Technische Universität München
Fakultät für Informatik
Diplomarbeit

Prototypisches Werkzeug für die agile Modellierung von Softwaresystemen

Markus Pister

Aufgabensteller:	Prof. Dr. Manfred Broy
Betreuer:	Dr. Bernhard Rumpe
Abgabedatum:	15. Juli 2002

Hiermit erkläre ich, dass ich die Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 19. Juli 2002

Markus Pister

Inhaltsverzeichnis

Kapitel 1

Einleitung

In den letzten Jahren wurde die Entwicklung von Softwaresystemen zunehmend aufwendiger und komplexer. So ist es nachvollziehbar, dass die Entwickler immer öfter zu Modellierungstechniken greifen, die nur einen für sie relevanten Teil des Systems abstrakt beschreiben und verständlich machen. Graphische Darstellungen, wie zum Beispiel Flussdiagramme, Aufrufbäume und Strukturdiagramme geben durch ihre Anschaulichkeit bei entsprechender Abstraktion oftmals sogar fachfremden Personen einen Einblick in ein Softwaresystem. Zusätzlich können mittels formaler Angaben Eigenschaften des Systems beschrieben werden, die aus dem Programmcode heraus kaum erkennbar oder lesbar wären. Im Laufe der Zeit haben sich sehr viele Modellierungstechniken für Softwaresysteme entwickelt. Allerdings waren die meisten auf die speziellen Anforderungen eines Systemtyps zugeschnitten. Aus Mangel eines Standards wurden verschiedene Modellierungstechniken entwickelt, die aber oftmals die gleichen Aspekte beschreiben.

Es wurden einige Versuche zur Vereinheitlichung der Modellierungstechniken unternommen. Ein sehr erfolgreiches Projekt war die Zusammenfassung einiger Modellierungstechniken in der *Unified Modeling Language* (UML - [Oes01]). In dieser Sprache wird lediglich die Notation, nicht aber die Methodik festgelegt, die ihren Einsatz beschreibt. Sie ist also überaus vielfältig einsetztbar. Inwiefern eine einheitliche Methodik für alle Softwarebereiche sinnvoll ist, ist ohnehin nicht abzusehen. Die UML konnte sich schnell verbreiten, denn in den Softwareprojekten konnte mit den vorhandenen Entwicklungsprozessen weitergearbeitet werden. Im Jahre 1997 wurde die UML bei der Object Management Group (OMG) eingereicht, welche als standardsetzende Organisation ein hohes Ansehen hat. Diese hat sie weiterentwickelt und inzwischen die Version 1.4[1] veröffentlicht. Es ist davon auszugehen, dass sich die UML in naher Zukunft als Standardmodellierungssprache etablieren wird. Bereits heute gibt es etliche Programme, wie zum Beispiel *Rational Rose*[2], die die Erstellung von UML-Modellen unterstützen.

Mittlerweile gibt es eine Vielzahl von Methoden und Softwareentwicklungsprozessen, die auf der Sprache UML basieren. Diese haben die starke Einbindung von Model-

[1] www.omg.org

[2] www.rational.com, Rational - Rational Rose

len in den Entwicklungprozess gemeinsam und schreiben dabei aber nur selten den gezielten Einsatz von Verfeinerung und Abstraktion vor. Bei den Softwareentwicklungsprozessen hat sich in den letzten Jahren nicht nur UML als Modellierungsmittel, sondern es haben sich oft auch die organisatorischen Rahmenbedigungen geändert. Das *Wasserfallmodell* mit nur einem Entwicklungszyklus wich zunächst iterativen Vorgehensweisen wie dem *Rational Unified Process* [Ver00] mit einigen, relativ langen Entwicklungszyklen. Mittlerweile werden immer mehr Methoden wie *eXtreme Programming* [Bec00] mit sehr kurzen Entwicklungszyklen verwendet, in denen eine sehr grobe Spezifikation erst im Verlauf der Entwicklung verfeinert bzw. sogar verändert wird. Im Frühjahr dieses Jahres (2002) findet sich in einer Computer-Zeitschrift ein Artikel mit folgender Überschrift:

> *CASE-Werzeuge helfen speziell in Großprojekten, den Überblick zu behalten. Nicht selten jedoch geht im Eifer der Programmierung die Konsistenz zwischen Modell und Implementierung verloren. Ein Top-Down-Ansatz kann Abhilfe schaffen.* (aus [Bre02])

Mit den Problemen bei der Einbindung der UML in kurzzyklische Entwicklungsprozesse beschäftigen sich mehrere Gruppen aus Wirtschaft und Forschung. Das Thememgebiet ihrer Arbeit ist unter dem Begriff *Agile Modeling* [Amb02] bekannt geworden. Eines der offenen Probleme ist die Erhaltung der Konsistenz zwischen Code und Modell. Ein vielversprechender Lösungsansatz besteht in dem automatischen Generieren von Code aus den Modellen. Allerdings ist es heute fast immer notwendig, den generierten Code mit handgeschriebenem zu ergänzen, was oftmals wieder zu Inkonsistenz führt. Diesem läßt sich auf zwei Wegen entgegenwirken.

Der eine besteht darin, die Generierung umkehrbar zu machen, und so Codeänderungen, die das Modell betreffen, in diesem sofort festzuhalten. Die Software *Together*[3] kann sowohl aus Klassendiagrammen Java generieren als auch umgekehrt. Allerdings weist dieser Weg sehr viele Probleme auf, inbesondere jenes, dass ein einzelnes Modell immer unter einem bestimmten Gesichtspunkt erstellt ist, also nur einzelne Aspekte abstrahiert. Für eine automatische Modellgenerierung ist es jedoch überaus schwierig, diese individuellen Interessen und Fragestellungen zu berücksichtigen.

Der zweite Weg besteht in einem *Top-Down*-Ansatz. Er ähnelt dem allgemeinen Verfahren bei der Einführung von höherwertigen Sprachen. Der Code der höherwertigen Sprache wird mit Versatzstücken aus einer niedrigeren Sprache ergänzt. So wurden in den Anfangszeiten von C Assemblerteile in den C-Code eingebettet. Der in dieser Arbeit betrachtete Ansatz spielt sich eine Ebene höher ab, in die Modellierungssprache wird eine Hochsprache eingebettet. Damit ist erreicht, dass der aus dem Modell generierte Code nicht mehr manuell verändert werden muss.

In dieser Diplomarbeit wird ein Implementierungsvorschlag für den oben genannten Top-Down-Ansatz eingehend beleuchtet. Als Zielsprache für die Implementierung bietet sich Java an, eine weit verbreitete objektorientierte Spache, mit der mittlerweile viele IT-Entwickler umgehen können. Dazu wird eine speziell für dieses Projekt geeignete Teilmenge der UML (UML/P) verwendet, die aus Klassendiagrammen, OCL und Objektdiagrammen besteht. Diese Teilmenge ist, entsprechend

[3] www.togethersoft.com, Togethersoft - Together

der Zielsprache Java, leicht modifiziert und wird um Java-Methodenrümpfe ergänzt.

In dieser Richtung gibt es bereits Software, die relativ weit entwickelt ist. So können mit dem Programm *Bold* (siehe [Rot02]) aus UML-Modellen von *Rational Rose* in Delphi persistente Objekte und Klassenrahmen erzeugt werden, in denen die UML so weit wie möglich umgesetzt wird. Der Entwickler muss allerdings bei diesem Programm noch einzelne, im generierten Code enthaltene Platzhalter von Hand füllen. *Bold* kontrolliert dabei, ob die nachträglich eingefügten Elemente mit den Modellen konform sind. Ähnlich läuft das Verfahren bei *Poseidon*[4] ab, einem Codegenerierungswerkzeug, dass sich aus einem universitären Projekt namens *Argo UML*[5] entwickelt hat. Auch hier wird die UML weitestgehend umgesetzt, und der Entwickler ergänzt dann den generierten Code mit handgeschriebenen Methodenrümpfen in reinem Java.

