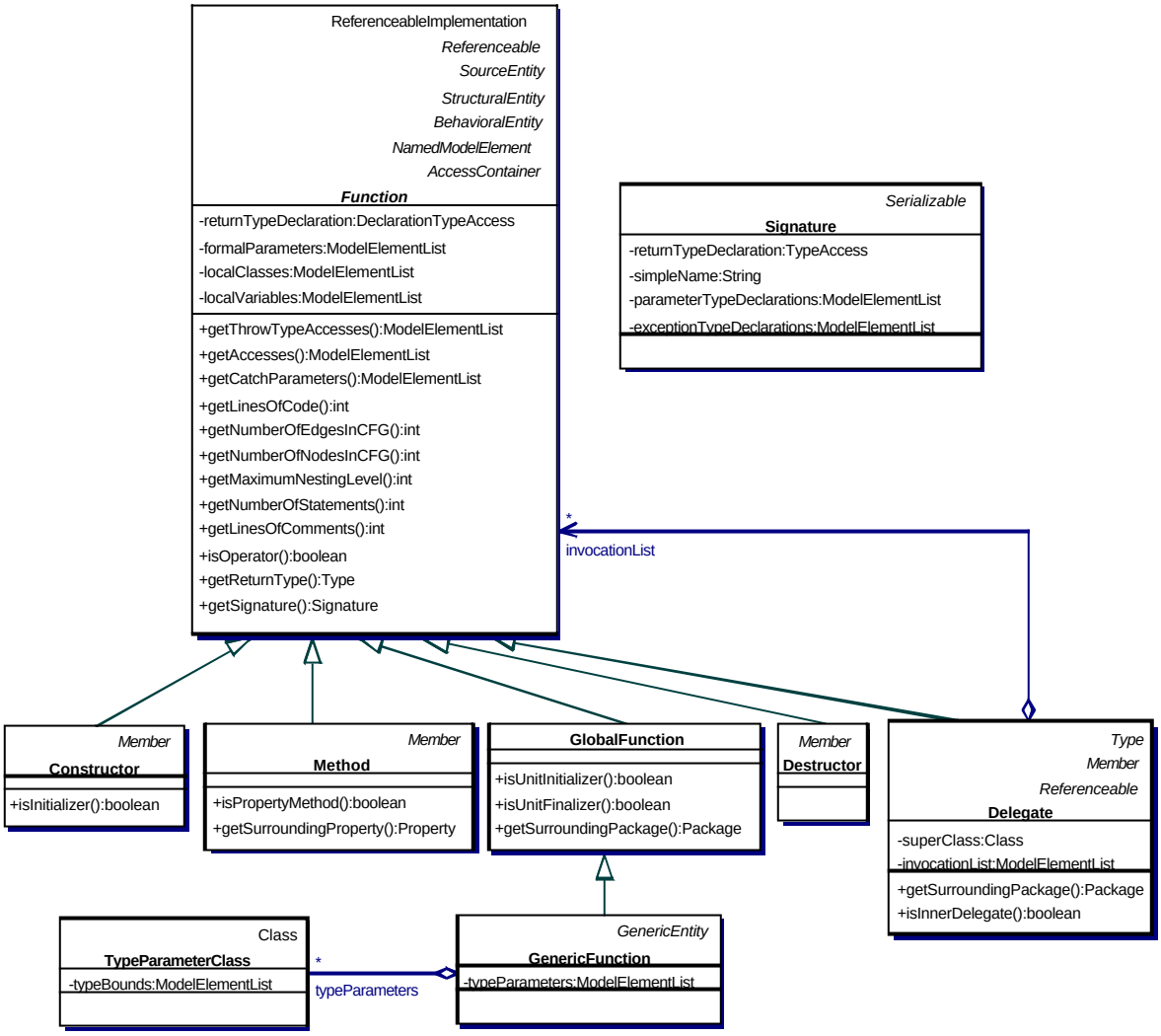

Abbildung 3.13: Die Hierarchie der Funktionen

identifiziert werden, handelt es sich bei ihnen auch um eine Implementierung der Schnittstelle *NamedModelElement*.

Abbildung 3.13 zeigt die für Funktionen bereitgestellten Modellinformationen. Dabei fällt zunächst auf, dass nahezu alle Informationen bereits in der abstrakten *Function* zur Verfügung gestellt werden. Es ist aber dabei noch zu beachten, dass die konkreten Klassen *Method*, *Constructor* und *Destructor* weitere Informationen über die Schnittstelle *Member* erben. *Constructor* und *Destructor* vererbt werden. Eine weitere Differenzierung der konkreten Unterklassen von *Function* wird sich in der Erweiterung des Analysemodells hin zu einem Transformationsmodell ergeben.

Function: Hierbei handelt es sich um die gemeinsame Oberklasse für sämtliche Arten von Funktionen. In ihr werden bereits die meisten Anfrage- und Transformationsoperationen beschrieben, da diese für die verschiedenen Arten von Funktionen identisch sind. Die Klasse selbst ist abstrakt, das heißt sie kann nicht instanziiert werden, da sie kein explizites Pendant in den jeweiligen Sprachmetamodellen besitzt.

Da die Funktion neben der Klasse eine der wichtigsten Modellierungsentitäten für Softwaresysteme darstellt, fällt die Anzahl der bereitgestellten Informationen ähnlich umfangreich aus. Im Einzelnen bietet die Klasse *Function* folgende persistente Attribute und Anfragemethoden:

Function	
Attribute	
returnTypeDeclaration	Ein Typzugriff, der auf den deklarierten Rückgabetyp dieser Funktion zeigt.
formalParameters	Liste mit den formalen Parametern der Funktion.
localClasses	Menge der in der Funktion deklarierten lokalen Klassen.
localVariables	Menge aller in der Funktion deklarierten lokalen Variablen.
Methoden	
getCatchParameters()	Gibt die Liste der behandelten Ausnahmen in dem Funktionsrumpf wieder.
getThrowTypeAccesses()	Gibt die Liste von *ThrowTypeAccesses* wieder.
getAccesses()	Liefert die Liste aller Zugriffe, die in dieser Methode enthalten sind, inklusive Rückgabetypdeklaration und *ThrowTypeAccesses*.
getLinesOfCode()	Gibt die Anzahl der Codezeilen der Funktion wieder, inklusive Funktionskopf, Kommentaren und Leerzeilen.
getNumberOfEdgesInCFG()	Gibt die Anzahl der Kanten im Kontrollflussgraph des Funktionsrumpfs wieder.
getNumberOfNodesInCFG()	Gibt die Anzahl der Knoten im Kontrollflussgraph der Funktion wieder, das heißt die Anzahl der Verzweigungen im Funktionsrumpf.
getMaximumNestingLevel()	Liefert die Maximale Schachtelungstiefe der Kontrollstrukturen im Funktionsrumpf.
getNumberOfStatements()	Gibt die Anzahl der Anweisungen wieder.
getLinesOfComments()	Gibt die Anzahl der Kommentarzeilen im Kontext der Funktion wieder. Einführende Kommentare vor dem Funktionskopf gehören ebenfalls dazu.
isOperator()	Liefert *wahr*, falls es sich bei dieser Funktion um einen Operator handelt. Operatoren sind als Funktionen modelliert und ihre Verwendung als *FunctionAccesses* wie später genauer erläutert.
getReturnType()	Gibt den deklarierten Rückgabetyp dieser Funktion wieder. Dabei sei zu beachten, dass diese Methode eine Abkürzung für *TypeAccess.getAccesstype()* ist.
getSignature()	Gibt ein *Signature* Objekt zurück, welches die wesentlichen Bestandteile der Signatur der Funktion vereint.

Für die konkreten Instanzen zeichnen sich die folgenden Klassen verantwortlich.

GlobalFunction: Eine globale Funktion ist eine Funktion, welche an jeder beliebigen Stelle innerhalb eines Systems sichtbar und somit zugreifbar ist. Der Gültigkeitsbereich ist also insbesondere nicht auf eine Klassendeklaration beschränkt. *GlobalFunction* wird ebenfalls dazu benutzt, Paketinitialisierer in Delphi zu modellieren.

GlobalFunction	
Methoden	
getSurroundingPackage()	Liefert einen Verweis auf das Paket, in welchem die globale Funktion enthalten ist.
isUnitInitializer()	Liefert *wahr*, falls diese Funktion ein Delphi Unit-Initialisierer modelliert.
isUnitFinalizer()	Liefert *wahr*, falls diese Funktion ein Delphi Unit-Finalisierer modelliert.

Method: Dieses Modellelement dient der Beschreibung von Methoden einer Klasse. Nicht dazu gehören Konstruktoren und Destruktoren, die zwar wie Methoden auch Merkmale einer Klasse sind, aber in unserem Modell explizit modelliert werden. Sowohl Methoden, als auch Konstruktoren und Destruktoren sind Merkmale der Klasse, in deren Kontext sie deklariert wurden. Es handelt sich bei ihnen somit um eine Implementierung der Schnittstelle *Member*. Die Klasse *Method* bietet die folgende Information:

Method	
Methoden	
isPropertyMethod()	Zeigt, ob die Methode eine Property-Zugriffsmethode ist. Siehe auch die Beschreibung der Klasse *Property*.
getSurroundingProperty()	Liefert für Property-Zugriffsmethode eine Referenz auf die sie direkt umgebende Property.

Constructor: Ein Konstruktor ist eine Funktion, die während der Erzeugung einer Instanz der definierenden Klasse ausgeführt wird. Konstruktoren werden in unserem Modell explizit modelliert und nicht als spezielle Methode betrachtet, da insbesondere einige Basistransformationen für Konstruktoren nicht sinnvoll bzw. erlaubt sind (z.B. Umbenennen in Java/C++). Klassen- und Feldinitialisierer sind ebenfalls als Konstruktoren modelliert. Da ein Initialisierer jedoch nicht ausdrücklich wie ein Konstruktore aufgerufen werden, wird zwischen ihnen mittels folgender Anfragen unterschieden:

Constructor	
Methoden	
isInitializer()	Liefert *wahr*, falls es sich um einen Initialisierer handelt.

Destructor: Ähnlich wie Konstruktoren werden Destruktoren aufgerufen, sobald eine Instanz der umgebenden Klasse gelöscht wird. Sie sind nicht wie Konstruktoren in allen objektorientierten Sprachen zu finden.

GenericFunction: Da eine Funktion nur in dem Fall generisch sein kann, wenn sie außerhalb einer Klasse, als eine globale Funktion deklariert wurde, erweitert die Klasse *GenericFunction* die Entität *GlobalFunction*. Ähnlich zur *GenericClass*, modelliert eine *GenericFunction* eine generische, globale Funktion. Sie erweitert *GlobalFunction* um die folgende Information:

GenericFunction	
Attribute	
typeParameters	Liste der Typparameter dieser generischen Funktion.

Signature: Repräsentiert die Signatur einer Methode.

Signature	
Attribute	
returnTypeDeclaration	Ein Typzugriff, der auf den deklarierten Rückgabetyp der Funktion hinweist.
simpleName	Das Merkmal *simpleName* der Klasse *Function*.
parameterTypeDeclarations	Liste der Typzugriffe auf die deklarierten Typen der formalen Parameter der Funktion.
exceptionTypeDeclarations	Liste der Typzugriffe auf deklarierte Typen der Funktionsausnahmen. Mehr Informationen dazu sind in den Ausführungen um das *TypeAccess* Modellelement zu finden.

Hierarchie der Variablen

So wie Funktionen das Verhalten eines Systems beschreiben, so beschreiben Variablen die in einem System verfügbaren Daten. Sie spielen daher in einem Metamodell wie diesem ebenfalls eine zentrale Rolle. Da Variablen zur Struktur des Systems beitragen, implementieren sie die Schnittstelle *StructuralEntity*. Da Variablen über ihren Namen angesprochen werden und folglich auch einen solchen besitzen, sind alle Typen von Variablen Unterklassen der Schnittstelle *NamedModelElement*. Die verschiedenen durch unser Metamodell unterstützten Variablentypen sind in Abbildung 3.14 zu finden und werden im Folgenden kurz beschrieben:

Variable: Die *Variable* ist die gemeinsame Oberklasse aller Typen von Variablen, welche analog zu *Function* die wichtigsten Anfrage- und Transformationsoperationen bereits beschreibt.

Neben Klassen und Funktionen, welche das Verhalten eines Softwaresystems beschreiben, sind Variablen ein weiteres wichtiges Modellierungselement innerhalb eines Softwaresystems. Abbildung 3.14 zeigt daher die Informationen, welche die entsprechenden Modellelemente zur Verfügung stellen.

Die abstrakte Oberklasse *Variable* bietet dabei, analog zur abstrakten Oberklasse *Function*, bereits die wesentlichen Informationen bezüglich einer Variable:

Variable	
Attribute	
typeDeclaration	Ein Typzugriff, der auf den deklarierten Typ dieser Variablen hinweist.
Methoden	

getType()	Gibt den deklarierten Typ dieser Variablen wieder. Dabei sei zu beachten, dass diese Methode eine Abkürzung für *TypeAccess.getAccessedType()* ist.
isConst()	Liefert *wahr*, falls diese Variable konstant deklariert wird.

Field: Das *Field* Modellelement kapselt das aus der objektorientierten Programmierung bekannte Klassenmerkmal Attribut. Es ist somit immer Bestandteil einer Klasse. Ein Feld kann in Verbindung mit einem *Property* verwendet werden. In solchen Fällen gibt die unten angeführte Anfrage *wahr* wieder.

Field	
Methoden	
isPropertyField()	Liefert *wahr*, falls dieses Feld das Lagerungsfeld eines *Propertys* ist.

GlobalVariable: Zur Modellierung globaler Variablen, wie sie zum Beispiel in C++ oder Delphi zu finden sind, wird dieses Modellelement benutzt. Da globale Variablen nicht im Kontext eines anderen Modellelements deklariert werden, gehören sie folglich zu den bereits angesprochenen freien Strukturierungsentitäten und sind somit primär über das Modellelement *Root* zu erreichen.

GlobalVariable	
Methoden	
getSurroundingPackage()	Liefert einen Verweis auf das Paket, in welchem die globale Variable enthalten ist.

LocalVariable: Lokale Variablen sind alle Variablendeklarationen, die innerhalb einer Funktion (sofern sie nicht innerhalb einer lokalen Klassendeklaration stehen) zu finden sind. Ihre Sichtbarkeit ist somit höchstens auf die Funktion selbst beschränkt. Eine feinere Unterteilung der Sichtbarkeit (das heißt ein genaueres *Scoping*) wird in unserem Modell nicht explizit durchgeführt. Lokale Variablen sind nur dann zu bestimmen, falls für die umgebende Funktion eine Implementierung vorliegt.

LocalVariable	
Methoden	
getSurroundingFunction()	Liefert die Funktion, in deren Deklaration diese lokale Variable deklariert wurde.

FormalParameter: Hierbei handelt es sich um die Repräsentation eines formalen Parameters innerhalb einer Funktionsdeklaration. Im Allgemeinen können sie in den durch das Metamodell unterstützten Sprachen wie lokale Variablen betrachtet werden. Wie wir aber noch sehen

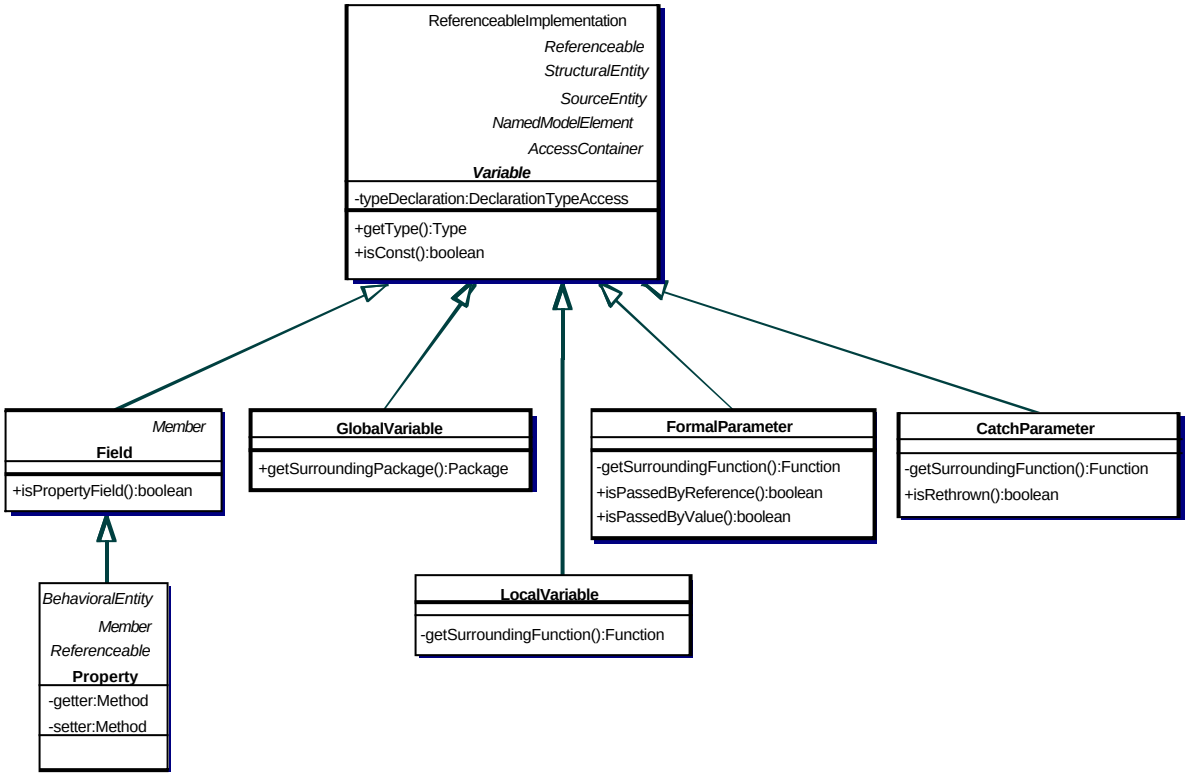

Abbildung 3.14: Hierarchie der Variablen

werden, unterscheiden sich formale Parameter und lokale Variablen in der Menge der bereitgestellten Transformationen, da nicht alle Transformationen für formale Parameter möglich sind. Weiterhin unterscheiden sich formale Parameter und lokale Variablen dahingehend, dass formale Parameter Teil der Schnittstelle sind und somit sichtbar sind, auch wenn die entsprechende Funktion keine Implementierung besitzt. Ein formaler Parameter hat folgende zusätzliche Felder und Methoden:

FormalParameter	
Methoden	
getSurroundingFunction()	Liefert die Funktion, in deren Deklaration dieser formale Parameter deklariert wurde.
isPassedByReference()	Liefert *wahr*, falls dieser Parameter als Referenz übergeben wird.
isPassedByValue()	Liefert *wahr*, falls dieser Parameter als Wert übergeben wird.

CatchParameter: Die Klasse *CatchParameter* modelliert einen Parameter in einem catch-Ausdruck. Er definiert folgende Methoden:

CatchParameter	
Methoden	
getSurroundingFunction()	Liefert die Funktion, in deren Deklaration dieser Catch-Parameter deklariert wurde.
isRethrown()	Liefert *wahr*, falls die Ausnahme, die von diesem Catch-Parameter behandelt wird, an den Aufrufer weitergeleitet wird.

Property: Die Klasse *Property* modelliert die *properties* von C#, wobei es sich im Prinzip nur um Felder mit den entsprechenden get/set-Methoden handelt. Falls auf das Feld lesend zugegriffen wird, wird die get-Methode aufgerufen, falls schreibend, dann die set-Methode. Daher definiert die Klasse die folgenden Felder:

Property	
Attribute	
getter	Die get-Methode dieser Property
setter	Die set-Methode dieser Property

Modellierung von Zugriffen

Mit Hilfe der bisher vorgestellten Modellelemente lassen sich sämtliche Strukturierungselemente der im Rahmen von QBench benutzten Programmiersprachen Java, C++, C# und Delphi abbilden. Die Kenntnis der in einem System vorhandenen Strukturen sind zwar Grundvoraussetzung für die meisten Qualitätsanalysen, -heuristiken und auch für die meisten Transformationen, doch bringt erst die Kenntnis der tatsächlichen Verbindungen zwischen den einzelnen Strukturen einen echten Mehrwert. Daher werden im vorgestellten Metamodell neben den Strukturierungselementen auch die Zugriffe auf diese Elemente modelliert. Eine Übersicht über diese Elemente ist in Abbildung 3.15 zu sehen.

Für jedes Strukturierungselement existiert ein entsprechendes Zugriffselement im Metamodell. Jeder Zugriff auf ein Strukturierungselement wird in einer Modellinstanz als Instanz eines entsprechenden Zugriffselements dargestellt. Die Charakterisierung eines Elements als Zugriffselement geschieht dabei mit Hilfe der Schnittstelle *Access*. Während die Elemente *FunctionAccess* und *VariableAccess* bezüglich ihrer Semantik keiner genaueren Beschreibung bedürfen, wird im Folgenden kurz auf das Element *TypeAccess* bzw. auf Unterklassen *DeclarationTypeAccess*, *CastTypeAccess*, *ThrowTypeAccess* und *InheritanceTypeAccess* eingegangen. Das Auftreten des Typnamens innerhalb des Quelltexts wird als *TypeAccess* modelliert, es sei denn es handelt sich dabei um den Identifizierer während der Typdefinition. Der Zugriff auf einen Konstruktor oder die Konstruktor- bzw. Destruktordefinition werden dabei nicht als *TypeAccess* betrachtet, da hierfür die spezielleren Modellelemente *FunctionAccess* bzw. *Constructor/Destructor* zur Verfügung stehen.

Das folgende Beispiel zeigt anhand einer kleinen Java Klasse, wie die Sprachelemente auf die einzelnen Metamodellelemente abgebildet werden.

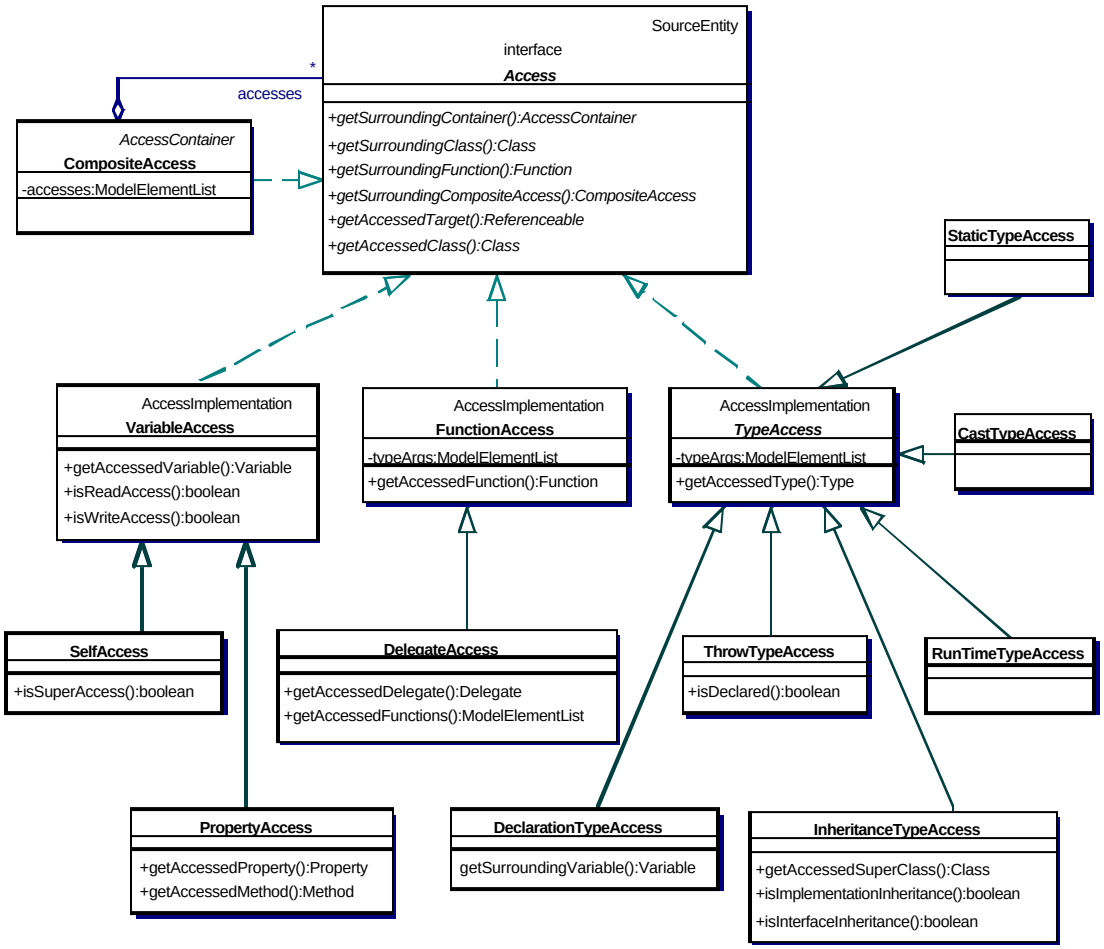

Abbildung 3.15: Die Zugriffshierarchie

Beispiel:

```
1  public class X extends Y {
2     private Z a;
3     private X b;
4
5     public X(Z p) {
6        a = p;
7     }
8
9     public void m(Z p) {
10       b = new X(p);
11    }
12 }
```

Eine Instanz der Metamodellklasse *Class* resultiert aus der Klassendefinition in Zeile 1. Die beiden Attribute in Zeile 2 und 3 werden von je einem *Field* Objekt gekapselt. Der Konstruktor

in Zeile 5 wird auf eine Instanz des Modellelements *Constructor* abgebildet. Analoges gilt für die Methode in Zeile 9 und das Element *Method*. In der vorliegenden Klasse gibt es einen Typzugriff (siehe später *InheritanceTypeAccess*): `Y` in Zeile 1. Zu beachten ist, dass die Identifizierer `X` in den Zeilen 1, 5 und 10 nicht als Typzugriffe modelliert werden. In Zeile 1 handelt es sich um den Identifizierer der Klasse während der Klassendefinition, in der Zeile 5 um den Identifizierer einer Konstruktordefinition und in Zeile 10 schließlich um einen Funktionszugriff.

Im Folgenden werden die Modellelemente, welche die Zugriffe auf die bisher beschriebenen Elemente modellieren, genauer vorgestellt. Neben der in Abbildung 3.15 vorgestellten Hierarchie der einzelnen Zugriffselemente ist im Wesentlichen noch die Information von Interesse, auf welches Element zugegriffen wird. Dies spiegelt sich auch in der Menge der bereitgestellten Informationen wieder. Eine möglichst geringe Anzahl von Merkmalen ist auch dahingehend wichtig, um den Speicherplatzverbrauch der einzelnen Instanzen gering zu halten, da in typischen Systemen eine große Anzahl dieser Instanzen vorhanden ist.

Access: Die *Access* Schnittstelle ist die Wurzel der Zugriffsunterhierarchie. Sie definiert die folgenden Methoden:

Access	
Methoden	
getAccessedTarget	Das Modellelement, auf welches zugegriffen wird. Es handelt sich hierbei um das statisch ermittelbare Modellelement, das heißt Polymorphie etc. wird nicht aufgelöst.
getAccessedClass	Die Klasse, die das zugegriffene Modellelement enthält. Wenn das zugegriffene Modellelement eine Klasse ist, diese Methode gibt die zugegriffene Klasse zurück.
getSurroundingContainer()	Liefert das *AccessContainer*-Modellelement, das diesen Zugriff enthält.
getSurroundingClass()	Falls es sich bei dem Ergebnis von *getSurroundingFunction()* um eine Methode, einen Konstruktor oder einen Destruktor handelt, so liefert diese Anfragemethode die lexikalisch umgebende Klasse, in welcher diese Funktion deklariert wurde. Ansonsten wird `null` zurückgegeben.
getSurroundingFunction()	Liefert die Funktion, in deren Rumpf dieser Zugriff steht.
getSurroundingCompositeAccess()	Liefert das *CompositAccess*-ModelElement, das diesen Zugriff enthält.

FunctionAccess: Die *Access* Schnittstelle ist durch drei Klassen implementiert: *FunctionAccess*, *VariableAccess* und *TypeAccess*. Jede dieser Klassen definiert eine Methode, die den entsprechenden Typ zurückgibt, auf den zugegriffen wird. Zum Beispiel definiert *FunctionAccess* folgende Methoden:

FunctionAccess	
Attribute	
typeArgs	Liste der tatsächlichen Typargumente, die durch den Anruf einer generischen Funktion geliefert werden.
Methoden	
getAccessedFunction()	Liefert die Funktion, auf die zugegriffen wird. Verfeinert dabei das Zugriffsziel der Oberklasse.

Wird ein Operator auf ein Objekt angewendet, so entspricht dies einem Funktionszugriff auf die entsprechende Operatorfunktion. Dementsprechend würde zum Beispiel ein "+" Operator wie ein Funktionszurgiff auf die *plus* Operatorfunktion mit der entsprechenden Signatur modelliert sein.

DelegateAccess: Ein spezieller Typ des Funktionszugriffs ist das *DelegateAccess* Modellelement. Aufgrund der Fähigkeit der Delegates als "multicast call forwarder" zu agieren kann der Zugriff auf Delegates zu verschiedenen aufgerufenen Methoden führen. Die Delegateszugriffs Klasse hat folgende Merkmale:

DelegateAccess	
Methoden	
getAccessedDelegate()	Gibt eine Referenz auf das Delegate zurück, auf das zugegriffen wird.
getAccessedFunctions()	Gibt eine Liste möglicher Funktionen wieder, die Resultate dieses Zugriffs sein könnten. Dabei sei zu beachten, dass aufgrund der Dynamik der *invocationList* eines Delegates keine exakte Methodenliste möglich ist. Eine konservative Herangehensweise wird daher bevorzugt, d.h. die zurückgegebene Liste enthält alle Funktionen, die möglicherweise aufgerufen werden.

CompositeAccess: Die Klasse *CompositeAccess* ist ein Hilfskonstrukt, das ein Argument einer Funktion modelliert.

CompositeAccess	
Attribute	
accesses	Eine geordnete Liste aller Zugriffe, die in diesem *CompositeAccess* enthalten sind.

VariableAccess: Der *VariableAccess* erweitert die *Access* Schnittstelle indem folgende Methoden hinzugefügt werden:

VariableAccess	
Methoden	
getAccessedVariable()	Liefert die Variable, auf die zugegriffen wird. Verfeinert dabei das Zugriffsziel der Oberklasse .
isReadAccess()	Prüft, ob die Variable gelesen wird.
isWriteAccess()	Prüft, ob die Variable geschrieben wird.

PropertyAccess: Wie Abbildung 3.14 zeigt, ist *Property* ein Untertyp von *Variable* und dementsprechend ist ein *PropertyAccess* ein Untertyp von *VariableAccess*. Da ein Property nicht mehr als ein Feld mit Zugriffsmethoden (get/set) ist, ist ein *PropertyAccess* ein Zugriff auf dieses Feld durch die entsprechende Zugriffsmethode (siehe *getAccessedMethod()* in der Tabelle). Daher definiert die Klasse *PropertyAccess* zwei Zugriffsziele: Eins für die Property selbst und ein weiteres für die aufgerufene Zugriffsmethode (get/set).

PropertyAccess	
Methoden	
getAccessedProperty()	Liefert die Property, auf die zugegriffenen wird, indem das Zugriffsziel ihrer Oberklasse verfeinert wird.
getAccessedMethod()	Liefert die Zugriffsmethode (get/set).

SelfAccess: Die Klasse *SelfAccess* modelliert *This*- und *Super*-Zugriffe.

SelfAccess	
Methoden	
isSuperAccess()	Lifert *wahr* wenn dieser Zugriff einen *Super*-Zugriff modelliert.

TypeAccess: Die *TypeAccess* Klasse ist die Wurzel aller Typzugriffen, von denen es mehrere Arten gibt: Typumwandlung (cast), Vererbungsbeziehung, Ausnahmedeklaration (exception), Variablentypdeklaration und Deklaration von Funktionsrückgabetypen. Die Deklaration einer Variablen und eines Rückgabetyps werden durch die *DeclarationTypeAccess* Klasse modelliert, während die anderen drei Typen ihre eigene Unterklasse haben. Das *TypeAccess* Modellelement definiert folgende Felder und Methoden:

TypeAccess	
Attribute	
typeArgs	Liste der tatsächlichen Typargumente die durch Instanziierung einer generischen Klasse geliefert werden.
Methoden	
getAccessedType()	Liefert eine Referenz auf den zugegriffenen Typ.
isCastAccess()	Gibt *wahr* zurück, falls dieser *TypeAccess* einen *cast*-Typzugriff modelliert.

CastTypeAccess: Die Klasse *CastTypeAccess* modelliert Typumwandlungen.

RunTimeTypeAccess: Die Klasse *RunTimeTypeAccess* modelliert Typidentifikationen zur Laufzeit.

DeclarationTypeAccess: Die Klasse *DeclarationTypeAccess* modelliert Variablen- und Rückgabetypen Deklarationen. Die Klasse definiert die folgenden Methoden:

DeclarationTypeAccess	
Methoden	
getSurroundingVariable()	Liefert die Variable, in deren Rahmen dieser Zugriff steht.

InheritanceTypeAccess: Der *InheritanceTypeAccess* besitzt eine Methode, die das zugegriffene Ziel weiter verfeinert:

InheritanceTypeAccess	
Methoden	
getAccessedSuperClass()	Liefert die Klasse, auf die zugegriffen wird, indem das Zugriffsziel seiner Oberklasse verfeinert wird.
isImplementationInheritance()	Liefert *wahr*, falls es sich bei dieser Vererbungsbeziehung um eine private oder protected Vererbung handelt.
isInterfaceInheritance()	Liefert *wahr*, falls es sich bei dieser Vererbungsbeziehung um public Vererbung handelt. Für mehr Informationen zu private, protected und public Vererbung siehe [ES90].

ThrowTypeAccess: Die Klasse *ThrowTypeAccess* modelliert throws-Deklarationen. Die Klasse definiert die folgenden Methoden:

ThrowTypeAccess	
Methoden	
isDeclared()	Zeigt, ob die Ausnahme in der Funktionssignatur deklariert wurde oder nicht.

StaticTypeAccess: Diese Klasse modelliert den Typzugriff auf die Klasse, die als Präfix vor einen Zugriff auf ein statisches Merkmal gesetzt wird.

3.4.3.4 Transformationsmetamodell

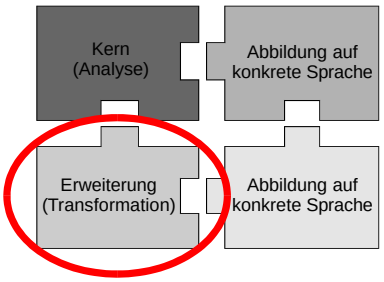

Abbildung 3.16: Einordnung Transformationssicht in den Aufbau des Modells

Dieser Abschnitt widmet sich dem sprachunabhängigen Transformationsteil des Metamodells. Wie Abbildung 3.16 zeigt, kann der Transformationsteil des Metamodells als eine Erweiterung des Analyseteils betrachtet werden. Zweck dieses sprachunabhängigen Transformationsmodells ist es, die Gemeinsamkeiten aller unterstützten Sprachen, die strukturelle Transformationen des analysierten Softwaresystems ausdrücken, abzudecken und die Auswirkungen, die diese Transformationen auf Qualitätsattribute haben, abzuschätzen. Die Verwendung eines sprachunabhängigen Transformationsmodells, das sich auf einer höheren Abstraktionsebene befindet als der abstrakte Syntaxbaum einer Sprache, hat zur Folge, dass man den Quelltext eines transformierten Systems nicht direkt restrukturieren kann. Dennoch könnte die Transformationsabfolge, die durch das abstrakte sprachunabhängige Transformationsmodell durchgeführt wurde, problemlos durch ein sprachspezifische Transformationswerkzeug, wie zum Beispiel Inject/J [GK01] durchgeführt werden. Dadurch kann die Qualität des Systems verbessert werden.

Die Erweiterung des Analyseteils des Metamodells, die oben bereits erwähnt wurde, besteht aus einer Anzahl von transformationsspezifischen Erweiterungen der Abstraktionen in Form von Zusatzfunktionen, sowie einer Anzahl neuer Abstraktionen, die das Metamodell auf Funktionsrumpf-Ebene weiter verfeinern. Die neuen Abstraktionen wurden eingeführt, um die verschiedenen Metriken auf Funktionsrumpf-Ebene nach Ausführung der Transformationen aktualisieren zu können.

Alle transfomationsspezifischen Funktionen, die in diesem Dokument dargestellt werden, sind Primitivfunktionen, die zur Konstruktion komplett rücksetzbarer Transformationen verwendet werden können, die sich durch transaktionales Verhalten auszeichnen.

Die folgenden Abschnitte stellen die Erweiterungen des Analyseteils des Metamodells dar.

Kern des Modells

Dieser Absatz erweitert den gleichnamigen des Unterabschnitt 3.4.3.3 und stellt nur die zusätzlichen Funktionen zu den bereits definierten Abstraktionen dar. Die Abstraktionen, welche keine zusätzlichen Funktionen haben, werden nicht aufgeführt.

NamedModelElement: Die Schnittstelle bietet die folgenden weiteren Operationen an:

NamedModelElement	
Methoden	
rename(String)	Ändert den einfachen Namen dieses *NamedModelElement*. Diese Methode überprüft nicht eventuell auftretende Namenskonflikte.