Auch in dem hier untersuchten Ansatz wird die Modellierungssprache mit Coderümpfen ergänzt. Dies geschieht aber im Gegensatz zu den anderen Programmen nicht erst im generierten Code, sondern der Entwickler gibt die Rümpfe eingebettet in den Modellen an. Mit diesem Konzept können Coderümpfe in ihrem sprachlichen Umfang sehr gut an UML/P angepasst werden. Um die Konformität zwischen Modell, Coderümpfen und dem generierten Code sicherzustellen, können automatische Korrekturen an den Rümpfen vorgenommen werden.

In dem ersten Kapitel dieser Arbeit wird die Sprache UML/P vorgestellt, die dem Entwickler als Modellierungs- und Entwicklungssprache zur Verfügung stehen soll. Das zweite Kapitel beschäftigt sich mit der Implementierung des Werkzeugs zur Codegenerierung. Dabei wird versucht, Anforderungen, die beim *Agile Modeling* auftreten, ebenso zu berücksichtigen, wie die Verwendbarkeit bei einer Weiterentwicklung. Die Implementierung kann in drei Teile (Parser, Auswertung und Generierung) gespalten werden, von denen dann in den folgenden Kapiteln jeder Teil bei einer feineren Spezifikation genauer untersucht wird.

[4]www.gentleware.de, Gentleware - Poseidon for UML
[5]www.argouml.org

Kapitel 2

UML/P - Eine Sprache für Agile Modeling

Die Sprache UML/P ist eine in [Rum02] beschriebene Teilmenge der Sprache UML. Sie hilft dem Entwickler, indem sie unterstützende Modellierungsmittel speziell für die Generierung von Java-Code zur Verfügung stellt. Die Auswahl und Modifikation der Sprachelemente ist dabei so gestaltet, dass ihre Nutzung insbesondere für den Bereich des *Agile Modeling* geeignet ist. Aus den zwei Bestandteilen, *Agile* und *Modeling*, des diskutierten Begriffs können bei jeweils einzelner Betrachtung die Rahmenbedingungen für die UML/P gut abgesteckt werden.

Der Begriff *Modeling* bedeutet, die Vorgehensweise basiert auf einer Modellierungssprache. Obwohl fertiger Java-Code generiert wird, haben die Sprachelemente der UML/P Modellcharakter. Es können also in der UML/P Detailangaben unterlassen oder nur Teilaspekte des gesamten Systems im Modell widergespiegelt werden. Ziel ist es, dem Modellierer so viele Vereinfachungen wie möglich anzubieten, um die Modelle entsprechend verständlich zu machen, und aber auf der anderen Seite immer genug Information zu haben, um mit einer automatischen Vervollständigung lauffähigen Code zu erhalten.

Die zweite Rahmenbedingung findet sich mit dem Begriff *Agile*. Die Sprache soll in einen Entwicklungsprozess eingebunden werden, der einerseits sehr effizient ist und andererseits sehr flexibel sein soll. Gerade hier kann eine gut strukturierte Modellierungssprache ihre Stärken ausspielen. Steht ein Entwicklungsziel noch nicht hundertprozentig fest, so werden nur Teile der fertigen Software mittels Modellen vorgegeben. Es ist nicht mehr nötig, sich bereits in der Entwicklungsphase für einen konkreten Prototypen zu entscheiden, der in allen Details programmiert werden muss. Sollten sich die Anforderungen im Projekt ändern, können des weiteren Modelle meistens einfacher angepasst werden als Code. Der Vorteil ist hier, dass durch die Abstraktion Lösungen zu Problemen explizit in einem Modell dargestellt werden können und nicht, wie in geschriebenem Code, in verschiedenen Programmstücken verzahnt sind.

Um einen schnellen flexiblen Entwicklungsprozess zu gewährleisten, muss ein Programm, welches in UML/P geschrieben ist, möglichst gut strukturiert sein. Nur so weiss ein Entwickler, wo er entsprechende Änderungen und Erweiterungen durchzuführen hat, ohne dabei in Konflikt mit anderen Modellen zu kommen, und ohne dass andere Entwickler erst lange nach den von ihm gemachten Änderungen suchen müssen. Eine gezielte Auswahl einer Teilmenge der UML soll diese Strukturierung unterstützen. Allerdings ist auch mit dieser Teilmenge eine gute Strukturierung nicht garantiert. Es wurde jedoch versucht, die Elemente der UML/P so zu wählen, dass diese möglichst disjunkt und komplementär in ihrer Aufgabe sind. Dazu wurden ebenfalls einige Modifikationen an den Sprachelementen selbst durchgeführt. So wird zum Beispiel aus den Teilen, die noch Java-Code selbst enthalten, jede Möglichkeit genommen, neue Klassen und Attribute zu definieren, da dies bereits mit den Klassendiagrammen realisiert wird (siehe Abschnitt 2.2).

Der in den folgenden Kapiteln beschriebene Prototyp verarbeitet vorerst einmal vier Textnotationen der UML/P. Die Basis für alle weiteren Elemente werden die Klassendiagramme sein (siehe Teil 2.1). Sie werden durch Coderümpfe ergänzt. Mit der Kombination dieser beiden Elemente können bereits in vollem Umfang Programme geschrieben werden. Zu der Sprache werden dann Objektdiagramme hinzugefügt, die über die OCL eingebunden werden. In der Architektur vorbereitet, aber nicht durchgeführt ist der Ausbau der OCL und es können andere Elemente der UML, wie zum Beispiel *State-Charts*, hinzugefügt werden. In diesem Kapitel wird nun erst einmal ein Kondensat der Elemente der UML/P für diese prototypische Version beschrieben.

2.1 Klassendiagramme

In einer objektorientierten Sprache wie Java wird der Code in Form von Klassen strukturiert. Klassendiagramme stellen diese Code-Struktur dar. In ihnen werden die Methodensignaturen und Attribute der Klassen (siehe Tabelle 2.3) und die Zusammenhänge zwischen diesen Klassen festgehalten. Sie bilden somit einen statischen Rahmen für alle weiteren Notationen und Diagrammarten, die sich auf den in Klassendiagrammen definierten Klassenstrukturen stützen. Klassendiagramme sind auch, aufgrund des hohen Bekanntheitsgrades des Konzepts *Klasse*, leicht kommunizierbar. Dank der eben genannten Eigenschaften sind Klassendiagramme ein weit verbreitetes Beschreibungsmittel zur Modellierung von Softwaresystemen.

In Tabelle 2.1 und 2.2 werden die wichtigsten Begriffe aus dem Bereich der Klassendiagramme erläutert. Auf ein paar von ihnen wird im Weiteren genauer eingegangen.

2.1.1 Klassen

Klassendiagramme stellen sowohl die Beziehungen zwischen Klassen als auch die interne Realisierung der Klasse selbst dar. Bei der Klasse werden die Elemente festgehalten, die deren Zustand beschreiben, und ihre Schnittstelle nach außen hin. Der Zustand einer Klasse wird mittels ihrer Attribute definiert. In einem Klassendiagramm wird nicht ein spezieller Zustand festgehalten, sondern nur die Variablen, in

Klasse. Eine Klasse besteht aus einer Sammlung von Attributen und Methoden, die den Zustand und das Verhalten ihrer *Instanzen* (*Objekte*) festlegt. Klassen sind durch Assoziationen und Vererbungsbeziehungen miteinander verknüpft. Ein *Klassenname* erlaubt es, die Klasse zu identifizieren.

Attribut. Die Zustandskomponenten einer Klasse werden als Attribute bezeichnet. Sie beinhalten grundsätzlich *Name* und *Typ*.