SourceEntity: Diese Schnittstelle bietet die folgenden Transformationsmethoden an:

SourceEntity	
Methoden	
moveToFile(File)	Diese Methode aktualisiert den entsprechenden Wert im *Position*-Objekt, abhängig davon, ob die spezifizierte Datei eine Assembly- oder eine Quelldatei ist. Falls diese *SourceEntity* ein Typ, eine globale Funktion oder eine globale Variable ist, wird die Enthaltensbeziehung in *File* wird ebenfalls aktualisiert.

GenericEntity: Diese Schnittstelle bietet die folgenden Transformationsmethoden an:

GenericEntity	
Methoden	
insertTypeParameter(TypeParameterClass, int)	Trägt einen neuen Typparameter an der spezifizierten Stelle der Typparameterliste dieser generischen Entität ein. Dieses Verfahren aktualisiert gleichzeitg auch alle Zugriffe auf diese generische Entität, indem ein extra *void* Typargument in diese eingetragen wird.
removeTypeParameter(TypeParameterClass)	Entfernt den spezifizierten Typparameter aus der Typparameterliste dieser generischen Entität. Diese Methode aktualisiert auch alle Zugriffe auf diese generische Entität, indem das entsprechende Typargument aus ihnen gelöscht wird.

Strukturierungselemente

Dieser Absatz erweitert das gleichnamige Element von Unterabschnitt 3.4.3.3 und beschreibt die zusätzlichen Funktionen, die den bereits definierten Abstraktionen beigefügt wurden. Die Abstraktionen, welche keine zusätzlichen Funktionen haben werden nicht aufgeführt.

Root: Die Klasse bietet die folgenden Transformationsmethoden an:

Root	
Methoden	
addFile(File)	Fügt dem Modell eine neue Datei hinzu.
removeFile(File)	Entfernt die spezifizierte Datei vom Modell. Das Entfernen einer Datei löscht zudem jeden *Type*, jede *GlobalFunction* oder *GlobalVariable*, die durch diese Datei definiert werden.
addPackage(Package)	Fügt dem Modell ein neues Paket hinzu.
removePackage(Package)	Entfernt das spezifizierte Paket aus dem Modell. Das Entfernen eines Pakets löscht zudem jedes *Package*, jeden *Type*, jede *GlobalFunction* oder *GlobalVariable*, die durch dieses Paket definiert wird, sowie alle Paket-imports, die sich auf das besagte Paket beziehen.

addType(Type)	Fügt dem Modell einen neuen Typ hinzu. Hier handelt sich um Typen die zu keinem Paket gehören, wie zum Beispiel primitive Typen.
removeType(Type)	Entfernt den spezifizierten Typ aus dem Modell. Das Entfernen eines Typs löscht alle Referenzen zu ihm. Hier handelt sich um Typen die zu keinem Paket gehören, wie zum Beispiel primitive Typen.

File: Der Dateityp (*sourceFile* oder *assembly*) ist unveränderlich. Diese Klasse bietet die folgenden Transformationsmethoden an:

File	
Methoden	
changePathName(String)	Ändert den Namen des Pfades, der dieser Datei zugeordnet ist.
addIncludedFile(File)	Fügt der angegebenen Datei eine Include-Beziehung hinzu.
removeIncludedFile(File)	Löscht die Include-Beziehung aus der spezifizierten Datei.
addImportedPackage(Package)	Fügt dem spezifizierten Paket eine Import-Beziehung hinzu.
removeImportedPackage(Package)	Entfernt die Import-Beziehung aus dem spezifizierten Paket.
addImportedGlobalFunction(GlobalFunction)	Fügt eine Import-Beziehung für die angegebene globale Funktion hinzu.
removeImportedGlobalFunction(GlobalFunction)	Entfernt die Import-Beziehung für die spezifizierte globale Funktion.
addImportedGlobalVariable(GlobalVariable)	Fügt eine Import-Beziehung für die spezifizierte globale Variable hinzu.
removeImportedGlobalVariable(GlobalVariable)	Entfernt die Import-Beziehung der spezifizierten globalen Variablen.
addImportedType(Type)	Fügt eine Import-Beziehung für den spezifizierten Typ hinzu.
removeImportedType(Type)	Entfernt die Import-Beziehung des spezifizierten Typs.
addGlobalFunction(GlobalFunction)	Fügt der Datei die Definition einer globalen Funktion hinzu.
removeGlobalFunction(GlobalFunction)	Entfernt die Definition einer globalen Funktion aus der Datei. Wird die Definition einer globalen Funktion aus einer Datei entfernt, so wird sie ebenso im enthaltenden Paket gelöscht. Darüberhinaus werden auch alle Referenzen sowie die Import-Beziehungen zu ihr gelöscht.
addGlobalVariable(GlobalVariable)	Fügt der Datei die Definition einer globalen Variablen hinzu.

removeGlobalVariable(GlobalVariable)	Entfernt die Definition einer globalen Variablen aus der Datei. Wird die Definition einer globalen Variablen aus einer Datei entfernt, so wird sie ebenso im enthaltenden Paket gelöscht. Darüberhinaus werden auch alle Referenzen sowie die Import-Beziehungen zu ihr gelöscht.
addType(Type)	Fügt der Datei eine Typdefinition hinzu.
removeType(Type)	Entfernt die Typdefinition aus der Datei. Wird eine Typdefinition aus einer Datei entfernt, so wird sie ebenso im enthaltenden Paket gelöscht. Darüberhinaus werden auch alle Referenzen sowie die Import-Beziehungen zu ihr gelöscht.

Package: Diese Klasse bietet die folgenden Transformationsmethoden an:

Package	
Methoden	
moveToRoot()	Macht ein Unterpaket zu einem Paket auf oberster Ebene. Auf Pakete oberster Ebene hat diese Funktion keine Auswirkungen.
moveToPackage(Package)	Verschiebt dieses Paket in das spezifizierte Paket. Falls es sich bei dem zu bewegenden Paket um ein Paket auf oberster Ebene handelt, wird es zuerst in ein Unterpaket umgewandelt.
addSubpackage(Package)	Fügt diesem Paket ein neues Unterpaket hinzu.
removeSubpackage(Package)	Entfernt das spezifizierte Unterpaket aus diesem Paket. Wird ein Unterpaket entfernt, so wird auch jedes *Package*, jeder *Type*, jede *GlobalFunction* oder *GlobalVariable*, die sowohl in dem besagten Unterpaket definiert wurden als auch die Import-Beziehungen, die sich auf das Unterpaket verweisen, gelöscht.
addGlobalFunction(GlobalFunction)	Fügt diesem Paket eine neue globale Funktion hinzu.
removeGlobalFunction(GlobalFunction)	Entfernt die spezifizierte globale Funktion aus diesem Paket. Wird eine globale Funktion aus einem Paket entfernt, so wird sie auch von der enthaltenden Datei gelöscht. Darüberhinaus werden auch alle Referenzen sowie alle Import-Beziehungen zu ihr entfernt.
addGlobalVariable(GlobalVariable)	Fügt diesem Paket eine neue globale Variable hinzu.

removeGlobalVariable(GlobalVariable)	Entfernt die spezifizierte globale Variable aus diesem Paket. Wird eine globale Variable aus einem Paket entfernt, so wird sie gleichzeitig auch aus der enthaltenden Datei gelöscht. Darüberhinaus werden alle Referenzen sowie alle Import-Beziehungen zu ihr entfernt.
addClass(Class)	Fügt diesem Paket eine neue Klasse hinzu.
removeClass(Class)	Entfernt die spezifizierte Klasse aus diesem Paket. Wird eine Klasse aus einem Paket entfernt, so wird sie gleichzeitig auch aus der enthaltenden Datei gelöscht. Darüberhinaus werden alle Referenzen sowie alle Import-Beziehungen zu ihr ebenfalls entfernt.
addDelegate(Delegate)	Fügt diesem Paket ein neues Delegate hinzu.
removeDelegate(Delegate)	Entfernt das spezifizierte Delegate aus diesem Paket. Wird ein Delegate aus einem Paket entfernt, so wird es gleichzeitig auch aus der enthaltenen Datei gelöscht. Darüberhinaus werden alle Referenzen sowie alle Import-Beziehungen zu ihm ebenfalls entfernt.
addTypeAlias(TypeAlias)	Fügt diesem Paket einen neuen Typalias hinzu.
removeTypeAlias(TypeAlias)	Entfernt den spezifizierten Typalias aus diesem Paket. Wird ein Typalias aus einem Paket entfernt, so wird er gleichzeitig auch aus der enthaltenden Datei gelöscht. Darüberhinaus werden alle Referenzen sowie alle Import-Beziehungen zu ihm ebenfalls gelöscht.

PackageAlias: Diese Klasse bietet die folgenden Transformationsmethoden an:

PackageAlias	
Methoden	
changeTargetPackage(Package)	Ändert das Ziel dieses Paketalias, indem das *aliasedPackage* Feld so aktualisiert wird, dass es auf das spezifizierte Paket zeigt.

Class: Diese Klasse bietet die folgenden Transformationsmethoden an:

Class	
Methoden	
convertToInterface()	Formt diese Klasse zu einer Schnittstelle um. Diese Transformation entfernt alle Konstruktoren und Destruktoren sowie alle Funktionsrümpfe aus der Klasse. Weitere Informationen über Funktionsrümpfe siehe Abbildung 3.17.
convertToClass()	Formt diese Schnittstelle zu einer abstrakten Klasse um. Es wird dabei nicht überprüft, ob dadurch eventuell unerlaubte Mehrfachvererbung entsteht.
moveToPackage(Package)	Verschiebt diese Klasse in das spezifizierte Paket. Falls es sich um eine innere oder eine lokale Klasse handelt, so wird diese zuerst in eine Top-level-Klasse umgewandelt. Eventuell entstehende Verletzungen von Sichtbarkeitsregeln und Namenskonflikte werden nicht berücksichtigt.
moveToClass(Class)	Verschiebt diese Klasse in die spezifizierte Klasse, wobei sie, falls dies erforderlich ist, in eine innere Klasse umwandelt wird. Eventuell entstehende Verletzungen von Sichtbarkeitsregeln und Namenskonflikte werden nicht berücksichtigt.
moveToFunction(Function)	Verschiebt diese Klasse in die spezifizierte Funktion, wobei sie, falls erforderlich, in eine lokale Klasse umwandelt wird. Eventuell entstehende Verletzungen von Sichtbarkeitsregeln und Namenskonflikte werden nicht berücksichtigt.
addInheritanceTypeAccess(InheritanceTypeAccess)	Fügt dieser Klasse einen neuen *InheritanceTypeAccess* hinzu.
removeInheritanceTypeAccess(InheritanceTypeAccess)	Entfernt den spezifizierten *InheritanceTypeAccess* aus dieser Klasse. Einige Merkmale könnten als Folge dieser Transformation aktualisiert werden. Siehe dazu auch *toggleOverride()* von *Member*.
addInnerClass(Class)	Fügt dieser Klasse eine neue innere Klasse hinzu.
removeInnerClass(Class)	Entfernt die spezifizierte innere Klasse. Wird eine innere Klasse entfernt, so werden gleichzeitig auch alle ihre Referenzen gelöscht.

addInnerDelegate(Delegate)	Fügt dieser Klasse ein neues inneres Delegate hinzu.
removeInnerDelegate(Delegate)	Entfernt das spezifizierte innere Delegate. Wird ein inneres Delegate entfernt, so werden gleichzeitig auch alle seine Referenzen gelöscht.
addInnerTypeAlias(TypeAlias)	Fügt dieser Klasse einen neuen inneren Typalias hinzu.
removeInnerTypeAlias(TypeAlias)	Entfernt den spezifizierten inneren Typalias. Wird ein innerer Typalias entfernt, so werden gleichzeitig auch alle seine Referenzen gelöscht.
addField(Field)	Fügt dieser Klasse ein neues Feld hinzu.
removeField(Field)	Entfernt das spezifizierte Feld aus dieser Klasse. Wird ein Feld entfernt, so werden gleichzeitig auch alle Referenzen auf es gelöscht.
addProperty(Property)	Fügt dieser Klasse ein neues Property hinzu.
removeProperty(Property)	Entfernt das spezifizierte Property aus dieser Klasse. Wird ein Property entfernt, so werden gleichzeitig auch alle Referenzen sowie die zugehörigen Zugriffsmethoden und Referenzen auf diese gelöscht.
addMethod(Method)	Fügt dieser Klasse eine neue Methode hinzu.
removeMethod(Method)	Entfernt die spezifizierte Methode aus dieser Klasse. Wird eine Methode entfernt, so werden gleichzeitig auch alle Referenzen auf sie gelöscht.
addConstructor(Constructor)	Fügt dieser Klasse einen neuen Konstruktor hinzu.
removeConstructor(Constructor)	Entfernt den spezifizierten Konstruktor aus dieser Klasse. Wird ein Konstruktor entfernt, so werden gleichzeitg auch alle Referenzen auf ihn gelöscht.
addDestructor(Destructor)	Fügt dieser Klasse einen neuen Destruktor hinzu.
removeDestructor(Destructor)	Entfernt den spezifizierten Destruktor aus dieser Klasse. Wird ein Destruktor entfernt, so werden gleichzeitig auch alle Referenzen auf ihn gelöscht.
addFriendClass(Class)	Fügt dieser Klasse eine Deklaration einer Friend-Klasse hinzu.

removeFriendClass(Class)	Entfernt die spezifizierte Deklaration der Friend-Klasse aus dieser Klasse. Diese Methode überprüft den potentiellen Missbrauch von Sichtbarkeiten, der durch diese Transformation hervorgerufen wird nicht.
addFriendFunction(GlobalFunction)	Fügt dieser Klasse eine Deklaration einer Friend-Funktion hinzu.
removeFriendFunction(GlobalFunction)	Entfernt die spezifierte Deklaration der Friend-Funktion aus dieser Klasse. Dieses Verfahren überprüft den potentiellen Missbrauch von Sichtbarkeiten, der durch diese Transformation hervorgerufen wird nicht.

Member: Diese Schnittstelle bietet die folgenden Transformationsmethoden an:

Member	
Methoden	
changeAccessSpecifier(int)	Ändert die Sichtbarkeit dieses Merkmals. Dieses Verfahren überprüft den Verstoß gegen Sichtbarkeitsregeln nicht.
toggleAbstract()	Schaltet das *abstract*-Flag dieses Merkmals um.
toggleFinal()	Schaltet das *final*-Flag dieses Merkmals um.
toggleStatic()	Schaltet das *static*-Flag dieses Merkmals um.
toggleExtern()	Schaltet das *extern*-Flag dieses Merkmals um.
toggleVirtual()	Schaltet das *virtual*-Flag dieses Merkmals um.
toggleNew()	Schaltet das *new*-Flag dieses Merkmals um.
toggleOverride()	Schaltet das *override*-Flag dieses Merkmals um.
toggleIntrospectable()	Schaltet das *introspectable*-Flag dieses Merkmals um.
moveToClass(Class)	Verschiebt dieses Merkmal in die spezifizierte Klasse. Dieses Verfahren überprüft keine Sichtbarkeiten und Namenskonflikte.

Typhierarchie

Dieser Absatz erweitert den gleichnamigen des Unterabschnitts 3.4.3.3 und beschreibt nur die zusätzlichen Funktionen zu denen, die bereits definiert wurden. Die Abstraktionen, die keine zusätzlichen Funktionen haben werden nicht erläutert.

Type: Diese Schnittstelle bietet die folgenden Transformationsmethoden an:

Type	
Methoden	
toggleReferenceType()	Ändert die vorgegebene Speicherverwaltung dieses Typs. Das Verfahren schaltet zwischen *reference* und *value* um.

GenericClass: Diese Klasse bietet die folgenden Transformationsmethoden an:

GenericClass	
Methoden	
insertTypeParameter(TypeParameterClass, int)	Trägt einen neuen Typparameter an der spezifizierten Stelle der Typparameterliste dieser generischen Klasse ein. Dieses Verfahren aktualisiert gleichzeitg auch alle Typzugriffe auf diese generische Klasse, indem ein extra *void* Typargument in diese eingetragen wird.
removeTypeParameter(TypeParameterClass)	Entfernt den spezifizierten Typparameter aus der Typparameterliste dieser generischen Klasse. Diese Methode aktualisiert auch alle Typzugriffe auf diese generische Klasse, indem das entsprechende Typargument aus ihnen gelöscht wird.

Eine nicht-generische Klasse in eine generische Klasse umzuwandeln ist eine komplexere Transformation, die anhand folgender Primitivfunktionen implementiert werden kann.

TypeAlias: Diese Klasse bietet die folgenden Transformationsmethoden an:

TypeAlias	
Methoden	
changeTargetType(Type)	Ändert das Ziel dieses Typalias. Das heißt, sie aktualisiert das *targetType*-Feld, um auf den spezifizierten Typ zu zeigen.
moveToPackage(Package)	Verschiebt diesen Typalias in das spezifizierte Paket. Falls es sich um einen inneren Typalias handelt, so wird dieses zuerst auf den Status eines Top-level-Typalias befördert. Diese Transformation überprüft den Verstoß gegen Sichtbarkeitsregeln und Namenskonflikte nicht, die durch diese Transformation verursacht werden könnten.
moveToClass(Class)	Verschiebt diesen Typalias in die spezifizierte Klasse, wobei er in einen inneren Typalias umgeformt wird, falls dies erforderlich ist. Dieses Transformation überprüft den Verstoß gegen Sichtbarkeitsregeln und Namenskonflikte nicht, die durch diese Transformation verursacht werden könnten.

Delegate: Das *SuperClass*-Feld dieser Klasse ist unveränderlich, da alle Delegates eine feste, gemeinsame Oberklasse besitzen. Diese Klasse bietet die folgenden Transformationsmethoden an:

Delegate	
Methoden	
moveToPackage(Package)	Verschiebt dieses Delegate in das angegebene Paket. Falls es sich dabei um ein inneres Delegate handelt, so wird dieser zuerst auf den Status eines Top-level-Delegates befördert. Dieses Transformation überprüft den Verstoß gegen Sichtbarkeitsregeln und Namenskonflikte nicht, die durch diese Transformation verursacht werden könnten.
moveToClass(Class)	Verschiebt dieses Delegate in die angegebene Klasse, wobei es in ein inneres Delegate umgewandelt wird, falls dies erforderlich ist. Dieses Transformation überprüft den Verstoß gegen Sichtbarkeitsregeln und Namenskonflikte nicht, die durch diese Transformation verursacht werden könnten.
addInvokedFunction(Function)	Fügt der Aufrufliste dieses Delegates die spezifizierte Funktion hinzu. Von dieser Transformation wird angenommen, dass sie korrekt ist. Deshalb wird keine Prüfung der Signaturen durchgeführt.
removeInvokedFunction(Function)	Entfernt die spezifizierte Funktion aus der Aufrufliste dieses Delegates.

Reference: Das *TargetType*-Feld dieser Klasse ist unveränderlich. Es ist ein semantischer Fehler, den Basistyp einer Referenz zu verändern. Um denselben Effekt zu erzielen könnte eine neue Referenz geschaffen werden, während die alte entfernt wird. Diese Klasse bietet die folgenden Transformationsmethoden an:

Reference	
Methoden	
toggleExplicit()	Dieses Verfahren schaltet zwischen einer *Explicit*- und *Implicit*-Referenz um.

Array: Das *TargetType*-Feld dieser Klasse ist unveränderlich. Es ist ein semantischen Fehler, den Basistyp eines Arrays zu verändern. Um denselben Effekt zu erzielen, könnte ein neues Array geschaffen werden, während das alte entfernt wird. Dasselbe gilt für die Anzahl von Dimensionen eines Arrays.

Hierarchie der Variablen

Dieses Kapitel ergänzt das gleichnamige des Unterabschnitts 3.4.3.3 und zeigt nur die zusätzlichen Funktionen. Die Modellelemente, zu denen keine zusätzlichen Funktionen vorliegen werden nicht aufgeführt.

Variable: Diese Klasse bietet die folgenden Transformationsmethoden an:

Variable	
Methoden	
changeType(DeclarationTypeAccess)	Ändert den deklarierten Typ dieser Variablen. Der alte *DeclarationTypeAccess* wird dabei entfernt. Bei den Zugriffen auf diese Variable wird keine Überprüfung der Typkonsistenz wird vorgenommen.
toggleConst()	Schaltet das *Const*-Flag dieser Variablen um.

Property: Diese Klasse bietet die folgenden Transformationsmethoden an:

Property	
Methoden	
changeGetter(Method)	Ändert die Get-Methode, die mit diesem Property verbunden ist. Die alte Get-Methode wird dabei entfernt.
changeSetter(Method)	Ändert die Set-Methode, die mit diesem Property verbunden ist. Die alte Set-Methode wird dabei entfernt.

GlobalVariable: Diese Klasse bietet die folgenden Transformationsmethoden an:

GlobalVariable	
Methoden	
moveToPackage(Package)	Verschiebt diese globale Variable in das spezifizierte Paket. Dieses Transformation überprüft den Verstoß gegen Sichtbarkeitsregeln und Namenskonflikte nicht, die durch diese Transformation verursacht werden könnten.

LocalVariable: Diese Klasse bietet die folgenden Transformationsmethoden an:

LocalVariable	
Methoden	
moveToFunction(Function)	Verschiebt diese lokale Variable in die spezifizierte Funktion. Dieses Transformation überprüft den Verstoß gegen Sichtbarkeitsregeln und Namenskonflikte nicht, die durch diese Transformation verursacht werden könnten.

FormalParameter: Diese Klasse bietet die folgenden Transformationsmethoden an:

FormalParameter
Methoden

togglePassedByReference()	Ändert die Übergabesemantik dieses formalen Parameters. Die Methode schaltet zwischen *passedByReference* und *passedByValue* um.

CatchParameter: Diese Klasse beinhaltet folgende zusätzlichen Merkmale:

CatchParameter	
Attribute	
catchBlock	Der eigentliche *CatchBlock* zu dem dieser *CatchParameter* gehört. Siehe dazu Abbildung 3.18.

Den Typ einer Variablen umzuwandeln (LocalVariable zu Field, GlobalVariable zu Field, FormalParameter zu Field, etc.) erfordert komplexere Transformationen, die durch die Primitivfunktionen, die in diesem Unterabschnitt dargestellt werden, implementiert werden können. Abschnitt 4.2 zeigt ein Beispiel einer solchen Umwandlung.

Hierarchie der Funktionen

Dieses Kapitel ergänzt das gleichnamige des Abschnitts 3.4.3.3 und stellt die zusätzlichen Funktionen zu den bereits definierten Abstraktionen dar, sowie die Änderungen in der *Function*-Klasse. Die Abstraktionen die nicht geändert wurden werden nicht aufgeführt. Abbildung 3.17 zeigt die wichtigsten Erweiterungen des Analyseteils des Metamodells, nämlich die Einführung von Anweisungen als Teil eines neu hinzugefügten Funktionsrumpfes.

Function: Diese Klasse beinhaltet folgende zusätzliche Merkmale:

Function	
Attribute	
body	Ein *BlockStatement*, das den Rumpf dieser Funktion representiert.
Methoden	
convertToOperator()	Wandelt diese Funktion in einen Operator um. Diese Methode überprüft nicht, ob der Funktionsname einer Operator-Definition entspricht.
convertToFunction()	Falls diese Funktion eine Operator darstellt, so wandelt ihn diese Methode in eine Funktion um. Dieses Verfahren überprüft nicht, ob der Funktionsname der Operator-Definition entspricht.
changeReturnType(DeclarationTypeAccess)	Verändert den Rückgabetyp dieser Funktion. Diese Methode überprüft Typkonflikte, die durch diese Transformation auftreten können, nicht. Der alte *DeclarationTypeAccess* wird dabei entfernt.

insertFormalParameter(FormalParameter, int)	Trägt einen formalen Parameter an der spezifizierten Stelle in der Signatur dieser Funktion ein.
removeFormalParameter(FormalParameter)	Entfernt den spezifizierten formalen Parameter aus der Signatur dieser Funktion.
addThrowTypeAccess(ThrowTypeAccess)	Fügt der Signatur dieser Funktion eine neue Ausnahmevereinbarung hinzu.
removeThrowTypeAccess(ThrowTypeAccess)	Entfernt die spezifizierte Ausnahmevereinbarung aus der Signatur dieser Funktion.
addLocalClass(Class)	Fügt dieser Funktion eine neue lokale Klasse hinzu.
removeLocalClass(Class)	Entfernt die spezifizierte lokale Klasse aus dieser Funktion.
addLocalVariable(LocalVariable)	Fügt dieser Funktion eine neue lokale Variable hinzu.
removeLocalVariable(LocalVariable)	Entfernt die spezifizierte lokale Variable aus dieser Funktion.
changeBody(BlockStatement)	Ändert den Rumpf dieser Funktion. Der alte Rumpf wird dabei entfernt.

GlobalFunction: Diese Klasse bietet die folgenden Transformationsmethoden an:

GlobalFunction	
Methoden	
convertToUnitInitializer()	Wandelt diese globale Funktion in einen Unit-Initialisierer um. Dieser Vorgang löscht die Signatur der Funktion.
convertToUnitFinalizer()	Wandelt diese globale Funktion in einen Unit-Finalisierer um. Dieser Vorgang löscht die Signatur der Funktion.
convertToGlobalFunction()	Falls diese globale Funktion für einen Paketinitializer steht, so wandelt diese Methode ihn in eine globale Funktion um. Die globale Funktion wird “init” genannt und wird keine Parameter erhalten und “void” zurückgeben.
moveToPackage(Package)	Verschiebt diese globale Funktion in das spezifizierte Paket. Diese Transformation überprüft den Verstoß gegen Sichtbarkeitsregeln und Namenskonflikte nicht, die durch diese Transformation verursacht werden könnten.

Constructor: Diese Klasse bietet die folgenden Transformationsmethoden an:

Constructor	
Methoden	
convertToInitializer()	Wandelt diesen Konstruktor in einen Initializer um. Dieser Vorgang löscht die Signatur des Konstruktors.

convertToConstructor()	Falls dies einen Initializer darstellt, so wandelt diese Methode ihn zu einem Konstruktor um. Eine no-args-Signaturvorgabe wird auf den neuen Konstruktor übertragen. Diese Methode überprüft nicht mögliche Konflikte, die durch die Transformation auftreten können.

Method: Diese Klasse bietet die folgenden Transformationsmethoden an:

Method	
Methoden	
convertToClassMethod()	Falls dies eine Property-Zugriffsmethode darstellt, so wandelt diese Methode sie in eine Klassenmethode um. Der Name der daraus resultierenden Methode wird aus dem Prefix *get-* oder *set-* und dem großgeschriebenen Namen des Propertys zusammengesetzt.

Eine Methode in eine andere Klasse zu verschieben ist eine komplexe Transformation, die mit den in diesem Abschnit beschriebenen Primitivtransformationen implementiert werden kann. Wie bereits erwähnt, verschiebt die von *Member* geerbte *moveToClass(Class)*-Methode die Methode ohne Verletzungen von Sichtbarkeiten zu berücksichtigen. Die vollständige Transformation ist beispielhaft in Abschnitt 4.2 beschrieben.

GenericFunction: Diese Klasse bietet die folgenden Transformationsmethoden an:

GenericFunction	
Methoden	
insertTypeParameter(TypeParameterClass, int)	Trägt einen neuen Typparameter an der spezifizierten Stelle in die Typparameterliste dieser generischen Funtkion ein. Diese Methode aktualisiert auch alle Funktionszugriffe auf diese generische Funktion, indem sie ein zusätzliches *Void*-Typargument einführt.
removeTypeParameter(TypeParameterClass)	Entfernt den spezifizierten Typparameter aus der Liste der Typparameter dieser generischen Funktion. Diese Methode aktualisiert auch alle Funktionszugriffe auf diese generische Funktion, indem das entsprechende Typargument von ihnen gelöscht wird.

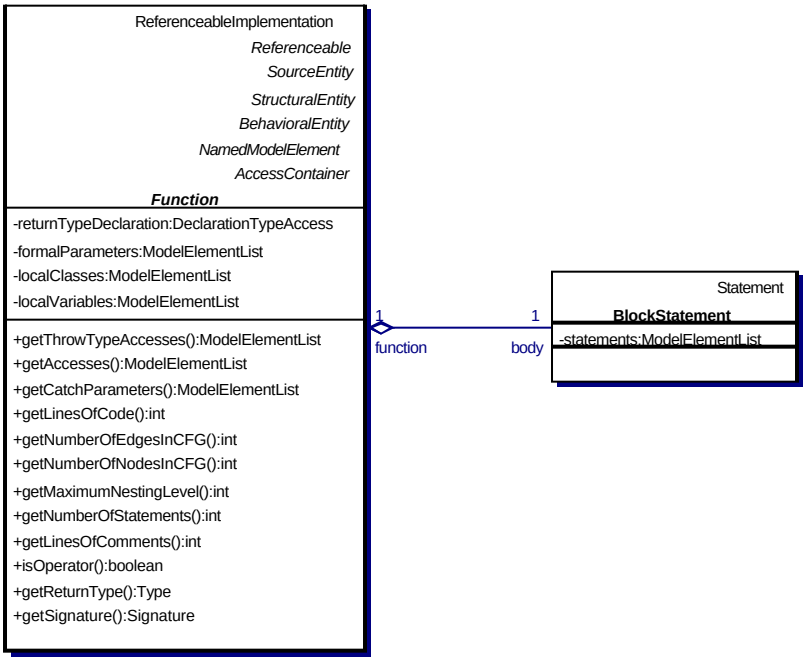

Abbildung 3.17: Funktionsrümpfe

Zwischen verschiedenen Arten von Funktionen umzuschalten (GlobalFunction zu Method, GlobalFunction zu GenericFunction, etc.) erfordert komplexere Transformationen, die durch die Primitivfunktionen, die hier aufgeführt werden implementiert werden können.

Hierarchie der Anweisungen

Wie Abbildung 3.17 zeigt, wurde der Analyseteil des Metamodells durch einen zusätzlichen Funktionsrumpf mit Anweisungen weiter verfeinert. Die Anweisungshierarchie wird in Abbildung 3.18 dargestellt. Zugriffe aus dem Funktionsrumpf heraus sind nun nicht mehr direkt mit den Funktionen verbunden, sondern mit den Anweisungen, die sie enthalten.

Statement: Die *Statement*-Klasse ist die Wurzel der Anweisungshierarchie und stellt eine Anweisung, die in einem Funktionsrumpf auftreten kann dar. Nicht-lokale Klassendeklarationen, Imports und andere OO-Sprachanweisungen, die nicht in einem Funktionsrumpf autreten können, werden nicht durch diese Klasse modelliert. Zu beachten ist, dass *Statement SourceEntity* implementiert und folglich jedes *Statement* eine spezifizierte Position durch ein *Position*-Objekt hat. *Statement* definiert folgende Merkmale:

Statement	
Attribute	
accesses	Eine geordnete Liste aller Zugriffe, die in dieser Anweisung enthalten sind.
container	Eine Referenz auf die enthaltende Anweisung oder auf *null*, falls diese Anweisung ein *BlockStatement* ist, das einen Funktionsrumpf darstellt.
Methoden	

getSurroundingFunction()	Gibt die Funktion wieder, die diese Anweisung enthält.
getLinesOfCode()	Liefert die Anzahl der Codezeilen die diese Anweisung umfasst.
getNumberOfEdgesInCFG()	Gibt die Anzahl der Kanten im Kontrollflussgraphen dieser Anweisung wieder.
getNumberOfNodesInCFG()	Liefert die Anzahl der Knoten im Kontrollflussgraphen dieser Anweisung.
getMaximumNestingLevel()	Gibt die Maximale Schachtelungstiefe der Kontrollstrukturen in dieser Anweisung wieder.
getNumberOfStatements()	Falls es sich hierbei nicht um ein *SimpleStatement* handelt, liefert diese Methode die Zahl der Anweisungen in dieser Anweisung.
getNumberOfComments()	Gibt die Anzahl der Kommentare, die mit dieser Anweisung verbunden sind wieder.
insertAccess(Access, int)	Trägt einen neuen Zugriff an der spezifizierten Stelle im Zugriffspfad dieser Anweisung ein.
removeAccess(Access)	Entfernt den spezifizierten Zugriff aus dem Zugriffspfad dieser Anweisung.

SimpleStatement: Die *SimpleStatement* Klasse modelliert alle Anweisungen, deren Kontrollfluss eine lineare Sequenz ist. Typisches Beispiel hierfür sind Zuweisungen.

JumpStatement: Diese Klasse modelliert unbedingte Sprünge jeder Art. Zu beachten ist dabei, dass das Auslösen von Ausnahmen und Return-Anweisungen ebenfalls als unbedingte Sprünge aufgefasst werden können. Diese Klasse definiert die folgende Merkmale:

JumpStatement	
Methoden	
isJump()	Liefert *true*, falls diese Anweisung ein lokaler Sprung ist, also ein Sprung, der innerhalb des selben Funktionsrumpf ist. Typische Beispiele hierfür sind: *break*, *continue*, *goto*, etc.
isThrow()	Liefert *true*, falls diese Anweisung das Auslösen einer Ausnahme darstellt.
isReturn()	Gibt *true* wieder, falls diese Anweisung eine return-Anweisung darstellt.

LoopStatement: Die *LoopStatement* Klasse modelliert jede Art von Loop-Anweisungen, die in OO-Programmiersprachen anzutreffen ist, wie zum Beispiel *for*, *while*, *do-while* or *foreach*. Die Klasse definiert die folgenden Merkmale:

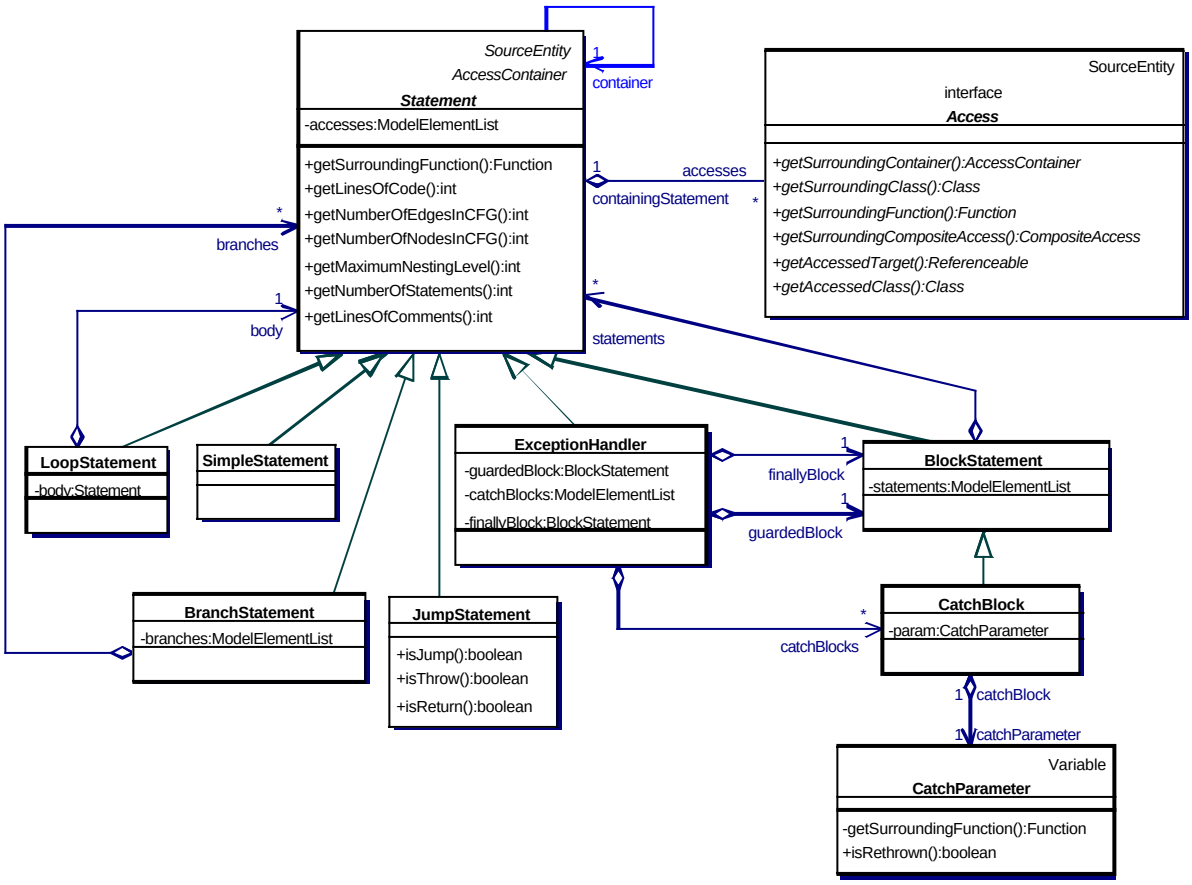

Abbildung 3.18: Statements

LoopStatement	
Attribute	
body	Rumpf der Schleife.
Methoden	
changeBody(Statement)	Ändert den Rumpf dieser Schleife. Der alte Schleifenrumpf wird dabei entfernt.

BranchStatement: Diese Klasse modelliert Verzweigungen, auch Auswahlanweisung oder auch bedingte Anweisung genannt. Typische Beispiele sind die *if* Anweisungen oder die *switch* Anweisungen. Die Klasse definiert die folgenden Merkmale:

BranchStatement	
Attribute	
branches	Eine geordnete Liste der Verzweigungen, die zu diesem *BranchStatement* gehören.
Methoden	
insertBranch(Statement, int)	Trägt einen neuen Zweig in die Verwzeigungsliste an der spezifizierten Stelle ein.
removeBranch(Statement)	Entfernt den spezifizierten Zweig dieses *BranchStatement*. Die Anweisung, die die Verzweigung repräsentiert, wird ebenfalls gelöscht.