Methode. Die Funktionalität einer Klasse ist in Methoden abgelegt. Eine Methode besteht aus einer *Signatur* und einem *Rumpf*, der die Implementierung beschreibt. Bei einer *abstrakten* Methode ist nur die Signatur gegeben.

Modifikator. Zur Festlegung von Sichtbarkeit, Instanziierbarkeit und Veränderbarkeit des modifizierten Elements können die Modifikatoren `public`, `protected`, `private`, `readonly`, `abstract`, `static` und `final` auf Klassen, Methoden und teilweise auf Attribute angewendet werden. Für die ersten vier genannten gibt es in UML/P die graphischen Varianten „`+`", „`#`" und „`-`" und „`?`".

Konstanten sind als spezielle Attribute mit den Modifikatoren `static` und `final` definiert.

Tabelle 2.1: Begriffsdefinition Klassendiagramm, Teil 1 ([Rum02])

denen der Zustand abgelegt ist. Die Schnittstelle einer Klasse nach außen ist in Form von Methoden definiert. Auch hier wird mit Klassendiagrammen nur die Signatur einer Methode, nicht aber eine konkrete Funktionalität vorgegeben.

Attribute

Attribute von Klassen werden im mittleren Feld von Klassendefinitionen angegeben (siehe Tabelle 2.3). Im Vergleich zu Java ist es bei Klassendiagrammen im Sinne der Abstraktion möglich, die Angaben zu einem Attribut unvollständig zu lassen. So kann zum Beispiel bei Unsicherheit über den Typ eines Attributs die Angabe zu diesem unterlassen werden.

Mit der graphischen Darstellung von Klassendiagrammen bietet sich ebenfalls an, einige Eigenschaften in einer kürzeren und vielleicht prägnanteren Form anzugeben. So kann ein Klassenattribut, das in Java mit dem Modifikator *static* versehen ist, durch einfaches Unterstreichen der Angaben wie in Tabelle 2.4 markiert werden. Ebenso bietet die UML Abkürzungen für die Modifikatoren der Zugriffsrechte an. Es steht „+" für *public*, „#" für *protected* und „-" für *private*.

Die UML/P weicht in ein paar Punkten von dem UML Standard ab. Die Zugriffsrechte können mit einem weiteren Modifikator namens *readonly*, kurz „?", definiert werden. Er lässt Lesezugriffe unbeschränkt zu, entspricht dort also *public*, bei Schreibzugriffen verhält er sich aber wie ein *private*-Modifikator. Mit diesem Modifikator

Vererbung. Stehen zwei Klassen in Vererbungsbeziehung, so vererbt die *Oberklasse* ihre Attribute und Methoden an die *Unterklasse*. Die Unterklasse kann weitere Attribute und Methoden hinzufügen und Methoden *umdefinieren* – soweit die Modifikatoren dies erlauben. Die Unterklasse bildet einen *Subtyp* der Oberklasse, der es nach dem *Substitutionsprinzip* erlaubt, Instanzen der Unterklasse dort einzusetzen, wo Instanzen der Oberklasse erforderlich sind.

Interface. Ein Interface (Schnittstelle) beschreibt die Signaturen einer Sammlung von Methoden. Im Gegensatz zur Klasse werden keine Attribute (nur Konstanten) und keine Methodenrümpfe angegeben. Interfaces dienen vor allem zur Definition abstrakter Schnittstellen und sind verwandt zu abstrakten Klassen. Interfaces können untereinander ebenfalls in einer Vererbungsbeziehung stehen.

Typ ist ein Basisdatentyp wie `int`, eine Klasse oder ein Interface.

Interface-Implementierung ist eine der Vererbung ähnliche Beziehung zwischen einem Interface und einer Klasse. Eine Klasse kann beliebig viele Interfaces implementieren.

Assoziation ist eine binäre Beziehung zwischen Klassen, die zur Realisierung struktureller Information verwendet wird. Eine Assoziation wird beschrieben durch einen *Assoziationsnamen*, für jedes Ende einen *Rollennamen*, eine *Kardinalität* und eine Angabe über die *Navigationsrichtungen*.

Kardinalität. Die Kardinalität (Multiplicity) wird für jedes Assoziationsende angegeben. Sie ist von der Form „`0-1`“, „`1`“ oder „`*`“ und beschreibt, ob eine Assoziation in dieser Richtung optional oder eindeutig ist beziehungsweise mehrfache Bindung erlaubt.

Tabelle 2.2: Begriffsdefinition Klassendiagramm, Teil 2 ([Rum02])

erspart man sich oftmals aufwendige Prüfungen bei Schreibzugriffen, wenn ein Attribut für alle lesbar sein soll. Eine weitere Abweichung, im Hinblick auf die zu generierende Zielsprache Java, ist die Verwendung der für Entwickler gewöhnteren Form „Typ attributName“ statt der in dem UML-Standard [Oes01] vorgegebenen Form „attributName: Typ“.

Attributangaben stellen normalerweise einen primitiven Datentyp oder eine einfache Referenzangabe zu einem Objekt dar. Die UML bietet dem Entwickler die Angabe einer Kardinalität zu einem Attribut (siehe Tabelle 2.2). So können Attribute mengenwertig werden, bzw. obligatorisch in einer Klasse vorhanden sein. Angaben zur Kardinalität haben bei Attributen die gleiche Auswirkung wie bei Assoziationen. Eine nähere Betrachtung findet deshalb in Abschnitt 2.1.3 statt.

Der Wert eines Attributs lässt sich oftmals erst zur Laufzeit in Abhängigkeit der Werte anderer Attribute berechnen. Solche Attribute werden als abgeleitete Attribute bezeichnet und graphisch mit einem vorangestellten „/“ markiert. Zum Aus-

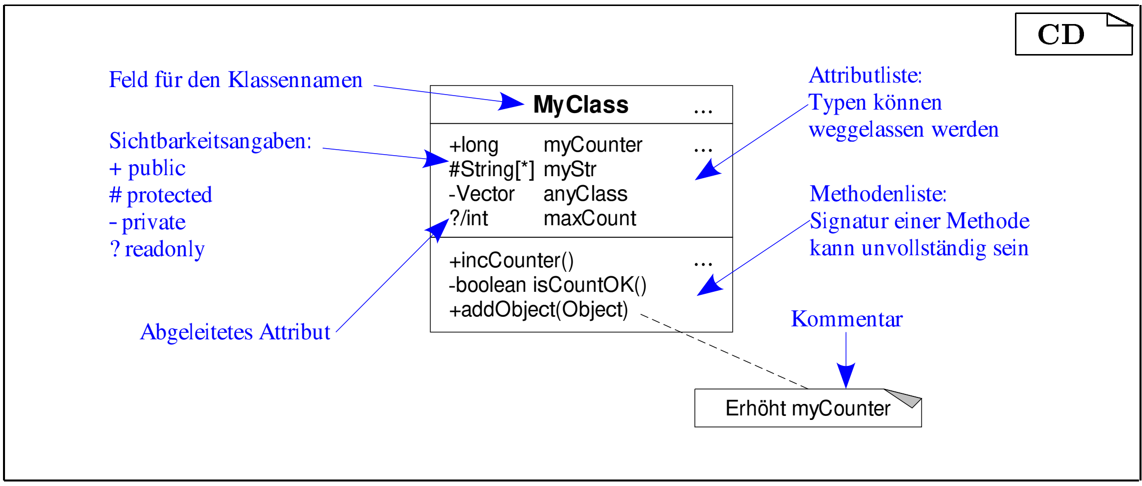


Tabelle 2.3: Muster für eine Klasse im Klassendiagramm

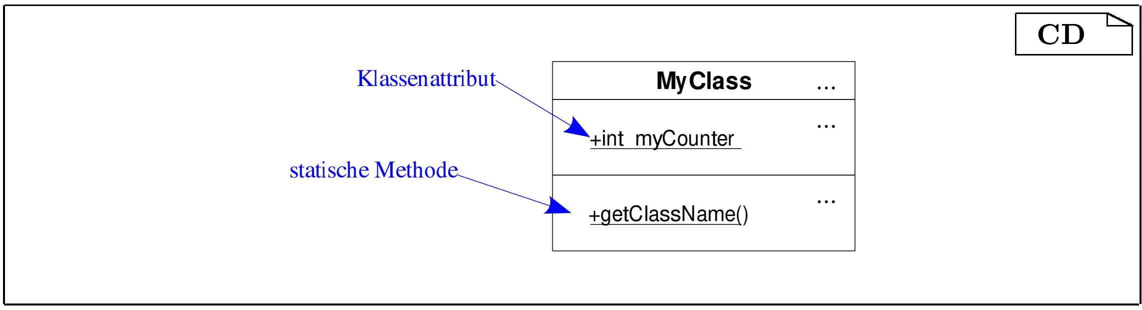


Tabelle 2.4: Klassenattribut und statische Methode

werten können beliebige Berechnungsformeln verwendet werden, die üblicherweise in Java-Ausdrücken oder OCL-Constraints festgehalten werden.