BlockStatement: Die *BlockStatement*-Klasse modelliert zusammengesetzte Anweisungen sowie Funktionsrümpfe. Sie definiert die folgende Merkmaler:

BlockStatement	
Attribute	
statements	Eine geordnete Liste der Anweisungen, die in dieser zusammengesetzten Anweisung enthalten sind.
function	Falls dieses *BlockStatement* einen Funktionsrumpf darstellt, bezieht sich dieses Feld auf die umfassende Funktion.
Methoden	
insertStatement(Statement, int)	Trägt eine neue Anweisung in die Anweisungsliste an der spezifizierten Stelle ein.
removeStatement(Statement)	Entfernt die spezifizierte Anweisung. Wird eine Anweisung von einem *BlockStatement* entfernt, so wird dabei auch die Anweisung selbst gelöscht.

ExceptionHandler: Diese Klasse stellt einen Ausnahmebehandlung dar. Eine typische Ausnahmebehandlung enthält einen *guarded block*, auch *try block* genannt und mehrere *catch blocks*. Der *finally block* kann optional auch enthalten sein:

ExceptionHandler	
Attribute	
guardedBlock	Der Guarded-Teil enthält diese Ausnahmebehandlung.
catchBlocks	Eine Liste von Catch-Blocks.
finallyBlock	Der optionale Finally-Block.
Methoden	
changeGuardedBlock(BlockStatements)	Ändert den Guarded-Block dieser Ausnahmebehandlung.
addCatchBlock(CatchBlock)	Fügt dieser Ausnahmebehandlung einen neuen Catch-Block hinzu.

removeCatchBlock(CatchBlock)	Entfernt den spezifizierten Catch-Block. Diese Methode überprüft keine möglicherweise entstehenden unbehandelten Ausnahmen, die durch diese Transformation auftreten können.
changeFinallyBlock(BlockStatements)	Ändert den Finally-Block dieser Ausnahmebehandlung.

CatchBlock: Diese Klasse repräsentiert einen Catch-Block. Sie erweitert die *BlockStatement*-Klasse und fügt folgende Merkmale hinzu:

CatchBlock	
Attribute	
param	Die Catch-Parameter dieses Catch-Blocks.
Methoden	
changeCatchParameter(CatchParameter)	Ändert die *CatchParameter* dieses Catch-Blocks. Die alten *CatchParameter* werden dabei entfernt.

Modellierung von Zugriffen

Dieses Kapitel erweitert das gleichnamige des Unterabschnitts 3.4.3.3 und zeigt nur die zusätzlichen Funktionen, die den bereits definierten Abstraktionen hinzugefügt werden. Die Abstraktionen, die keine zusätzlichen Funktionen aufweisen werden nicht aufgeführt.

Access: Diese Schnittstelle beinhaltet folgende zusätzlichen Methoden:

Access	
Methoden	
getSurroundingStatement()	Liefert eine Referenz auf die Anweisung, die diesen Zugriff beinhaltet.
changeTarget(Referenceable)	Ändert das Ziel dieses Zugriff. Das neue Ziel muss mit dem Zugriff kompatibel sein. Zum Beispiel kann man das Ziel eines *FunctionAccess* nicht ändern um auf einen Typ zu zeigen. Neben diesen kleinen Kompatibilitätskontrollen überprüft diese Methode keine Widersprüchlichkeiten, die durch diese Transformation verursacht werden könnten.

FunctionAccess: Diese Klasse bietet die folgenden Transformationsmethoden an:

FunctionAccess
Methoden

insertTypeArgument(Type, int)	Trägt ein neues Typargument an der spezifizierten Stelle in der Liste der Typargumente dieses Funktionszugriffs ein.
removeTypeArgument(int)	Entfernt das spezifische Typargument aus der Typargumentliste dieses Funktionszugriffs.

CompositeAccess: Diese Klasse bietet die folgenden Transformationsmethoden an:

CompositeAccess	
Methoden	
insertAccess(Access, int)	Trägt einen neuen Zugriff an der spezifizierten Stelle im Zugriffspfad dieses *CompositeAccess* ein.
removeAccess(Access)	Entfernt den spezifizierten Zugriff aus dem Zugriffspfad dieses *CompositeAccess*.

VariableAccess: Diese Klasse bietet die folgenden Transformationsmethoden an:

VariableAccess	
Methoden	
toggleWriteAccess()	Schaltet den Typ des Zugriffs dieses *VariableAccess* zwischen *read access* und *write access* um.

SelfAccess: Diese Klasse bietet die folgenden Transformationsmethoden an:

SelfAccess	
Methoden	
toggleSuperAccess()	Schaltet den Typ des Zugriffs dieses *SelfAccess* zwischen *this access* und *super access* um.

TypeAccess: Diese Klasse bietet die folgenden Transformationsmethoden an:

TypeAccess	
Methoden	
insertTypeArgument(Type, int)	Trägt ein neues Typargument an der spezifizierten Stelle der Liste der Typargumente dieses Typzugriffs ein.
removeTypeArgument(int)	Entfernt das spezifizierte Typargument aus der Liste der Typargumente dieses Typzugriffs.

InheritanceTypeAccess: Diese Klasse bietet die folgenden Transformationsmethoden an:

InheritanceTypeAccess	
Methoden	
toggleImplementationInheritance()	Schaltet den Vererbungstyp, der durch diesen *InheritanceTypeAccess* ausgedrückt wird, zwischen *implementation inheritance* und *interface inheritance* um.

Code-Duplikate

Durch die Identifizierung von Codeclones können wertvolle Informationen für die Qualitätsbeurteilungsphase gewonnen werden, außerdem erhält man grundlegende Informationen zur Qualitätsverbesserung. Leider arbeiten Instrumente zur Cloneerkennung momentan noch auf Quellcodeebene. Daher erscheinen Clones als Regionen von Zeilennummern. Problematisch an dieser Darstellung ist, dass es unmöglich ist Informationen über Codeclones zu aktualisieren, nachdem Transformationen am Modell vorgenommen wurden, da die Position des verschobenen ModelElements im Quelltext ungültig wird. Um dieses Problem zu lösen,haben wir die Regionen der Zeilenummern auf Anweisungsfolgen, die durch solche Transformationen nicht betroffen werden, abgebildet. Abbildung 3.19 zeigt die Abstraktionen, die verwendet werden, um Clones darzustellen. Dabei sei zu beachten, dass die Darstellung von Clones als Anweisungsfolgen nicht erlaubt, Clones darzustellen, die die Grenzen des Funktionsrumpf überschreiten. In solchen Fällen wird der Clone in mehrere unterschiedliche Clones aufgeteilt, wobei jeder einzelne die Duplikation des Inhaltes eines einzelnen Funktionsrumpf ist. Dies ist keine Einschränkung unserer Darstellung, da Schnittstellenduplikation durch den Listenvergleich von Merkmalern zweier Klassen identifiziert werden können. Darüberhinaus beseitigt unsere Darstellung einen gewissen Grad an Beliebigkeit bei der Codeduplikationsanalyse, die durch neu geordnete Merkmale verursacht wurde. Siehe dazu das untere Beispiel:

Beispiel:

```
1  public class A {
2     public void swap(int a, int b) {
3        int tmp;
4        tmp = a;
5        a = b;
6        b = tmp;
7     }
8
9     public int max(int a, int b) {
10       if (a > b)
11          return a;
12       else
13          return b;
14    }
15 }
16
```

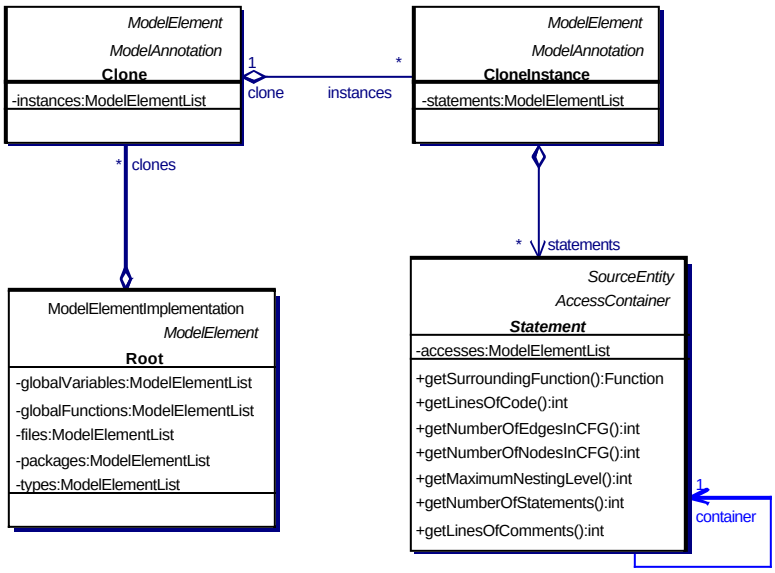

Abbildung 3.19: Clones

```
17 public class B {
18    public void swap(int a, int b) {
19       int tmp;
20       tmp = a;
21       a = b;
22       b = tmp;
23    }
24
25    public int max(int a, int b) {
26       if (a > b)
27          return a;
28       else
29          return b;
30    }
31 }
```

Zwischen Klasse A und B liegt offensichtlich eine Codeduplikation vor. Eine textorientierte Cloneerfassung würde einen Clone mit zwei Instanzen entdecken: Die Zeilen 2 bis 15 und die Zeilen 18 bis 31. Zu beachten sei nun, dass Klasse B die selben Methoden nur in umgekehrter Reihenfolge hat. In diesem Fall würde eine textorientierte Cloneerfassung zwei Clones entdecken, wobei jeder zwei Instanzen aufweist. Der erste Clone wäre die *Swap*-Methode und hätte zwei Instanzen: Die Zeilen 2 bis 7 und die Zeilen 25 bis 30. Der zweite Clone wäre die *Max*-Methode und würde ebenfalls zwei Instanzen aufweisen: Die Zeilen 9 bis 14 und die Zeilen 18 bis 23. Sogar wenn Whitespaces und Rauschen, die durch Klammern (}){dargestellt werden ignoriert werden, so ist es immer noch schwierig, basierend auf den Informationen, die durch den Cloneerkenner gewonnen werden, zu sagen, dass die beiden Situationen identisch sind. Bei unserer Darstellung würden die beiden Situationen, da wir Clones als Anweisungsfolgen darstellen, als identisch erscheinen.

Clone: Diese Klasse stellt eine geclonte Anweisungsfolge dar. Zu beachten sei dabei, dass sie *ModelAnnotation* implementiert, so dass sie an verschiedene *ModelElements* angehängt werden kann. Alle *Clone* Objekte sind dem *Root* Objekt angefügt. Die Klasse definiert folgende Merkmale:

Clone	
Attribute	
instaces	Die Liste der Cloneinstanzen.

CloneInstance: Die *CloneInstance* Klasse stellt eine Cloneinstanz dar. Sie besitzt sowohl eine Referenz auf den Clone, den sie beinhaltet, als auch eine Referenz auf die tatsächlichen Anweisungen, die zu diesem Beispiel gehören. *CloneInstances* sind *ModelAnnotations*, aber sie sind an die Funktionen, die die Anweisungen der *CloneInstance* enthält, angehängt.

CloneInstance	
Attribute	
clone	Der Clone, zu dem diese Cloneinstanz gehört.
statements	Die Liste der duplizierten Anweisungen, die zu dieser Cloneinstanz gehört.

Strukturelle Abstraktionen

Um die Erzeugung von höherwertigen strukturellen Abstraktionen wie *Beans*, *Komponenten* oder *Subsystemen* zu erlauben, haben wir einen einfachen Gruppierungsmechanismus eingeführt, der in Abbildung 3.20 zu sehen ist.

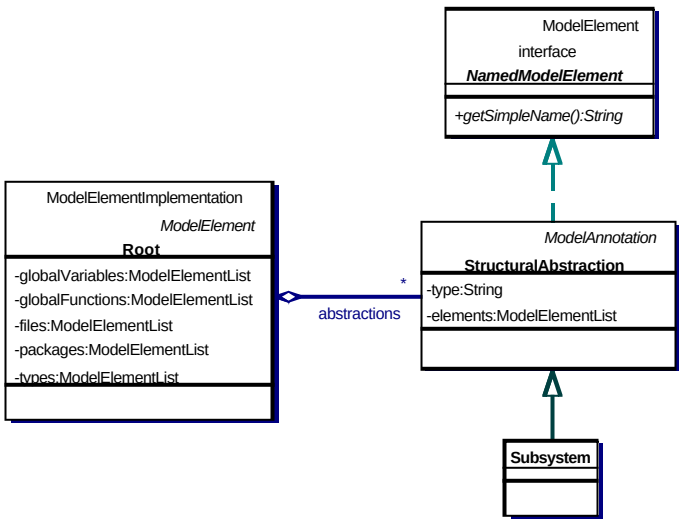

Abbildung 3.20: Strukturelle Abstraktionen

StructuralAbstraction: Diese Klasse dient als Oberklasse für alle strukturellen Abstraktionen, die auf höheren Schichten definiert werden. Diese Klasse implementiert *ModelAnnotation*, so dass sie jedes Modellelement annotieren kann. Alle *StructuralAbstraction* Objekte sind dem *Root* Objekt angefügt. Die Klasse definiert die folgenden Merkmale:

StructuralAbstraction	
Attribute	
type	Ein Typname für den Typ dieser strukturellen Abstraktion.
elements	Eine Liste von Modellelementen, die von dieser strukturellen Abstraktion zusammengefasst werden.

Subsystem: Diese Klasse modelliert Subsysteme. Für Subsysteme der Typname der strukturellen Abstraktion ist immer “Subsystem”.

Kapitel 4

Beispiele

Dieses Kapitel beschreibt Beispiele für die Anwendung des Analyse- und Transformationsmodells, das im Abschnitt 3.4 dargestellt wurde. Es besteht aus zwei Abschnitten. Der erste zeigt drei Beispiele komplexerer Anfragen, die durch die Primitivfunktionen des Analysemodells konstruiert werden können, während der andere Teil drei Beispiele komplexer Transformationen, die durch das Transformationsmodell definiert werden können aufführt. Zur Darstellung der Beipiele wurden UML Sequenzdiagramme gewählt, um die Interaktionen und die Navigation zwischen den Objekten des Modells darzustellen.

4.1 Anfragen

Dieser Abschnitt zeigt drei Beipiele komplexerer Abfragen, die auf dem Analysemodell implementiert wurden und verwendet werden, um drei der Problemmuster, die im QBench Problemmuster-Katalog aufgeführt werden zu erfassen. Die folgenden Unterabschnitte beschreiben die

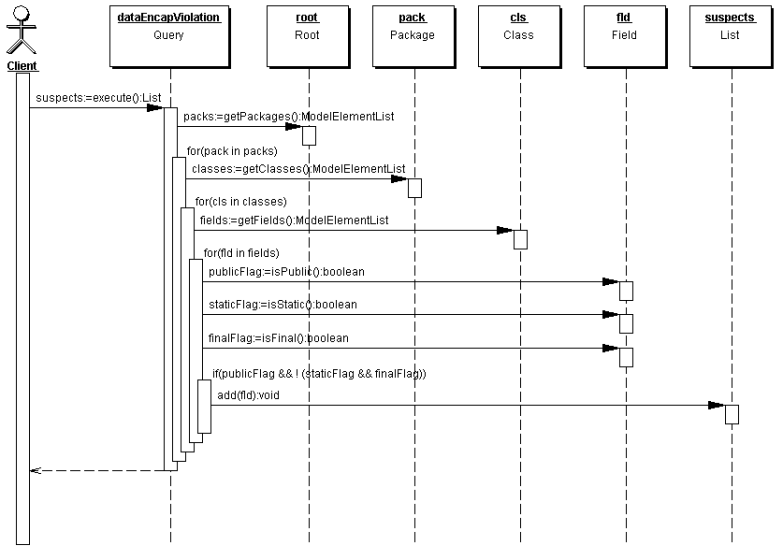

Abbildung 4.1: Datenkapselaufbruch

Suche nach den bereits oben erwähnten Problemmustern.

4.1.1 Datenkapselaufbruch

Dieses Problemmuster involviert die Verwendung von öffentlichen Feldern in OO-Quelltext. Abbildung 4.1 zeigt eine Möglichkeit, wie dieses Problemmuster erkannt werden kann. Die Implementierung der Abfrage, die nach diesem Muster sucht, iteriert über alle Pakete, die im *Root*-Element dieses Modells enthalten sind. Für jedes *Package* wird über die Liste der dort vorhandenen Klassen und der in diesen Klassen vorhandenen Felder iteriert. Jedes öffentliche *Field*, außer solchen die sowohl statisch als auch final sind, wird als verdächtig angesehen und demnach in die Liste der Verdächtigen aufgenommen. Diese Liste wird von der Anfrage zurückgegeben.

4.1.2 Tote Attribute

Dieses Problemmuster betrachtet die Existenz nicht genutzter Felder innerhalb einer Klasse. Ein nicht genutztes Feld ist ein Feld, auf das es keinen Zugriff gibt. Die Abfrage, die dieses Problemmuster erfasst, wird in Abbildung 4.2 dargestellt. Der erste Teil der Abfrage entspricht dem vorangehenden und besteht aus einer Iteration über alle Felder, aller Klassen, aller Pakete dieses Modells hinweg. Für jedes *Field*, wird die Liste der auf es verweisenden Zugriffe überprüft. Falls sie leer ist bedeutet dies, dass das Feld ungenutzt ist.

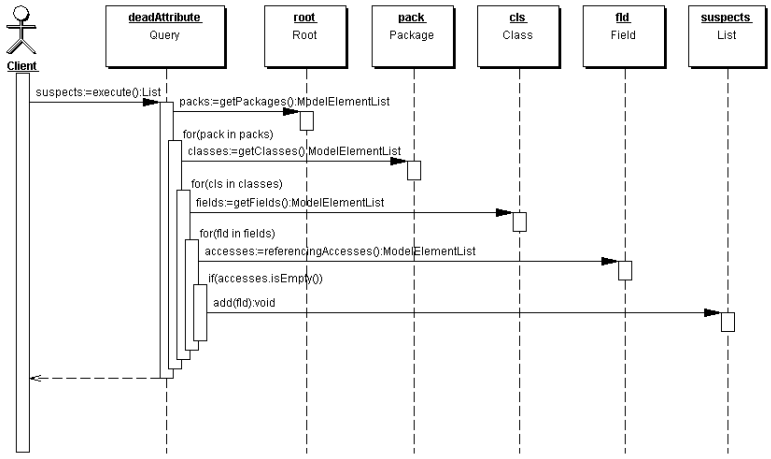

Abbildung 4.2: Tote Attribute

4.1.3 Unvollständige Vererbung

Dieses Problemmuster befasst sich mit einem häufigen Defizit in Vererbungshierarchien, nämlich der erstmaligen Definition derselben Methode in mehreren Klassen einer Hierarchiestufe, anstatt einer Definition in einer der gemeinsamen Oberklassen. Die Abfrage sucht nach Methodenpaaren, die dieselbe Signatur haben und die das erste Mal in der Hierarchie definiert werden, das heißt sie überschrieben keine Methode, die in einer gemeinsamen Oberklasse definiert ist. Zuerst erstellt die Abfrage eine temporäre Liste, die alle Methoden innerhalb des Modells enthält durch Iteration über alle Methoden, aller Klassen, aller Pakete in dem Modell. Danach sucht die Abfrage für jede Methode in allen Klassen des Modells nach passenden Signaturen. Wenn eine Übereinstimmung gefunden wird, beginnt die Suche nach einer gemeinsamen Oberklasse in der

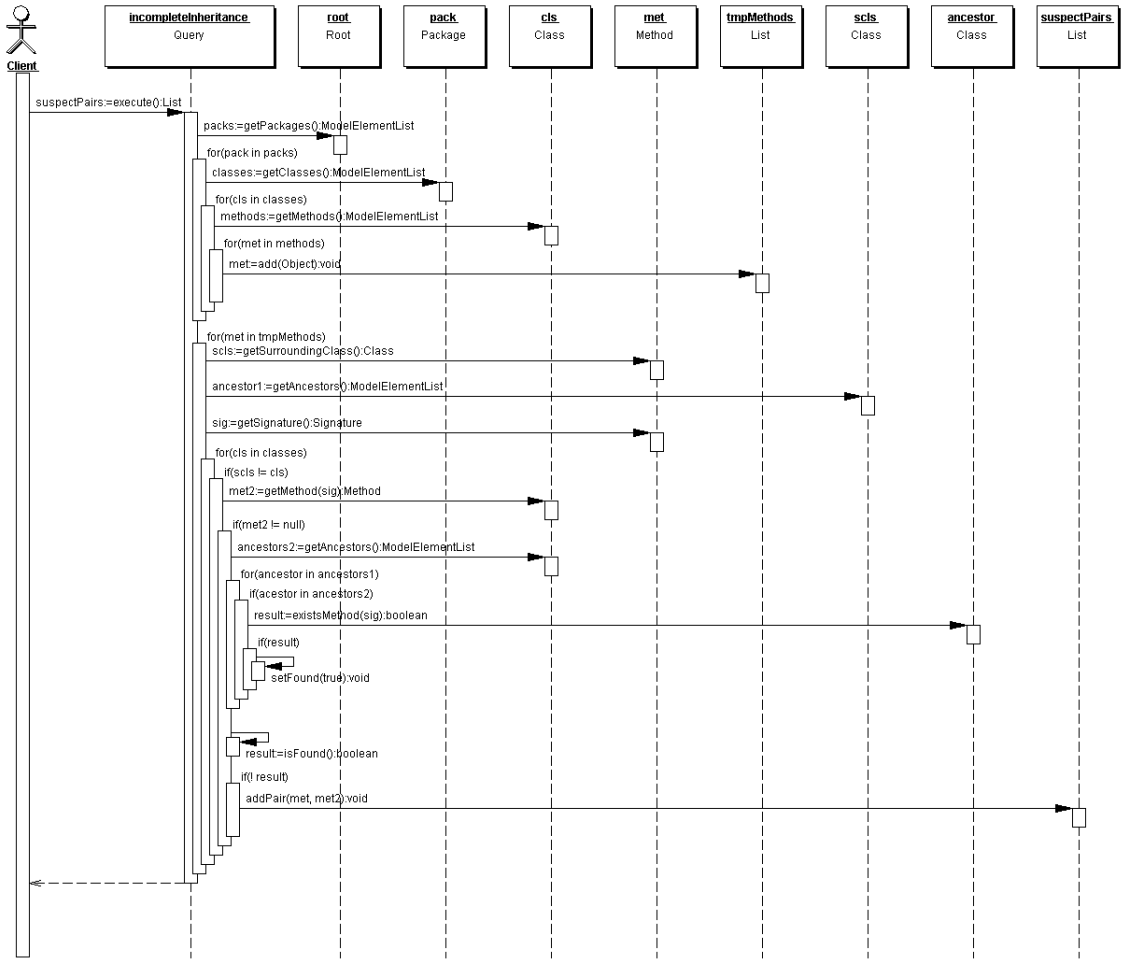

Abbildung 4.3: Unvollständige Vererbung

eine Methode mit derselben Signatur definiert wird. Wenn keine gemeinsamer Oberklasse gefunden wird, wird das Methodenpaar der Liste der Verdächtigen hinzugefügt. Abbildung 4.3 zeigt die Abfrage.

4.2 Transformationen

Dieser Abschnitt stellt drei Beispiele komplexerer Transformation dar, die mit Hilfe der Primitivfunktionen des Transformationsmodells implementiert werden. Zwei der Transformationen (Feld Kapseln und Methode Verschieben) stellen mögliche Lösungen für zwei der Problemmuster (Datenkapselaufbruch bzw. Unvollständige Vererbung) dar, die im vorigen Abschnitt dargestellt wurden, während die dritte eine typische Transformation darstellt, die verwendet wird um globale Variablen zu entfernen. Die folgenden Unterabschnitte beschreiben die Implementierung dieser Transformationen.

4.2.1 Umwandlung einer globale Variable zu einem Feld

Als schlechter Stil werden globale Variablen in OO Code angesehen. Grund dafür ist, dass globale Variabeln versteckte Kommunikation zwischen Objekten erlauben, die schwer nachzuvollziehen ist und daher die Wartbarkeit ernsthaft behindern kann. In vielen Fällen ist es möglich, solche globalen Variablen in Felder umzuformen und sie in eine angemessene Klasse zu verschieben. Die Transformation, die in Abbildung 4.4 gezeigt wird, stellt einen möglichen Weg dafür dar. Die Transformation beginnt mit der Konstruktion eines neuen *Field*-Objekts, wobei der Name sowie der deklarierte Typ der globalen Variablen verwendet wird. Das neu geschaffene *Field* wird als öffentlich markiert und einer Klasse, die als Parameter spezifiziert ist, der Transformation angefügt. Schließlich wiederholt sich die Transformation über die Liste der Zugriffe, die auf die globalen Variablen verweisen und gleicht das Ziel jedes solchen Zugriff an, um auf das neu geschaffene *Field* zu zeigen. Zu beachten ist, dass diese Transformation nur eine Annäherung an die Operationen, die tatsächlich im Quelltext ausgeführt werden müssen, ist, da sie weder Deklarationen noch korrespondierende Zugriffe auf Objekte, die das Feld enthält einführt. Allgemein ist es nicht möglich, solche Deklarationen und Zugriffe automatisch einzuführen, weil die Laufzeitstruktur des Systems unbekannt ist.

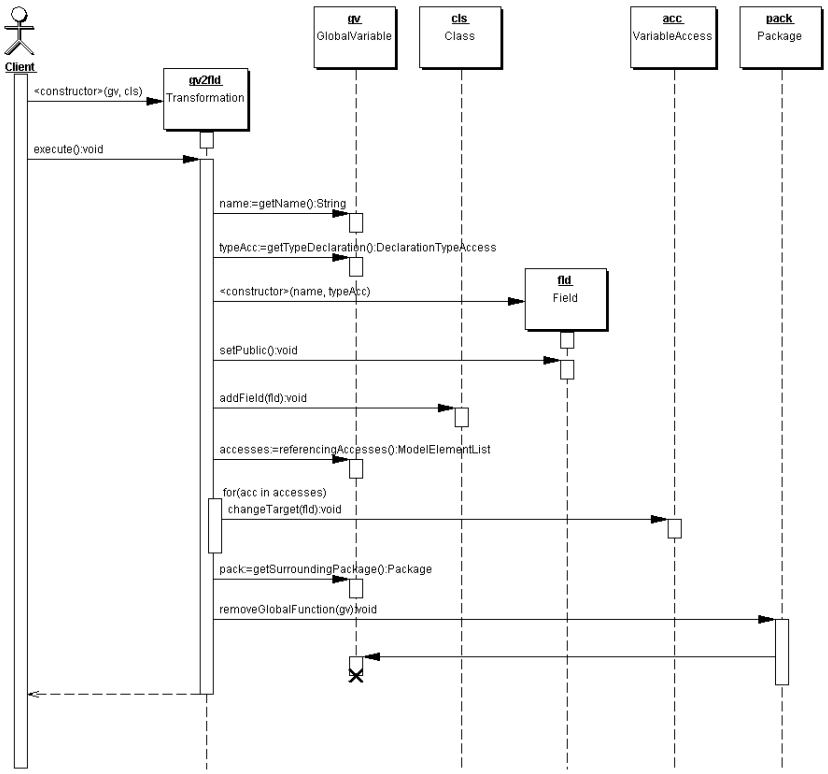

Abbildung 4.4: Umwandlung einer globale Variable zu einem Feld

4.2.2 Feld kapseln

Diese Transformation stellt eine der möglichen Lösungen für das Problemmuster Datenkapselaufbruch dar. Die Idee, die dahinter steht, ist der Versuch das öffentliche Feld zu kapseln. Das bedeutet, dass das Feld private gemacht wird und geeignete Zugriffsmethoden geschaffen werden. Abbildung 4.5 zeigt eine mögliche Implementierung dieser Transformation. Theoretisch sind die Schritte zur Kapselung eines Feldes wohl bekannt und können relativ gut automatisiert werden. Zuerst wird das Feld als privat markiert und zwei öffentliche Zugriffsmethoden werden geschaffen und den Klassen, die das Feld enthält hinzugefügt. Der folgende Schritt involviert, dass alle Zugriffe, die auf das Feld (*VariableAccess*) hinweisen durch Zugriffe auf die korrespondierende Zugriffsmethode (*FunctionAccess*) ersetzt werden. Die geeignete Zugriffsmethode wird basierend auf dem Typ des *VariableAccess* festgelegt: (*read access* oder *write access*).

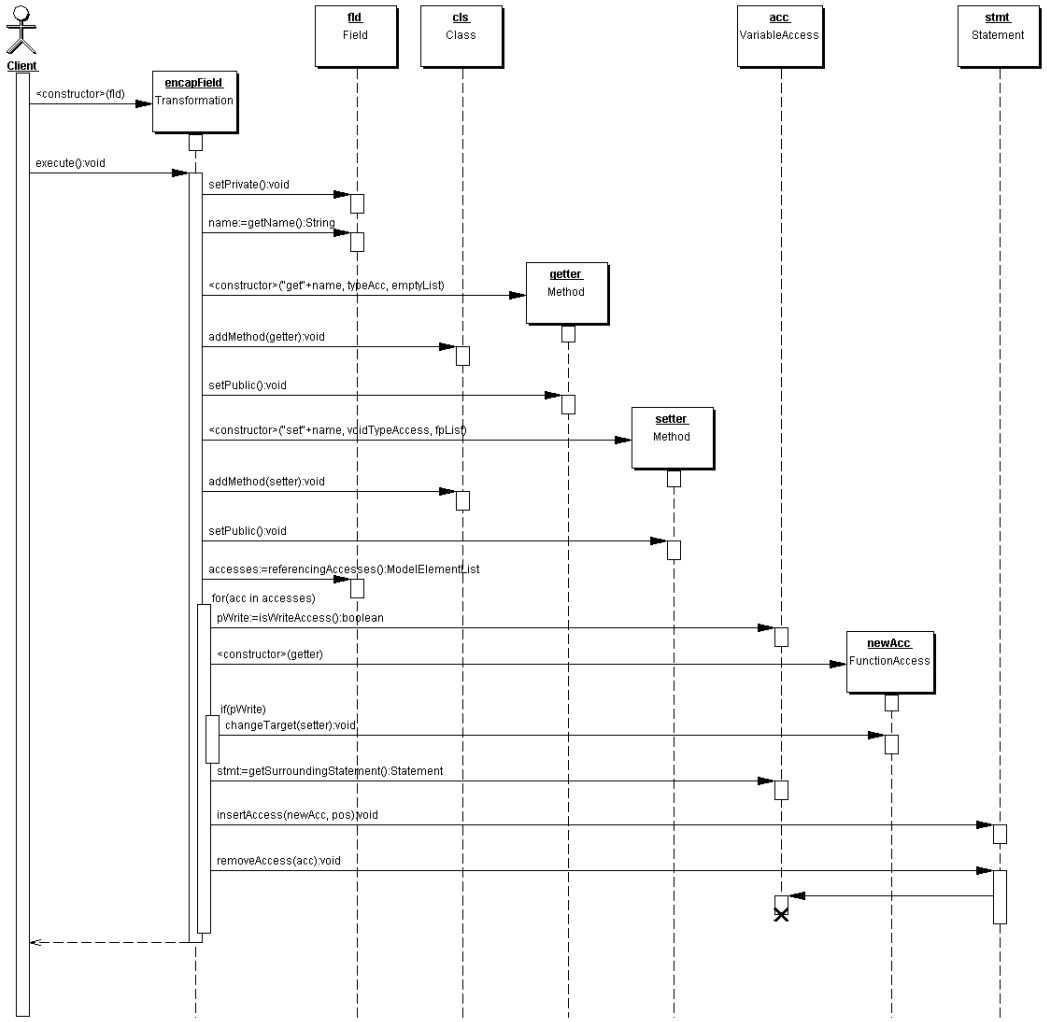

Abbildung 4.5: Feld Kapseln

4.2.3 Methode verschieben

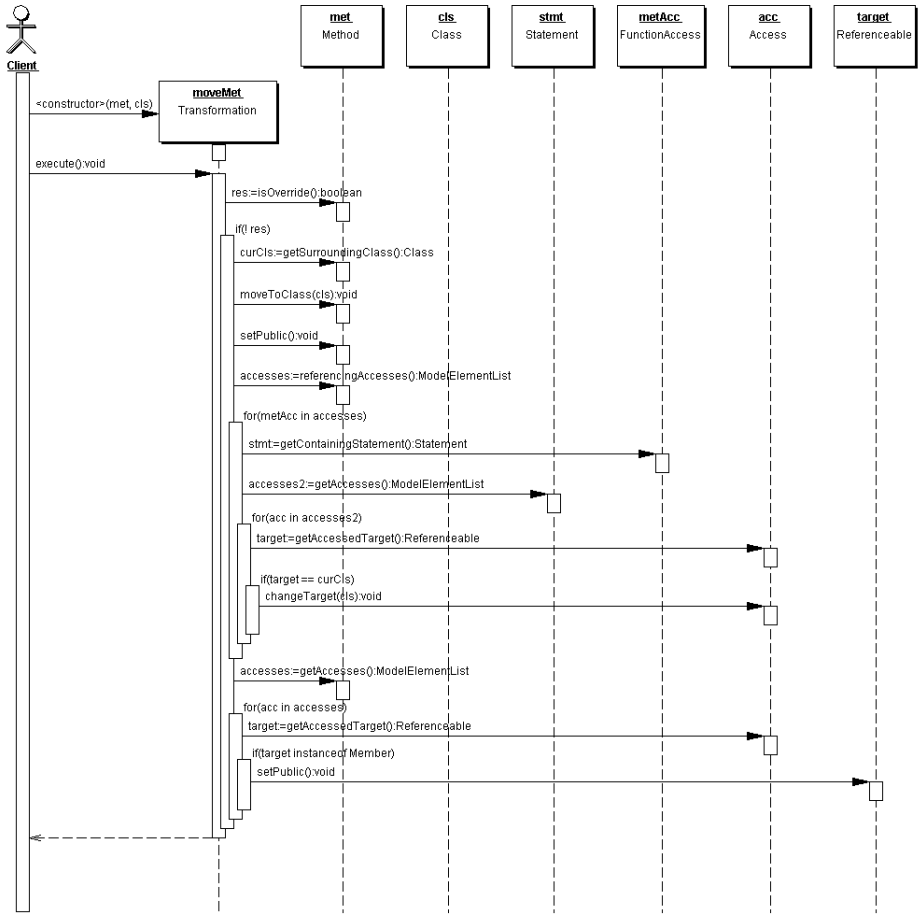

Abbildung 4.6: Methoden Verschiebung

Diese Transformation könnte eine Lösung für das Problemmuster unvollständiger Vererbung darstellen. Abhängig von den Implementierungen der Methoden gleicher Signatur könnte es sein, dass sie komplette Codeduplikate sind. In diesem Fall könnte es die beste Lösung sein, die eine von ihnen zu entfernen, und die andere zu einer gemeinsamen Oberklasse zu verschieben. Eine Methode zu einer anderen Klasse zu verschieben ist eine komplexe Transformation, die besondere Vorsicht erfordert. Die Implementierung wird in Abbildung 4.6 gezeigt. Sie beginnt damit zu überprüfen, ob die besagte Methode eine polymorphe Methode ist. Falls sie es ist, so kann sie nicht verschoben werden. Falls nicht, so wird die Methode als öffentlich gekennzeichnet und zu der Zielklasse, die als Argument für die Transformation spezifiziert wird verchoben. Dann versucht die Transformation für alle Zugriffe, die auf die Methode verweisen, den Zugriffspfad anzupassen. Zu beachten ist, dass dies ebenfalls eine Näherungslösung ist, da keine Deklarationen für die neue Klasse an jedem Zugriffsort hinzugefügt werden. Schließlich wiederholt sich die Transformation über alle Zugriffe innerhalb der Methode und markiert alle Merkmale der ursprünglichen Klasse, auf die zugegriffen wird, als öffentlich.

Kapitel 5

Modellpersistenz

5.1 Methoden zur persistenten Speicherung von Modellen

Die persistente Speicherung eines Modells erlaubt es, das Modell zu einem späteren Zeitpunkt und gegebenenfalls an einem anderen Ort wieder zu rekonstruieren, ohne Zugriff auf die Daten zu haben, aus welchen das Modell ursprünglich erstellt wurde. Diese Eigenschaft ist insbesondere für das in Abschnitt 3.4 vorgestellte Analyse-/Transformationsmodell interessant. Dies hat mehrere Gründe:

- Die an den Analyseergebnissen interessierten Kunden erlauben oftmals nicht den unkontrollierten, sich über einen längeren Zeitraum erstreckenden Zugriff auf den zu analysierenden Quelltext beziehungsweise die Weitergabe oder Bereitstellung des Quelltexts, da es sich bei den Quelltexten um Betriebsgeheimnisse handelt. Aus dem Analyse-/Transformationsmodell lässt sich der Quelltext eines Systems nicht wieder generieren, sondern höchstens Coderahmen. Durch das Analyse-/Transformationsmodell wird somit nur mehr oder weniger das Design des zu analysierenden Systems zur Verfügung gestellt, die eigentliche Business-Logik dagegen nicht. Daher ist die Bereitstellung des Analyse-/Transformationsmodells nicht in gleichem Maße sicherheitskritisch wie die Bereitstellung des Quelltexts.
- Die Durchführung der Analyse kann offline, das heißt zu einem späteren Zeitpunkt wie die Faktenextraktion und ohne Zugriff auf den Quelltext erfolgen.
- Die Analyse kann parallel/mehrmals (im Sinne von unabhängig voneinander) von mehreren Assessoren durchgeführt werden, auf der gleichen Datenbasis

Die Speicherung des Modells kann auf mehrere Arten erfolgen. Eine Möglichkeit ist, das Modell in einer für den Menschen lesbaren Textdatei persistent zu speichern. Die Speicherung in einem Textformat kann dabei wiederum dadurch unterschieden werden, ob ein proprietäres oder ein standardisiertes Format verwendet wird. Ein Beispiel für ein proprietäres textbasiertes Format ist das bisher für Analysemodelle am FZI benutzte Simple Relational Format (SRF). Ein Vorteil eines proprietären Formats ist im Allgemeinen, dass es maßgeschneidert für den jeweiligen Anwendungszweck ist. Dadurch ist es meist auch einfacher benutzbar. Im Falle von SRF ist zum Beispiel eine nachträgliche manuelle Änderung der Textdatei ohne größere Probleme möglich. Durch die Optimierung auf den jeweiligen Anwendungszweck sind solche speziellen Textformate oft auch speichersparender, verglichen mit standardisierten Lösungen. Ein Nachteil eines proprietären Textformats ist die nicht vorhandene Werkzeugunterstützung. Es existieren initial keine Parser, Import- und Exportfunktionen für IDEs, Visualisierungswerkzeuge etc. Alle diese Werkzeuge müssen explizit für dieses Textformat erstellt werden.