Methoden

Im unteren Feld einer Klassenangabe im Klassendiagramm (siehe Tabelle 2.3) werden die Methoden angegeben. Klassendiagramme enthalten keine Methodenrümpfe, sondern lediglich die Signaturen. Wie bei den Attributen wurde die Schreibweise vom UML-Standard „methodenName (...) : Typ“ auf die Java ähnliche Notation „Typ methodenName (...)“ abgeändert. Auch hier können die Angaben unvollständig sein. So kann man man zum Beispiel Aussagen über den Rückgabetyp bzw. die Übergabeparameter weglassen.

Unterlässt man die Angabe der Übergabeparameter, so tritt eine Zweideutigkeit bei der Interpretation auf. Methoden können in Java überladen werden. Mit anderen Worten, es können zwei Methoden existieren, die den gleichen Namen haben, aber verschiedene Übergabeparameter, und so werden sie wie zwei verschiedene Methoden behandelt. Gibt man nun im Diagramm keine Übergabeparameter an, so kann die Aussage entweder heißen: „Es gibt eine Methode XY, für die gilt ...“ oder „Für alle Methoden XY gilt ...“. In den meisten Fällen ist davon auszugehen, dass Methoden mit gleichem Namen gleiche Eigenschaften besitzen, und so wird im weiteren die zweite Semantik verwendet.

Im Rahmen der graphischen Darstellung können die Sichtbarkeitsangaben mit „+“, „#“ und „-“ angegeben werden. Statische Methoden, die kein spezielles Referenzobjekt zu ihrer Ausführung benötigen, werden graphisch wie Klassenattribute mit Unterstreichung markiert (siehe Tabelle 2.4). Lediglich die Modifikatoren *final* und *abstract* müssen in voller Länge ausgeschrieben werden.

Vererbung und Interfaces

Bei der prozeduralen Programmierung hat man große Prozeduren in mehrere kleine Teilprozeduren aufgeteilt. Diese Vorgehensweise hatte zwei Vorteile. Zum einen wurde der geschriebene Code übersichtlicher, da eine Strukturierung einer Prozedur deutlicher und die Funktionalität der kleinen Prozeduren überschaubarer war. Zum anderen konnte man die einfacheren Funktionalitäten der Prozeduren auch für andere Prozeduren benutzen und so redundanten Code verhindern. Den gleichen Sinn hat der Mechanismus der Vererbung, der sich auf der Ebene der Klassen abspielt. Es kann gemeinsamer Code von Klassen in einer hierarchisch höher gestellten Klasse abgelegt werden und steht dann allen Unterklassen zur Verfügung. So können Klassen in einer Gruppe zusammengefasst werden, die teilweise identische Attribute und Methoden haben (siehe Tabelle 2.5). In Hinblick auf die Generierung von Java wird in UML/P nur Einfachvererbung zugelassen.

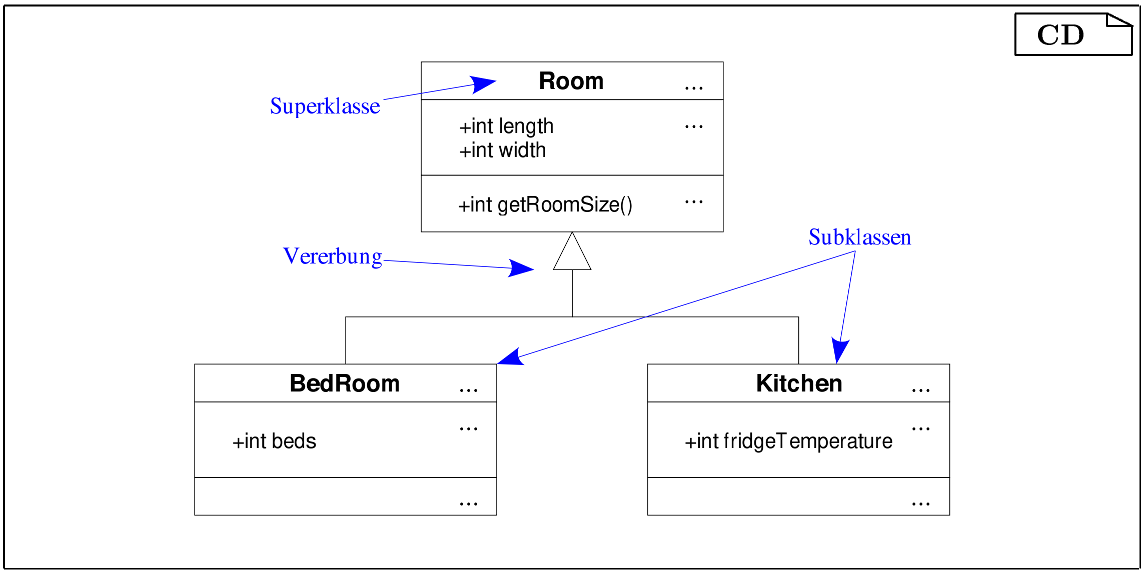


Tabelle 2.5: Vererbung bei Klassen

Neben dem Vererbungsmechanismus können auch Interfaces zur Strukturierung von Code verwendet werden. Etwas größere Klassen bieten meistens sehr viel Funktionalität an, wobei manche Methoden nur einen bestimmten Teil davon benötigen. Interfaces beschreiben mit Methodensignaturen und Konstanten eine Funktionalität einer Klasse. Implementiert eine Klasse ein Interface, so bietet sie mindestens diese Funktionalität an. Eine Methode, die nur eine bestimmte Funktionalität einer Klasse benötigt, kann so mittels des Interface alle Klassen verarbeiten, die dieses Interface implementieren. So wird wiederum redundanter Code verhindert und eine Struktur in der Funktionalität verdeutlicht. Interfaces selbst können in einer Erweiterungshierachie strukturiert werden. Sowohl für die Implementierungen in einer Klasse, als auch für die Erweiterungen der Interfaces selbst können mehrere Interfaces, ähnlich wie bei Mehrfachvererbung, angegeben werden (siehe Tabelle 2.6).

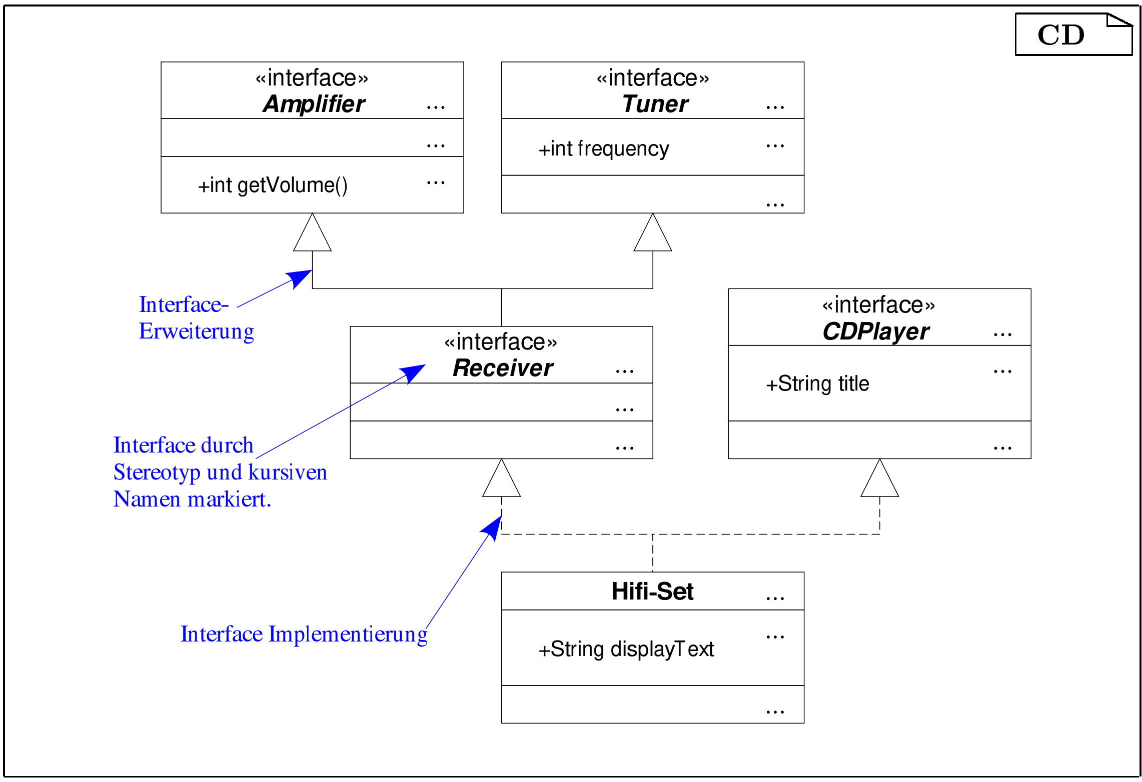


Tabelle 2.6: Implementierung und Erweiterung von Interfaces

2.1.2 Stereotypen und Merkmale

Eine natürliche Sprache wandelt sich im Laufe der Zeit und passt sich so den neuen Anforderungen an das Vokabular an. Basierend auf einer einheitlichen Grammatik verwendet zum Beispiel ein Börsenmakler einen anderen Wortschatz als ein Informatiker. Durch diese Nutzung spezifischer Wörter kann man sich, entsprechend seinem Umfeld, präzise ausdrücken. Ein ähnliches Konzept zur spezifischen Anpassung an Projekte und Anforderungen bietet die UML mit Stereotypen und Merkmalen (siehe Tabelle 2.7). Mit ihnen können in eingeschränkter Weise neue Kontrollstrukturen in die UML eingefügt werden, indem neue Modellierungselemente definiert werden. Ein Paradebeispiel ist der Stereotyp *interface*, der einer Klassenangabe eine Semantik jenseits der herkömmlichen verschafft.

Stereotyp. Ein Stereotyp klassifiziert Modellelemente wie beispielsweise Klassen oder Attribute. Durch einen Stereotyp wird die Bedeutung des Modellelements spezialisiert und kann so beispielsweise bei der Codegenerierung spezifischer behandelt werden. Ein Stereotyp kann eine Menge von Merkmalen besitzen.

Merkmal. Ein Merkmal beschreibt eine Eigenschaft eines Modellelements. Ein Merkmal wird notiert als Paar bestehend aus *Schlüsselwort* und *Wert*. Mehrere solche Paare können zu einer Merkmalsliste zusammengefasst werden.

Modellelement. Modellelemente sind die (wesentlichen) Bausteine der UML-Diagramme. Beispielsweise hat das Klassendiagramm als Modellelemente Klassen, Interfaces, Attribute, Methoden, Vererbungsbeziehungen und Assoziationen. Merkmale und Stereotypen können auf Modellelemente angewandt werden, sind aber selbst keine Modellelemente.

Tabelle 2.7: Begriffsdefinition Merkmal und Stereotyp ([Rum02])

Stereotypen

Stereotypen beschreiben Eigenschaften von Modellelementen. So können, wie im eben genannten Beispiel, *konstruktive* Stereotypangaben wie *interface* eine direkte Auswirkung auf den generierten Code haben. Alternativ können Entwickler auch *deskriptive* Stereotypen zur besseren Übersichtlichkeit einführen. Ein Programmierer könnte mit einem Stereotyp *testClass* ausdrücken, dass eine Klasse lediglich zu Testzwecken genutzt werden soll.

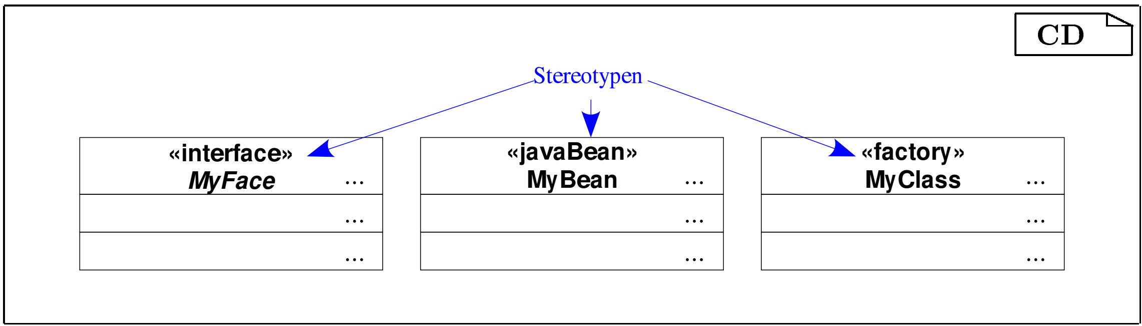


Tabelle 2.8: Stereotypen für Klassen

Stereotypen werden, wie in Tabelle 2.8 zu sehen, vor dem Klassennamen in umgekehrten französischen Anführungszeichen geschrieben. Eine Unterscheidung in der Schreibweise zwischen den Stereotypklassen selbst (konstruktiv, deskriptiv) gibt es nicht. Eine Sonderrolle haben die beiden Stereotypen „©“ und „...“. Sie beschreiben nicht das zu modellierende Element, sondern geben an, ob eine Angabe vollständig ist oder nicht. Dementsprechend werden sie aus der Standardschreibweise herausgenommen und getrennt, in der oberen rechten Ecke der Teilfelder einer Klassenangabe, notiert.

Merkmale

Neben Stereotypen, die hauptsächlich die Modellelemente in Gruppen gliedern, gibt es bei Merkmalen die Möglichkeit zur Angabe von Invarianten zu einem Element. Merkmale bestehen aus einem untypisierten Name-Wert-Paar, in dem als Wert Zeichen- und Ziffernfolgen stehen. So können, wie in Tabelle 2.9 zu sehen, die Projektdaten zu einer Klasse gespeichert werden. Eine Besonderheit stellen Merkmale wie $\{ordered\}$ dar, die keine explizite Wertangabe haben. Implizit ist dieses Merkmal einem $\{ordered = true\}$ gleichzusetzen.