Neben der textuellen Speicherung des Modells bietet sich auch die Möglichkeit der binären Speicherung des im Speicher befindlichen Modells. Dies kann sowohl in einer Datei als auch

in einer entsprechenden Datenbank beziehungsweise in einem Modellrepository erfolgen. Ein Vorteil der binären Speicherung ist der geringe Speicherbedarf, da es sich um eine effiziente Repräsentation handelt. Bei entsprechender Codierung benötigt die binäre Datei nicht mehr Platz als das Modell im Speicher. Ein Nachteil der binären Speicherung ist die Abhängigkeit von einer Plattform beziehungsweise von der Datenbank. Hierdurch wird die Interoperabilität zwischen verschiedenen Werkzeugen im Allgemeinen sehr eingeschränkt. Weiterhin ist es dem Benutzer aufgrund der binären Codierung des Analyse-/Transformationsmodells nicht möglich, diese Repräsentation mit Hilfe eines Texteditors nachträglich zu manipulieren. Eine Ausnahme bilden die Datenbanken, die allgemein akzeptierte Standards, wie z.B. SQL unterstützen. Hier hat der Benutzer sogar die Möglichkeit selber SQL-Anfragen über das gespeicherte Modell formulieren.

Sowohl im Falle einer binären Speicherung als auch bei Benutzung einer textbasierten Repräsentation kann die Möglichkeit der Serialisierung von Objekten genutzt werden, wie sie zum Beispiel verschiedene Rahmenwerke (unter anderem MFC, .NET oder auch Java) zur Verfügung stellen. Eine andere Möglichkeit, die für eine persistente Speicherung relevanten Daten aus dem Modell zu erhalten, liegt in der Benutzung von Introspektion, das heißt der Untersuchung der einzelnen Objekte durch ein entsprechendes Metaprogramm.

Um das in Abschnitt 3.4 vorgestellte Analyse-/Transformationsmodell persistent speichern zu können, werden im weiteren Projektverlauf zwei von den oben genannten Methoden zur Speicherung - die Serialisierung in Java-Objekte bzw. eine SQL-Datenbank - implementiert. Im folgenden Abschnitt geben wir das SQL-Schema des Metamodells an.

5.2 SQL-Schema des Metamodells

In diesem Abschnitt stellen wir das SQL-Datenbankschema vor. Die *kursiv* geschriebenen Attribute stellen redundante, berechnete Informationen dar. Sie sind aus Effizienzgründen notwendig, weil sie in zukünftigen SQL-Anfragen (zum Beispiel bei der Berechnung von Metriken) häufig benutzt werden und die erneute Berechnung dieser Attribute einerseits die Anfrage komplizierter andererseits langsamer macht.

Die Modellelementen des Systemmetamodells werden auf die folgenden Tabellen abgebildet:

- TModelElements: Speichert die Instanzen von Unterklassen von *ModelElement*.
- TAnnotations: Abbildung der *annotations* Assoziation.
- TSourceEntities: Speichert eine Position, entspricht der Schnittstelle *SourceEntity*.
- TFiles: Speichert Instanzen von *File*.
- TPackages: Speichert Instanzen von *Package*.
- TMembers: Speichert Instanzen von Unterklassen von *Member*.
- TTypes: Speichert Instanzen der Typhierarchie.
- TFriends: Abbildung der *friendClasses* und *friendFunctions* Assoziationen.
- TTypeParameters: Abbildung der *typeparameters* Assoziation.
- TTypeArguments: Abbildung der *typeArgs* Assoziation.
- TFunctions: Speichert Instanzen von Unterklassen von *Function*.
- TSignatures: Speichert Instanzen von *Signature*.
- TVariables: Speichert Instanzen von Unterklassen von *Variable*.

- TStatements: Speichert Instanzen von Unterklassen von *Statement*.
- TAccesses: Speichert Instanzen von Unterklassen von *Access*.
- TImports: Speichert die imports-Beziehungen.
- TInheritances: Speichert die transitive Hülle der Vererbungsbeziehungen.
- TPackageContainmentRelations: Speichert die transitive Hülle der Paketzugehörigkeitsbeziehung.
- TClassContainmentRelations: Speichert die transitive Hülle der . Klassezugehörigkeitsbeziehung.
- TClones: Speichert Code-Duplikate.
- TCloneInstances: Speichert Code-Duplikat-Instanzen.
- TCloneInstanceStatements Speichert die Beziehung zwischen Code-Duplikat-Instanzen und Anweisungen.
- TAbstractions: Speichert Instanzen von *StructuralAbstraction* und Unterklassen von *StructuralAbstraction*.
- TAbstractionElements: Speichert die containment-Beziehungen für die Instanzen von *StructuralAbstraction*.
- TComments: Speichert die Code-Kommentare.
- TConstants: Speichert symbolische Konstante.

Im folgenden Abschnitt beschreiben wir die Tabellen detaillierter. Für jedes Feld wird den Namen, den SQL-Datentyp und eine kurze Beschreibung gegeben. Weil nicht alle SQL-Datenbanksysteme den primitiven Typ *Boolean* implementieren, wir haben ihn als *32 Bit Integer* kodiert.

5.2.1 Tabellen

TModelElements	
Attribute	
Id	Eindeutiger Bezeichner vom Element (32 Bit Integer).
Name	Der einfache (nicht voll qualifizierte) Name vom Element (Varchar(256)).
Statute	Siehe die Beschreibung von der *ModelElement* Klasse (32 Bit Integer). Es kann die folgenden Werte aufnehmen: STATUTE_NORMAL STATUTE_LIBRARY STATUTE_IMPLICIT STATUTE_FAILEDDEP
KindOfElement	Bestimmt den Typ des Modellelementes (32 Bit Integer). Es kann die folgenden Werte aufnehmen: ROOT FILE_SOURCE FILE_ASSEMBLY COMMENT PACK_PACKAGE

	PACK_SUBPACKAGE
	PACK_PACKAGEALIAS
	TYPE_PRIMITIVE
	TYPE_INTERFACE
	TYPE_CLASS
	TYPE_GENERICINTERFACE
	TYPE_GENERICCLASS
	TYPE_TYPEPARAMCLASS
	TYPE_ARRAY
	TYPE_TYPEALIAS
	TYPE_IMPLICITREFERENCE
	TYPE_EXPLICITREFERENCE
	FUNC_TYPE_DELEGATE
	FUNC_GLOBALFUNC
	FUNC_GENERIC
	FUNC_METHOD
	FUNC_CONSTRUCTOR
	FUNC_DESTRUCTOR
	FUNC_INITIALIZER
	FUNC_PROPGETTER
	FUNC_PROPSETTER
	FUNC_UNITINITIALIZER
	FUNC_UNITFINALIZER
	VAR_FIELD
	VAR_LOCALVAR
	VAR_GLOBALVAR
	VAR_FORMALPARAM
	VAR_CATCHPARAM
	VAR_PROPERTY
	STATEMENT_SIMPLE
	STATEMENT_THROW
	STATEMENT_RETURN
	STATEMENT_JUMP
	STATEMENT_LOOP
	STATEMENT_BRANCH
	STATEMENT_BLOCK
	STATEMENT_EXCEPTIONHANDLER
	STATEMENT_CATCHBLOCK
	VARACCESS_READ
	VARACCESS_WRITE
	PROPACCESS_READ
	PROPACCESS_WRITE
	SELFACCESS_THIS
	SELFACCESS_SUPER
	COMPOSITEACCESS
	FUNCACCESS
	DELEGATEACCESS
	TYPEACCESS_INTERFACEINHERITANCE
	TYPEACCESS_IMPLEMENTATIONINHERITANCE
	TYPEACCESS_DECLAREDTHROW
	TYPEACCESS_THROW
	TYPEACCESS_DECLARATION

	TYPEACCESS_CAST TYPEACCESS_RTTI TYPEACCESS_STATIC CLONE_CLONE CLONE_INSTANCE ABSTRACTION_STRUCTURALABSTRACTION ABSTRACTION_SUBSYSTEM
Scope	Scope vom Element (32 Bit Integer). Es kann die folgenden Werte aufnehmen: SCOPE_GLOBAL SCOPE_PACKAGE SCOPE_CLASS SCOPE_LOCAL innerhalb einer Methode. SCOPE_PROPERTY innerhalb einer Property. SCOPE_DECLARATION innerhalb einer Deklaration. SCOPE_STATEMENT innerhalb einer Anweisung. SCOPE_ACCESS innerhalb eines Zugriffs. SCOPE_FILE
ParentId	Eindeutiger Bezeichner vom Element, das dieses enthält (32 Bit Integer).

TAnnotations	
Attribute	
ModelElementId	Die eindeutige ID eines annotierten Modellelements (32 Bit Integer).
AnnotationId	Die eindeutige ID der Annotation (32 Bit Integer). Dabei ist zu beachten, dass in dieser Datenbank ausschließlich Annotationen, die gleichzeitig Modellelemente sind, verzeichnet sind.

TSourceEntities	
Attribute	
Id	Eindeutiger Bezeichner von diesem Element (32 Bit Integer).
SourceFileId	Eindeutiger Bezeichner von der Quelldatei, die diese Entität enthält (32 Bit Integer).
StartLine	Startposition dieser Entität (32 Bit Integer).
StartChar	Startposition dieser Entität (32 Bit Integer).
EndLine	Endposition dieser Entität (32 Bit Integer).
EndChar	Endposition dieser Entität (32 Bit Integer).
AssemblyFileId	Eindeutiger Bezeichner von der Assembly, die diese Entität enthält (32 Bit Integer).

TFiles	
Attribute	
Id	Eindeutiger Bezeichner der Datei (32 Bit Integer).
KindOfFile	Dateityp (32 Bit Integer). Mögliche Werte sind: FILE_SOURCE FILE_ASSEMBLY
Pathname	Vollständiger Pfad der Datei (Varchar(1024)).
LinesOfCode	Die Anzahl der Codezeilen in dieser Datei (32 Bit Integer).
≪Metrics≫	Beliebige Metriken, die notwendig sein können.

TPackages	
Attribute	
Id	Eindeutiger Bezeichner vom Paket (32 Bit Integer).
KindOfPackage	Typ vom Paket (32 Bit Integer). Es kann die folgenden Werte aufnehmen: PACK_PACKAGE PACK_SUBPACKAGE PACK_PACKAGEALIAS
Name	Der einfache (nicht voll qualifizierte) Name des Pakets (Varchar(256)).
FullName	Der voll qualifizierte Name des Pakets (Varchar(1024)).
ParentPackageId	Die eindeutige ID des umgebenden Pakets (32 Bit Integer).
AliasedPackageId	Falls dieses Programmelement ein *PackageAlias* ist, gibt dieses Feld die ID des Pakets an, auf das der Alias sich bezieht (32 Bit Integer).
≪Metrics≫	Beliebige Metriken, die notwendig sein können.

TMembers	
Attribute	
Id	Eindeutiger Bezeichner von diesem Element (32 Bit Integer).
KindOfMember	Art des Merkmals (32 Bit Integer). Es kann die folgenden Werte aufnehmen: TYPE_INTERFACE TYPE_CLASS TYPE_GENERICINTERFACE TYPE_GENERICCLASS TYPE_TYPEALIAS FUNC_TYPE_DELEGATE FUNC_METHOD FUNC_CONSTRUCTOR FUNC_DESTRUCTOR FUNC_INITIALIZER FUNC_PROPGETTER FUNC_PROPSETTER VAR_FIELD

	VAR_PROPERTY
Name	Der einfache (nicht voll qualifizierte) Name des Merkmals (Varchar(256)).
ClassId	Die Klasse, die dieses Element in sich kapselt (32 Bit Integer).
Visibility	Sichtbarkeit (32 Bit Integer). Es kann die folgenden Werte aufnehmen: VISIBILITY_PUBLIC VISIBILITY_PROTECTED VISIBILITY_STRICTPROTECTED VISIBILITY_PACKAGE VISIBILITY_PRIVATE
IsInternal	Siehe die Beschreibung vom *Member* Modellelement (32 Bit Integer).
IsAbstract	Siehe die Beschreibung vom *Member* Modellelement (32 Bit Integer).
IsFinal	Siehe die Beschreibung vom *Member* Modellelement (32 Bit Integer).
IsStatic	Siehe die Beschreibung vom *Member* Modellelement (32 Bit Integer).
IsVirtual	Siehe die Beschreibung vom *Member* Modellelement (32 Bit Integer).
IsExtern	Siehe die Beschreibung vom *Member* Modellelement (32 Bit Integer).
IsNew	Siehe die Beschreibung vom *Member* Modellelement (32 Bit Integer).
IsOverride	Siehe die Beschreibung vom *Member* Modellelement (32 Bit Integer).
IsTypeParamMember	Siehe die Beschreibung vom *Member* Modellelement (32 Bit Integer).
IsIntrospectable	Siehe die Beschreibung vom *Member* Modellelement (32 Bit Integer).
OverridenMemberId	Falls dieses Merkmal ein geerbtes Merkmal überschreibt, gibt dieses Feld die ID des überschriebenen Merkmals an (32 Bit Integer).

TTypes	
Attribute	
Id	Eindeutiger Bezeichner vom Typ (32 Bit Integer).
KindOfType	Typsart (32 Bit Integer). Es kann die folgenden Werte aufnehmen: TYPE_PRIMITIVE TYPE_INTERFACE TYPE_CLASS TYPE_GENERICINTERFACE TYPE_GENERICCLASS TYPE_TYPEPARAMCLASS TYPE_ARRAY TYPE_TYPEALIAS TYPE_IMPLICITREFERENCE TYPE_EXPLICITREFERENCE

	FUNC_TYPE_DELEGATE
Name	Der einfache (nicht voll qualifizierte) Name (Varchar(256)).
FullName	Der voll qualifizierte Name (Varchar(1024)).
PackageId	Falls dieser Typ eine Klasse, eine Schnittstelle, ein Typalias oder ein Delegate ist, enthält dieses Feld den Bezeichner des Paketes, das diesen Typ enthält (32 Bit Integer).
ClassId	Falls dieser Typ eine innere Klasse, eine innere Schnittstelle, ein inneres Typalias oder ein inneres Delegate ist, enthält dieses Feld den Bezeichner der Klasse, das diesen Typ enthält (32 Bit Integer).
DecoratedTypeId	Falls es um einen Typdekorator geht, der Bezeichner des dekorierten Typs (32 Bit Integer).
ArrayDimensions	Falls es sich um ein mehrdimensionales Array handelt, zeigt dieses Feld die Zahl der Dimensionen an (32 Bit Integer).
IsReferenceType	Siehe die Beschreibung vom *Type* Modellelement (32 Bit Integer).
≪Metrics≫	Beliebige Metriken, die notwendig sein können.

TFriends	
Attribute	
TypeId	Die eindeutige ID des Typs, der die Friend-Deklaration enthält (32 Bit Integer).
KindOfFriend	Art des Friend-Elements (32 Bit Integer). Es kann die folgenden Werte aufnehmen: TYPE_CLASS TYPE_GENERICCLASS TYPE_TYPEPARAMCLASS FUNC_TYPE_DELEGATE FUNC_GLOBALFUNC FUNC_GENERIC FUNC_METHOD FUNC_CONSTRUCTOR FUNC_DESTRUCTOR FUNC_PROPGETTER FUNC_PROPSETTER
FriendId	Die eindeutige ID des Friend-Elements (Funktion oder Klasse) (32 Bit Integer).

TTypeParameters	
Attribute	
TypeParameterId	Die eindeutige ID der *TypeParameterClass* (32 Bit Integer).
position	Die Stelle dieser *TypeParameterClass* in der Liste im generischen Element (32 Bit Integer).
KindOfGenericElement	Der Typ des generischen Elements (32 Bit Integer). Er kann einen der folgenden Werte haben: TYPE_GENERICINTERFACE TYPE_GENERICCLASS

GenericElementId	FUNC_GENERIC Eindeutiger Bezeichner des generischen Elementes (Funktion oder Klasse) (32 Bit Integer).

TTypeArguments	
Attribute	
AccessId	Die ID des Zugriffs (FunctionAccess oder TypeAccess), der das generische Element instanziiert (32 Bit Integer).
TypeParameterId	Die eindeutige ID der *TypeParameterKlasse* (32 Bit Integer).
TargetTypeId	Eindeutiger Bezeichner des Typs woran der Parameter gebunden ist (32 Bit Integer).

TFunctions	
Attribute	
Id	Eindeutiger Bezeichner der Funktion (32 Bit Integer).
KindOfFunction	Art der Funktion (32 Bit Integer). Es kann die folgenden Werte aufnehmen: FUNC_TYPE_DELEGATE FUNC_GLOBALFUNC FUNC_GENERIC FUNC_METHOD FUNC_CONSTRUCTOR FUNC_DESTRUCTOR FUNC_INITIALIZER FUNC_PROPGETTER FUNC_PROPSETTER FUNC_UNITINITIALIZER FUNC_UNITFINALIZER
Name	Der einfache (nicht voll qualifizierte) Name der Funktion (Varchar(256)).
ReturnTypeDeclarationId	Die eindeutige ID des *TypeAccess*, der den Vereinbarungen des Rückgabetyps dieser Funktion entspricht (32 Bit Integer).
PackageId	Bezeichner des Paketes, das diese Funktion enthält (32 Bit Integer).
ClassId	Falls diese Funktion ein Merkmal ist, enthält dieses Feld den Bezeichner der Klasse, das diese Funktion enthält (32 Bit Integer).
PropertyId	Falls diese Funktion ein Property-Zugriffsmethode ist, enthält dieses Feld den Bezeichner des Propertys, das diese Funktion enthält (32 Bit Integer).
IsOperator	Siehe die Beschreibung vom *Function* Modellelement (32 Bit Integer).
NumberOfStatements	Anzahl der Anweisungen (32 Bit Integer).
NumberOfLines	Anzahl der Codezeilen (32 Bit Integer).
NumberOfComments	Anzahl der Kommentare (32 Bit Integer).
NumberOfExceptions	Anzahl der Ausnahmen (32 Bit Integer).