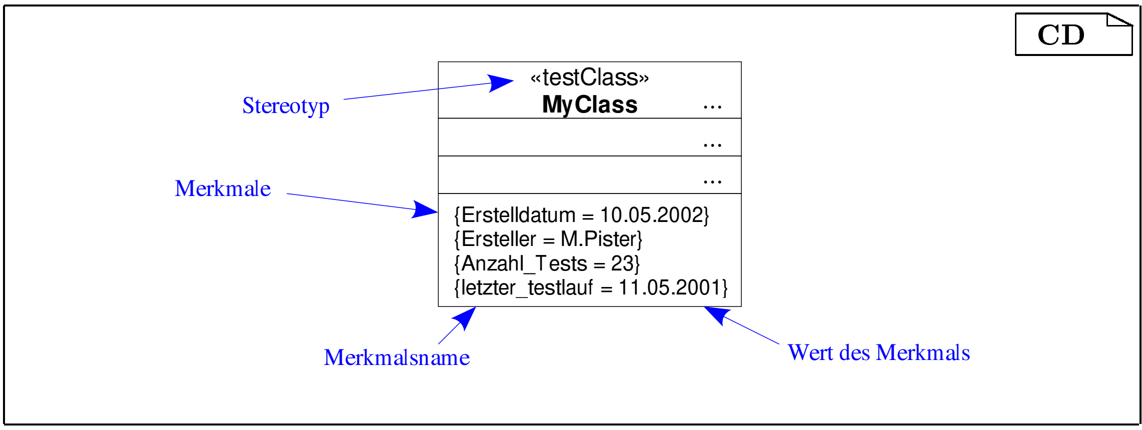


Tabelle 2.9: Merkmale bei Klassen

Merkmale unterscheiden sich deutlich von Attributen. So sind Merkmale nicht für Zugriffe im generierten Code erfassbar und ändern ihre Werte nicht zur Laufzeit des beschriebenen Programms. Sie können lediglich Einfluss auf die Codegenerierung haben. Merkmale werden oft dazu benutzt, Projektinformationen zu Klassen zu speichern, Kommentare anzuhängen oder Rahmenbedingungen anzugeben, die nicht im Sprachumfang der Standardsprache enthalten sind.

2.1.3 Assoziationen

Neben den Angaben zu den Klassen selbst werden in Klassendiagrammen auch Angaben zu den Relationen zwischen ihnen gemacht. Diese Relationen werden, wie in Tabelle 2.10 zu sehen, als Assoziationen dargestellt und drücken so eine Referenzierung von einer Klasse zu einer anderen aus. Mit Assoziationen können komplexe Datenstrukturen gebildet werden, innerhalb derer sich die Objekte mit Assoziationselementen gegenseitig referenzieren können. Um innerhalb der Datenstuktur zu navigieren, können den Assoziationen Namen gegeben werden bzw. an deren Enden Rollennamen zum Zugriff angegeben werden.

Assoziationen können in zwei große Gruppen eingeteilt werden. Es gibt die unidirektionalen Assoziationen, die von ihrem Verhalten Attributen entsprechen, und die bidirektionalen Assoziationen. Bei Zweiteren referenzieren sich die beiden Objekte eines Assoziationselements gegenseitig. Eine Änderung eines Assoziationselements an einem Objekt zieht somit immer eine implizite Änderung des jeweiligen Partner-

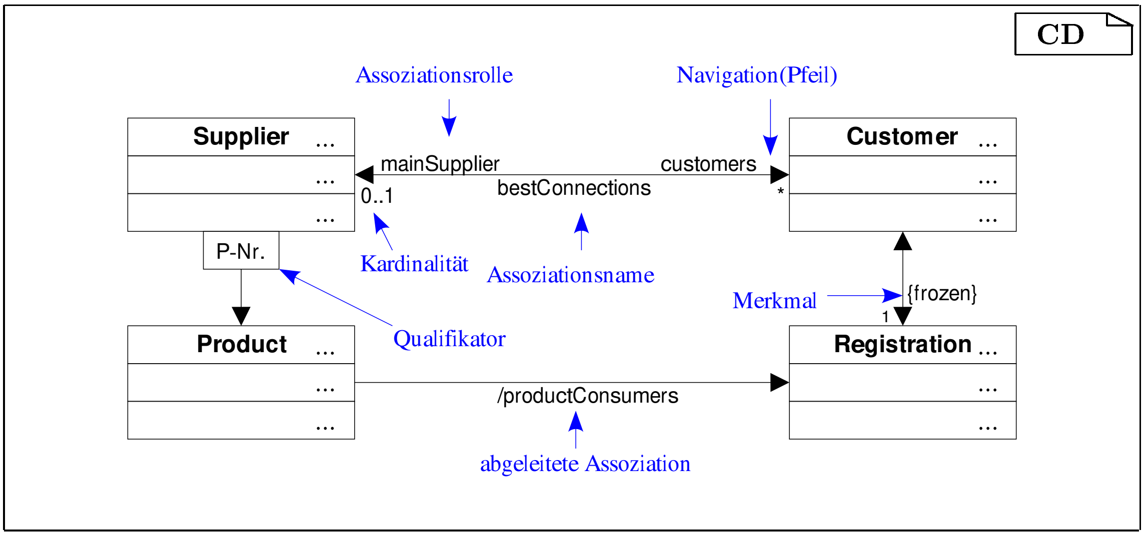


Tabelle 2.10: Assoziationen

objekts mit sich. Ein besonderer Fall sind reflexive bidirektionale Assoziationen, bei denen Rollennamen zu einer einwandfreien Navigation immer nötig sind.

Wie bereits bei den Attributen erwähnt, können auch Assoziationsenden mit einer Kardinalität versehen werden. So entspricht die Angabe [0..1] einem herkömmlichen Eintrag ohne weiteren Zusätzen. Er kann einen beliebigen Wert haben, oder *null* sein. Mit dem Merkmal *frozen* kann ein Assoziationseintrag, wenn er einmal erstellt ist, nicht mehr verändert werden. Bei einer unidirektionalen Assoziation verhält sich ein derartiger Eintrag nach seiner Initialisierung wie ein *final*-Attribut. Eine solche Eigenschaft wäre zum Beispiel für Initialisierungsroutienen interessant.

Manchmal möchte man im Code nicht überprüfen müssen, ob ein Eintrag existiert, also nicht *null* ist. Mit der Kardinalitätsangabe [1] legt man eine permanente Belegung des Eintrages fest. Man kann also diesen Eintrag immer dereferenzieren. Eine besondere Form der [1]-Kardinalität ist der Zusatz mit dem Merkmal *frozen*. Ein Objekt kann so permanent und fest an ein anderes Objekt gebunden werden. Jedes Objekt „Customer“ in Tabelle 2.10 hat zum Beispiel immer eine Registrierung, die ihm lebenslang zugeordnet ist.

Außerdem gibt es noch die Kardinalitätsangabe [*], die einem Assoziationsende eine Menge von Objekten zuweist. So kann ein Objekt mit einer beliebig großen, typisierten Menge von anderen Objekten verbunden sein. Mit dem Merkmal *addOnly* wird das Löschen von Einträgen verhindert, was zum Beispiel bei einer Protokollklasse sinnvoll sein könnte. Standardmäßig werden mit [*] markierte Enden mengenwertig, d.h. ohne Sortierung, realisiert. Mit dem Merkmal *ordered* werden statt Mengen Listen verwendet, auf denen eine Ordnung existiert. Mit der bloßen Angabe von *ordered* wird die zeitliche Reihenfolge der Einträge gesichert. Oft möchte der Anwender jedoch eine eigene Ordnung auf den Elementen definieren, bzw. in einer ungeordneten Menge gezielte Zugriffe auf die Elemente machen können. Zu diesem Zwecke wird die qualifizierte Assoziation eingeführt, bei der jedem Eintrag beim Erstellen ein Qualifikator zugewiesen wird, der dann zur Abfrage als Schlüssel verwendet werden kann. So können zum Beispiel bei einer Produktliste die Produkte über ihre Produktnummern referenziert werden.

Wie bei den Attributen gibt es auch bei Assoziationen die Variante der abgeleiteten Assoziation. Mit ihr können Assoziationen dargestellt werden, die sich aus anderen Assoziationen und Attributen zusammensetzen. Ein Beispiel ist die Assoziation „productConsumer“ in Tabelle 2.10, die die hauptsächliche Kundengruppe bestimmt, die ein Produkt erreichen kann. Abgeleitete Assoziationen werden mit einem vorangestellten Schrägstrich (“/“) markiert. Zum Bestimmen einer Ableitungsvorschrift bietet sich auch hier die Verwendung von OCL (siehe Abschnitt 2.3) an.

2.2 Coderümpfe

Hat man mit Klassendiagrammen deren Schnittstelle nach außen hin deklariert, so fehlt noch eine Beschreibung der Funktionalität der Methoden. Das eigentliche Ziel, auch hier eine Modellierungssprache zu verwenden, scheitert oftmals an den Möglichkeiten bzw. dem Aufwand, daraus lauffähigen Code zu generieren. Die beste Lösung ist daher, die Funktionalität in textueller Form, also mit Java-Code zu beschreiben. Dieser muss allerdings nicht direkt in der Klassenbeschreibung des Diagramms stehen, sondern kann gesondert abgelegt werden. Der Vorteil liegt dabei in der freien Strukturierungsauswahl des Quellcodes. Es können Methoden neben einem Klassendiagramm stehen, aber auch getrennt in einer Datei, die eine bestimmte Funktionalitätsgruppe kapselt. Die Zuordnung einer Methode zu einer Klasse läuft ähnlich der bisherigen Zuordnung in Java.

Java

```
java-code {
  class mypackage.MyClass {
    int myMethod (String s) {
      /* Hier steht jetzt dann der Methodenrumpf */
      ...
      return s;
    }
  }
}
```

Im Vergleich zu echtem Java-Code fehlt das „package“-Statement und außerdem wird der Klassenname mit einem Pfadnamen angegeben. Das Verlagern der Paket-Zugehörigkeit zu den Klassen lässt zu, mehrere Methoden von verschiedenen Klassen nebeneinander zu schreiben.

Die Methodenrümpfe sind dem bekannten Java sehr ähnlich. Es muss jedoch eine Einschränkung beachtet werden. Es ist nicht möglich irgendeine Art von *inner Classes* zu erzeugen. Innere Klassen sind Klassen, die nur innerhalb einer Methode sichtbar sind. Es sind oftmals kleine Helferklassen, mit denen zum Beispiel Interfaces implementiert werden. Auch wenn diese inneren Klassen meist sehr klein und sehr einfach aufgebaut sind, macht es keinen Sinn, diese vor anderen Methoden zu verstecken. Benötigt eine andere Methode nun aus irgendwelchen Gründen eben genau die gleiche Klasse, so muss diese innere Klasse nochmals in dieser anderen Methode implementiert werden. Redundantes Festhalten von Code entspricht jedoch nicht den *Coding-Standards*. Ein weiterer Grund, die inneren Klassen aus den Coderümpfen zu nehmen, ist die disjunkte Trennung von den Klassendiagrammen. Ein Entwickler sollte, wann immer er eine Klasse suchen oder erstellen will, auf die Klassendiagramme zurückgreifen. So muss er bei einer Klassensuche nie zusätzlich die Coderümpfe durchsuchen.

Java 1.4 bietet bereits eine Möglichkeit, zu Testzwecken eine *assert*-Anweisung (siehe [Fla02] S.60 ff) zu nutzen. Mit dieser können Bedingungen geprüft werden, die bei einer Nichteinhaltung mit einer *Exception* den Programmlauf der Methode abbrechen und an gegebener Auffangstelle den Lauf fortsetzen. Assert-Anweisungen bringen im laufenden Endprogramm leider nichts, denn dort muss die Bedingung immer wahr sein, aber in Testphasen, in denen der Code noch in Entwicklung ist, können so schnell Fehlerquellen aufgespürt werden. In den in diesem Prototyp gegebenen Coderümpfen, die nach Java 1.2 umgesetzt werden, ist ebenfalls eine assert-Anweisung zulässig, die in Testläufen zu einem Abbruch führen kann.

2.3 OCL

In Klassendiagrammen betrachtet man Klassen als eine Art Implementierungsbeschreibung. Es werden die Attribute beschrieben, die die Daten der Objekte und somit deren Zustand festhalten. Ebenso wird notiert, welche Methodensignaturen und Assoziationen implementiert werden müssen. Nutzt man dann noch Coderümpfe als Ergänzung, ist eine Implementierungsvorgabe komplett. Die Nutzung der *Object Constraint Language* (OCL), also einer Eigenschaftssprache, geht mit einer etwas anderen Sichtweise an den Begriff Klasse heran. Eine Klasse wird hier als eine *Extension* gesehen, also als eine Menge von Objekten (siehe Tabelle 2.11). Beschrieben wird diese Menge von Objekten mit *Invarianten* und *Methodenspezifikationen*, den *Constraints* der OCL, die immer für ein Objekt dieser *Extension* gelten müssen. Diese Invarianten können auch Zusammenhänge zwischen mehreren Klassen festhalten, die immer gelten müssen, wenn ein Objekt einer Klasse existieren soll.

2.3.1 OCL im UML/P-Konzept

In Klassendiagrammen werden bei Attributen, vorausgesetzt sie sind nicht abgeleitet, keine Aussagen über deren möglichen Belegungen im Verlauf eines Programms gemacht. Mit der OCL kann die Menge der Belegungen der Attribute mittels einer gegebenen Bedingung in Abhängigkeit des Systemzustandes begrenzt werden. Ebenso werden in Klassendiagrammen nur die Signaturen der Methoden festgehalten, womit aber nichts über die Funktionalität der Methoden gesagt ist. Mit der OCL kann mittels Bedingungen die Menge der Funktionalitäten, die eine Methode haben kann, eingeschränkt werden. Die OCL kann so als eine Ergänzung zu Klassendiagrammen gesehen werden, mit der eine Verfeinerung dieser groben Modellierung der Diagramme realisiert werden kann.

In Bezug auf Coderümpfe hat die OCL nicht die Aufgabe, wie bei Klassendiagrammen, die Funktionalität einer Methode einzuschränken, da diese bereits eindeutig gegeben ist. Sie dient hier zum Herausheben von Eigenschaften der Funktionalität einer Methode, da Coderümpfe von sich aus meist schwer lesbar sind und keine Abstraktionsfähigkeit besitzen. Der generierte Programmcode wird so weiterhin von den Coderümpfen bestimmt. Da die OCL aber oft leichter zu verstehen ist und mehr Modellierungscharakter hat, wird sie dazu genutzt, um die Eingenschaften der gegebenen Methodenfunktionalität abstrakt zu beschreiben. Die OCL wird in Tests umgewandelt, die, insofern möglich, die generierte Methode testen und somit kontrollieren, ob eine mit OCL gegebene Eigenschaft tatsächlich im Coderumpf umgesetzt wurde (siehe Abschnitt 6.2.2). Sollten in den Tests Abweichungen bei der Umsetzung erkannt werden, so muss der Modellierer den geschiebenen Coderumpf anpassen, bzw. das Constraint ändern.

In Kombination mit abgeleiteten Attributen wird die OCL ähnlich wie bei Coderümpfen verwendet. Eine Ableitungsvorschrift kann entweder ein Java-Ausdruck sein, oder eine Invariante in Form von OCL. Allerdings ist die Ableitungsvorschrift von abgeleiteten Attributen meist eindeutig und so kann bei der Verwendung von OCL meist eine Berechnungsvorschrift durch Java ersetzt werden und umgekehrt. In diesen Fällen liegt es am Entwickler, ob er eine eindeutige Berechnungsvorschrift lieber in Java oder in OCL schreibt. Es werden hier voraussichtlich beide Möglichkeiten erhalten bleiben.