MaxNestingLevel	Die Maximale Schachtelungstiefe der Kontrollstrukturen im Funktionsrumpf (32 Bit Integer).
NumberOfNodes	Anzahl der Knoten im Kontrollflussgraph des Funktionsrumpfs (32 Bit Integer).
NumberOfEdges	Anzahl der Kanten im Kontrollflussgraph des Funktionsrumpfs (32 Bit Integer).
≪Metrics≫	Beliebige Metriken, die notwendig sein können.

TSignatures	
Attribute	
FunctionId	Die eindeutige ID der Funktion (32 Bit Integer).
Signature	Die Darstellung als *String* der Signatur der Funktion (Varchar(4096).

TVariables	
Attribute	
Id	Der eindeutige Bezeichner von der Variable (32 Bit Integer).
KindOfVariable	Variablenart (32 Bit Integer). Es kann die folgenden Werte aufnehmen: VAR_FIELD VAR_LOCALVAR VAR_GLOBALVAR VAR_FORMALPARAM VAR_CATCHPARAM VAR_PROPERTY
Name	Der einfache (nicht voll qualifizierte) Name der Variable (Varchar(256)).
PackageId	Bezeichner des Paketes, das diese Variable enthält (32 Bit Integer).
ClassId	Falls diese Variable ein Merkmal ist, enthält dieses Feld den Bezeichner der Klasse, das diese Variable enthält (32 Bit Integer).
FunctionId	Bezeichner der Funktion, die diese Variable enthält (32 Bit Integer).
TypeDeclarationId	Die eindeutige ID des *TypeAccess*, der der Deklaration des Typs dieser Variablen entspricht (32 Bit Integer).
position	Nur für *FormalParameter* von Bedeutung. Die Stelle dieses formalen Parameters in der Liste in der Funktion (32 Bit Integer).
IsConst	Siehe die Beschreibung vom *Variable* Modellelement (32 Bit Integer).
IsPassedByRef	Nur für *FormalParameter* von Bedeutung. Siehe die Beschreibung vom *FormalParameter* Modellelement (32 Bit Integer).
IsRethrown	Nur für *CatchParameter* von Bedeutung. Siehe die Beschreibung vom *CatchParameter* Modellelement (32 Bit Integer).

TStatements	
Attribute	
Id	Die eindeutige ID der Anweisung (32 Bit Integer).
KindOfStatement	Art der Anweisung (32 Bit Integer). Es kann die folgenden Werte aufnehmen: STATEMENT_SIMPLE STATEMENT_THROW STATEMENT_RETURN STATEMENT_JUMP STATEMENT_LOOP STATEMENT_BRANCH STATEMENT_BLOCK STATEMENT_EXCEPTIONHANDLER STATEMENT_CATCHBLOCK
FunctionId	Bezeichner der Funktion, die diese Anweisung enthält (32 Bit Integer).
ParentStatementId	Die eindeutige ID der umgebenden Anweisung (32 Bit Integer).
position	Die Stelle dieser Anweisung in der Liste in der umgebenden Anweisung (32 Bit Integer).

TAccesses	
Attribute	
Id	Eindeutiger Bezeichner vom Zugriff (32 Bit Integer).
KindOfAccess	Art des Zugriffs (32 Bit Integer). Es kann die folgenden Werte aufnehmen: VARACCESS_READ VARACCESS_WRITE PROPACCESS_READ PROPACCESS_WRITE SELFACCESS_THIS SELFACCESS_SUPER COMPOSITEACCESS FUNCACCESS DELEGATEACCESS TYPEACCESS_INTERFACEINHERITANCE TYPEACCESS_IMPLEMENTATIONINHERITANCE TYPEACCESS_DECLAREDTHROW TYPEACCESS_THROW TYPEACCESS_DECLARATION TYPEACCESS_CAST TYPEACCESS_RTTI TYPEACCESS_STATIC
position	Die Stelle dieses Zugriff in der Liste in des umgebenden Elementes (32 Bit Integer).
PackageId	Bezeichner des Pakets, das diesen Zugriff enthält (32 Bit Integer).
ClassId	Bezeichner der Klasse, die diesen Zugriff enthält (32 Bit Integer).
FunctionId	Bezeichner der Funktion, die diesen Zugriff enthält (32 Bit Integer).

ClassId	Bezeichner der Anweisung, die diesen Zugriff enthält (32 Bit Integer).
SourceId	Der Bezeichner des Elementes, das diesen Zugriff enthält (32 Bit Integer).
TargetId	Der Bezeichner vom Element, worauf zugegriffen wird (32 Bit Integer).

TImports	
Attribute	
FileId	Eindeutiger Bezeichner der Datei, die dieses Import enthält (32 Bit Integer).
KindOfTarget	Art des importierten Elementes (32 Bit Integer). Es kann die folgenden Werte aufnehmen: TYPE_INTERFACE TYPE_CLASS TYPE_GENERICINTERFACE TYPE_GENERICCLASS TYPE_TYPEPARAMCLASS TYPE_TYPEALIAS FUNC_TYPE_DELEGATE FUNC_GLOBALFUNC FUNC_GENERIC VAR_GLOBALVAR PACK_PACKAGE PACK_SUBPACKAGE FILE_SOURCE
TargetId	Der Bezeichner des importierten Elementes (32 Bit Integer).

TInheritances	
Attribute	
ClassId	Eindeutiger Bezeichner der erbenden Klasse (32 Bit Integer).
SuperId	Eindeutiger Bezeichner der Oberklasse (32 Bit Integer).
DepthOfInheritance	Tiefe der Vererbungshierarchie (32 Bit Integer).

TPackageContainmentRelations	
Attribute	
PackageId	Eindeutiger Bezeichner des Unterpakets (32 Bit Integer).
ContainingPackageId	Eindeutiger Bezeichner des Oberpakets (32 Bit Integer).
DepthOfContainment	Tiefe der Paketzugehörigkeit (32 Bit Integer).

TClassContainmentRelations	
Attribute	
ClassId	Eindeutiger Bezeichner der inneren Klasse (32 Bit Integer).
ContainingClassId	Eindeutiger Bezeichner der außeren Klasse (32 Bit Integer).
DepthOfContainment	Tiefe der Klassezugehörigkeit (32 Bit Integer).

TClones	
Attribute	
Id	Eindeutiger Bezeichner des Code-Duplikates (32 Bit Integer).
NumberOfStatements	Die Anzahl der Anweisungen einer Code-Duplikat-Instanz (32 Bit Integer).
NumberOfLines	Die Anzahl der Codezeilen einer Code-Duplikat-Instanz (32 Bit Integer).

TCloneInstances	
Attribute	
Id	Eindeutiger Bezeichner der Code-Duplikat-Instanz (32 Bit Integer).
CloneId	Bezeichner des Code-Duplikates, das diese Instanz enthält (32 Bit Integer).
FunctionId	Bezeichner der Funktion, die die Anweisungen von dieser Instanz enthält (32 Bit Integer).

TCloneInstanceStatements	
Attribute	
StatementId	Eindeutiger Bezeichner der Anweisung (32 Bit Integer).
CloneInstanceId	Eindeutiger Bezeichner der Code-Duplikat-Instanz, die die Anweisung enthält (32 Bit Integer).

TAbstractions	
Attribute	
Id	Eindeutiger Bezeichner der Abstraktion (32 Bit Integer).
Name	Der Name von der Abstraktion (Varchar(256)).
Type	Der Typname des Typs von der Abstraktion (Varchar(256)).
KindOfAbstraction	Art der Abstraktion (32 Bit Integer). Es kann die folgenden Werte aufnehmen: ABSTRACTION_STRUCTURALABSTRACTION ABSTRACTION_SUBSYSTEM

TAbstractionElements	
Attribute	
AbstractionId	Bezeichner der Abstraktion, die das Modellelement enthält (32 Bit Integer).
ModelElementId	Bezeichner des Modellelementes, die in der Abstraktion enthielt wird (32 Bit Integer).

TComments	
Attribute	
Id	Eindeutiger Bezeichner des Kommentars (32 Bit Integer).
IsFormal	Siehe die Beschreibung vom *Comment* Modellelement (32 Bit Integer).
NumberOfTodos	Die Anzahl der *TODO*-, *HACK*- oder *FIXME*-Tags in diesem Kommentar (32 Bit Integer).
NumberOfLines	Die Anzahl der Codezeilen in diesem Kommentar (32 Bit Integer).
CommentText	Der Kommentartext (Text).

TConstants	
Attribute	
Name	Der Name der symbolischen Konstante (Varchar(256)). Es kann die folgenden Werte aufnehmen: ROOT FILE_SOURCE FILE_ASSEMBLY COMMENT PACK_PACKAGE PACK_SUBPACKAGE PACK_PACKAGEALIAS TYPE_PRIMITIVE TYPE_INTERFACE TYPE_CLASS TYPE_GENERICINTERFACE TYPE_GENERICCLASS TYPE_TYPEPARAMCLASS TYPE_ARRAY TYPE_TYPEALIAS TYPE_IMPLICITREFERENCE TYPE_EXPLICITREFERENCE FUNC_TYPE_DELEGATE FUNC_GLOBALFUNC FUNC_GENERIC FUNC_METHOD

	FUNC_CONSTRUCTOR
	FUNC_DESTRUCTOR
	FUNC_INITIALIZER
	FUNC_PROPGETTER
	FUNC_PROPSETTER
	FUNC_UNITINITIALIZER
	FUNC_UNITFINALIZER
	VAR_FIELD
	VAR_LOCALVAR
	VAR_GLOBALVAR
	VAR_FORMALPARAM
	VAR_CATCHPARAM
	VAR_PROPERTY
	STATEMENT_SIMPLE
	STATEMENT_THROW
	STATEMENT_RETURN
	STATEMENT_JUMP
	STATEMENT_LOOP
	STATEMENT_BRANCH
	STATEMENT_BLOCK
	STATEMENT_EXCEPTIONHANDLER
	STATEMENT_CATCHBLOCK
	VARACCESS_READ
	VARACCESS_WRITE
	PROPACCESS_READ
	PROPACCESS_WRITE
	SELFACCESS_THIS
	SELFACCESS_SUPER
	COMPOSITEACCESS
	FUNCACCESS
	DELEGATEACCESS
	TYPEACCESS_INTERFACEINHERITANCE
	TYPEACCESS_IMPLEMENTATIONINHERITANCE
	TYPEACCESS_DECLAREDTHROW
	TYPEACCESS_THROW
	TYPEACCESS_DECLARATION
	TYPEACCESS_CAST
	TYPEACCESS_RTTI
	TYPEACCESS_STATIC
	CLONE_CLONE
	CLONE_INSTANCE
	ABSTRACTION_STRUCTURALABSTRACTION
	ABSTRACTION_SUBSYSTEM
	SCOPE_GLOBAL
	SCOPE_PACKAGE
	SCOPE_CLASS
	SCOPE_LOCAL
	SCOPE_PROPERTY
	SCOPE_DECLARATION
	SCOPE_STATEMENT
	SCOPE_ACCESS
	SCOPE_FILE

	VISIBILITY_PUBLIC VISIBILITY_PROTECTED VISIBILITY_STRICTPROTECTED VISIBILITY_PACKAGE VISIBILITY_PRIVATE STATUTE_NORMAL STATUTE_LIBRARY STATUTE_IMPLICIT STATUTE_FAILEDDEP
Value	Der Wert der symbolischen Konstante (32 Bit Integer).

5.2.2 Beispiele für SQL-Anfragen

In diesem Abschnitt geben wir zwei Beispiele für SQL-Anfragen basierend auf dem spezifizierten Schema an.

5.2.2.1 Datenkapselaufbruch

Die erste SQL-Anfrage gibt eine Liste von öffentlichen Feldern, die sind keine Konstanten und werden von Außerhalb der Klasse zugegriffen.

```
SELECT DISTINCT
    classOfM.id AS ContainingClassIdOfAccessingFunction,
    classOfM.fullname AS ContainingClassNameOfAccessingFunction,
    m.id as AccessingFunctionId,
    m.name as AccessingFunctionName,
    classOfField.id AS ContainingClassIdOfField,
    classOfField.fullName as ContainingClassNameOfField,
    field.id AS AccessedFieldId,
    field.name as AccessedFieldName
FROM
    TTypes classOfField,
    TMembers member_classField,
    TVariables field,
    TConstants constantField,
    TTypes classOfM,
    TMembers class_m,
    TConstants constantPublic,
    TFunctions m,
    TAccesses tAccesses
WHERE
    classOfField.id != classOfM.id AND
    member_classField.ClassId = classOfField.Id AND
    member_classField.Id = field.Id AND
    field.KindOfVariable = constantField.Value AND
    constantField.Name = 'VAR_FIELD' AND
    member_classField.visibility = constantPublic.Value AND
    constantPublic.Name = 'VISIBILITY_PUBLIC' AND
    NOT (member_classField.isStatic = 't' AND
            member_classField.isFinal = 't') AND
```

```
    tAccesses.targetId = field.id AND
    m.id = tAccesses.functionId AND
    class_m.classId = classOfM.id AND
    class_m.id = m.id AND
    NOT EXISTS (
        SELECT *
        FROM
            TInheritances i
        WHERE
            i.classID = classOfM.id AND
            i.superID = classOfField.id
    );
```

5.2.2.2 Tote Attribute

Die zweite SQL-Anfrage gibt eine Liste von tote Attribute (Feldern).

```
SELECT DISTINCT
    ttype.id AS ClassId,
    ttype.fullName AS ClassFullName,
    field.id As UnusedFieldId,
    field.name As UnusedFieldName
FROM
    TVariables AS field,
    TConstants AS tConstantField,
    TTypes As ttype,
    TMembers As tmember
WHERE
    field.kindOfVariable = tConstantField.Value AND
    tConstantField.name = 'VAR_FIELD' AND
    ttype.id = tmember.classId AND
    tmember.id = field.id

EXCEPT

SELECT DISTINCT
    ttype.id AS ClassId,
    ttype.fullName AS ClassFullName,
    field.id As UnusedFieldId,
    field.name As UnusedFieldName
FROM
    TVariables AS field,
    TConstants AS tConstantField,
    TTypes As ttype,
    TMembers As tmember,
    TAccesses tAccesses
WHERE
    tAccesses.targetId = field.id AND
    field.kindOfVariable = tConstantField.Value AND
    tConstantField.name = 'VAR_FIELD' AND
    ttype.id = tmember.classId AND
    tmember.id = field.id;
```

Kapitel 6

Zusammenfassung

In diesem Dokument wurde das QBench-Systemmetamodell vorgestellt. Dieser Teil besteht aus drei Schichten, dessen Modelle ein System mit unterschiedlichem Abstraktionsgrad beschreiben können. Hierzu wurde einerseits die Notwendigkeit der einzelnen Schichten motiviert, andererseits auch deren Metamodelle spezifiziert (falls sie nicht bereits existierten). Die drei in diesem Dokument betrachteten Schichten sind die folgenden:

2. *Analyse-/Transformationsmodell für objektorientierte Sprachen*
1. *Strukturbaum*
0. *Quelltext*

Da die Metamodelle für den Quelltext und den Strukturbaum für die einzelnen Programmiersprachen durch die jeweilige Sprachspezifikation entweder explizit oder implizit definiert werden, wobei eine mögliche Implementierung eines Java-Strukturbaums in Form von Recoder beschrieben wurde, lag der Fokus auf der genauen Spezifikation eines Metamodells zur (sprachunabhängigen) Analyse und Transformation von großen Softwaresystemen. Dieses Metamodell bietet eine abstrakte Sicht auf ein Softwaresystem, wobei trotzdem zahlreiche zum Beispiel für Qualitätsanalysen wichtigen Implementierungsdetails in das Metamodell übernommen wurden. Um das vorgestellte Analyse-/Transformationsmodell für konkrete Systeme, welche in konkreten Programmiersprachen geschrieben sind, benutzen zu können, wurde die Notwendigkeit der Abbildung des Analyse-/Transformationsmodells auf eine Sprache motiviert. Darüberhinaus wurde im Rahmen dieses Dokuments eine Abbildung auf die Sprache Java spezifiziert. Zuletzt wurden Möglichkeiten der persistenten Speicherung der Modelle betrachtet. Die Speicherung der Modelle (Metamodellinstanzen) erfolgt dabei aufbauend auf einer SQL-Datenbank.

Literaturverzeichnis

[AMS+04] C. ANDRIESSENS, T. MOHAUPT, O. SENG, F. SIMON, A. TRIFU und M. WINTER: *Modellzentrierte Softwareentwicklung - Stand der Technik, FZI Report 1-6-12/2*. Technischer Bericht, Forschungzentrum Informatik, Karlsruhe, 2004.

[Bor03] BORLAND SOFTWARE CORPORATION: *Delphi Language Specification*. Delphi IDE Help System, 2003.

[ES90] MARGARET A. ELLIS und BJARNE STROUSTRUP: *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[GHJV95] E. GAMMA, R. HELM, R. JOHNSON und J VLISSIDES: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJSB00] JAMES GOSLING, BILL JOY, GUY STEELE und GILAD BRACHA: *The Java Language Specification*. Addison-Wesley, 2000.

[GK01] THOMAS GENSSLER und VOLKER KUTTRUFF: *Inject/J WWW Page*. http://injectj.sourceforge.net/, 2001.

[Int02] ECMA INTERNATIONAL: C# *Language Specification*. http://www.ecma-international.org/publications/files/ecma-st/Ecma-334.pdf, 2002.

[KOP+03] GERGELY KIS, JOZSEF OROSZ, MARTON PINTER, ZOLTAN LASZLO und THOMAS GENSSLER: *Metaprogramming Library for the* C#*Programming Language*. In: *Proceedings of the Joint Modular Language Conference*. Springer LNCS, August 2003.

[Kut03] V. KUTTRUFF: *CompoBench-Metamodell (untere Schichten), CompoBench Projektergebnis*. Technischer Bericht, Forschungzentrum Informatik, Karlsruhe, 2003.

[Lis00] RAY LISCHNER: *Delphi in a Nutshell*. O'Reilly & Associates, 2000.

[LN02] ANDREAS LUDWIG und RAINER NEUMANN: *RECODER Webseite*. http://sourceforge.net/projects/recoder/, 2002.

[Obj01] OBJECT MANAGEMENT GROUP: *Unified Modeling Language Specification, Version 1.4*, 2001. http://www.omg.org/technology/documents/formal/uml.htm.

[Obj02] OBJECT MANAGEMENT GROUP: *MOF 1.4 Specification*, 2002. http://www.omg.org/cgi-bin/doc?formal/02-04-03.pdf.

[ST05] O. SENG und M. TRIFU: *Faktenextraktoren, QBench Projektergebnis*. Technischer Bericht, Forschungzentrum Informatik, Karlsruhe, 2005.

[Str97] BJARNE STROUSTRUP: *The C++ Programming Language*. Addison-Wesley, 1997.

[Tea02] RECODERCS TEAM: *Recoder.*C# *WWW Page*. http://recoder-cs.sourceforge.net/, 2002.