2.3.2 Anwendungen der OCL

Die hier in diesem Implementierungvorschlag verwendete OCL/P deckt nur eine kleine Teilmenge der gesamten OCL/P ab. Mit dem Ziel, möglichst schnell einen ersten Prototypen implementieren zu können, wird nur der nun folgende kleine OCL/P-Anteil betrachtet. Basierend auf dem in dieser Arbeit realisierten Teil kann die OCL/P dann voll ausgebaut werden.

Mit *OCL-Constraints* werden logische Bedingungen ausgedrückt. Diese Bedingungen müssen jedoch innerhalb eines Kontextes ausgedrückt werden, dass heißt, es muss klar sein, auf welche Elemente sich eine Bedingung bezieht. Der Aufbau eines *OCL-Constraints* sieht daher vor, zuerst einen Kontext zu definieren, in dem eine OCL-Bedingung gültig sein soll. Möchte man zum Beispiel sagen, dass in einer Klasse XY ein Attribut a immer ungleich 5 sein soll, so definiert man dies mit folgendem Constraint:

```
context XY xy inv:                                    OCL
  !xy.a.equals(5);
```

Ist der Kontext mittels der Anweisung „context“ gesetzt, so können die nun folgenden Bedingungen beliebig auf die dort definierten Namen zugreifen. Wie man sehen kann, steht nach der Kontextanweisung das Schlüsselwort „inv“. Dies ist eine Abkürzung für invariant und die darauf folgende Bedingung muss im gegebenen Kontext also immer *true* sein. Die OCL bietet drei Schlüsselworte für die Gültigkeit von Bedingungen an. Neben dem eben genannten „inv“, welches immer gilt, können auch mit den Schlüsselwörtern „pre“ und „post“ Eigenschaften der Funktionalität von Methoden beschrieben werden. So soll folgendes *OCL-Constraint* eine Wurzel-Methode beschreiben:

```
context MyNumber MyNumber.squareRoot(MyNumber n1)          OCL
  pre: n1 > 0
  post: (result * result) == n1
```

Mit dem *pre-Constraint* werden alle Zustände beschrieben, von denen einer vor Aufruf der Methode gelten muss, damit nach Ausführung der Methode das *post-Constraint* gilt. Es ist also mit der OCL möglich, eine Aussage zu einer Funktionalität auf eine Menge von Anfangszuständen zu beschränken. So wird in diesem Beispiel eine Aussage über das Ziehen einer Wurzel im positiven Bereich getroffen. Sollte die Eingabe aber negativ sein, so ist alles offen. Die *post*-Bedingung gilt immer nach Ausführung der Methode unter der Voraussetzung der *pre*-Bedingung. Eine Besonderheit ist hierbei das Schlüsselwort „result“, das das Ergebnis der Methode nach ihrer Berechnung repräsentiert.

Die erste Version der OCL/P ist hier bereits abgeschlossen. Die verwendeten Bedingungen sind vorerst Java-Ausdrücke, die einen boolschen Wert liefern müssen. Die volle Version der OCL/P wird später eine eigene *OCL-Expression* enthalten, die spezielle Elemente zu boolschen Aussagen bietet. Ein kleine Sonderanwendung findet sich in Abschnitt 2.4.3, in dem Objektdiagramme eingebunden werden.

2.4 Objektdiagramme

Stellt man mit Klassendiagrammen die Struktur der Klassen dar, so macht man statische Aussagen zu einer Softwarekonstruktion. Manchmal ist es hilfreich, auch einzelne Zustände der Software, also eine Belegung der Attribute und Assoziationen, mit konkreten Werten festzuhalten. Sinnvoll wäre dies zum Beispiel, um den Startzustand eines Systems oder Systemteils zu bestimmen. Man kann ebenso in Kombination mit OCL die Funktionalität einer Methode beschreiben, indem man als Vor- bzw. Nachbedingung ein Objektdiagramm angibt, welches dann gelten muss.

Im Vergleich zu den anderen Elementen der UML/P sind Objektdiagramme immer nur für einen bestimmten Zeitraum, und nicht permanent gültig. Es können also mehrere verschiedene Objektdiagramme ein und denselben Systemteil zu verschiedenen Zeiten beschreiben. Objektdiagramme bieten dem Entwickler eine sehr gute Darstellung der jeweiligen Zustände und können gerade in der OCL wesentlich besser lesbar sein, als eine Beschreibung mittels eines Ausdruckes.

In Tabelle 2.11 werden die wichtigsten Begriffe aus dem Bereich Objektdiagramme kurz erläutert. Wie auch bei Klassendiagrammen wird im Folgenden auf ein paar von ihnen genauer eingegangen.

Objekt im Objektdiagramm. Ein Objekt ist Instanz einer *Klasse* und enthält die von ihr festgelegten *Attribute*. Diese Attribute sind mit einem Wert initialisiert (ggf. noch uninitialisiert). Im Objektdiagramm werden *prototypische Objekte* verwendet, um exemplarische Situationen zu illustrieren. Zwischen den im Diagramm sichtbaren *prototypischen Objekten* und den echten Objekten des Systems besteht normalerweise keine 1:1-Beziehung.

Objektname erlaubt die eindeutige Benennung des Objekts im Objektdiagramm. Als Objektname wird ein sprechender Name gewählt, der normalerweise im echten System nicht wiederzufinden ist, da dort systemspezifische Objektidentifikatoren verwendet werden.

Attribut beschreibt eine Zustandskomponente eines Objekts. Ein Attribut im Objektdiagramm ist charakterisiert durch den Attributnamen, den Typ und einen konkreten Wert. Weitere Charakteristika, wie Sichtbarkeit, können dem Attribut angeheftet werden. In *abstrakten Objektdiagrammen* können statt konkreten Werten auch Variablennamen oder Ausdrücke eingesetzt werden, deren Inhalt im Diagramm „unspezifiziert" bleibt. Attributtyp oder -wert können auch ausgelassen werden.

Link ist eine Ausprägung einer Assoziation zwischen zwei Objekten, deren Klassen jeweils an der Assoziation partizipieren. Navigationsrichtung, Assoziations- und Rollennamen können auf dem Link ebenfalls dargestellt werden.

Kompositionslink ist eine Sonderform des Links, bei dem neben der reinen Verbindung weitere Abhängigkeiten des Teilobjekts vom Ganzen bestehen. Ein Kompositionslink ist eine Ausprägung einer Komposition.

Tabelle 2.11: Begriffsdefinition Objektdiagramm ([Rum02])

2.4.1 Objekte

Objektorientierte Sprachen wie Java basieren, wie der Name bereits sagt, auf Objekten, also gekapselten Speichereinheiten. Der Zustand der Software kann durch ein Festhalten der Belegung dieser Speichereinheiten bestimmt werden. Die Beschreibung eines Objekts ist hier, analog zur Klasse in Klassendiagrammen eines der Kernelemente.

Wie in Tabelle 2.12 zu sehen, gibt es bei Objektdiagrammen lediglich eine einfache Teilung in zwei Felder. Das obere Feld enthält optional den Typ und den Namen des Objektes, wobei der Typ mit einem vorangestellten Doppelpunkt markiert ist. Die Angabe des Typs ist jedoch obligatorisch, wenn er aus dem Kontext des Objektes nicht abgeleitet werden kann. Der Objektname tritt in der tatsächlichen Implementierung nie auf, da die Objekte dort mittels nahezu kryptischer Objektreferenzen benannt werden. In dem Diagramm kann er allerdings bei einer späteren Einbindung von OCL als Referenz genutzt werden.

In Objektdiagrammen werden prototypische Objekte angegeben. Es ist also nicht immer nötig, dass ein Objekt in exakt der angegebenen Form tatsächlich existiert. Diese Anschauung der Objekte verschafft eine zusätzliche Freiheit bei der Typangabe. Es können neben Standardklassen auch abstrakte Klassen und Interfaces als Typ angegeben werden, die im lauffähigen Code durch andere, sie vervollständigende Klassen ersetzt werden müssen.

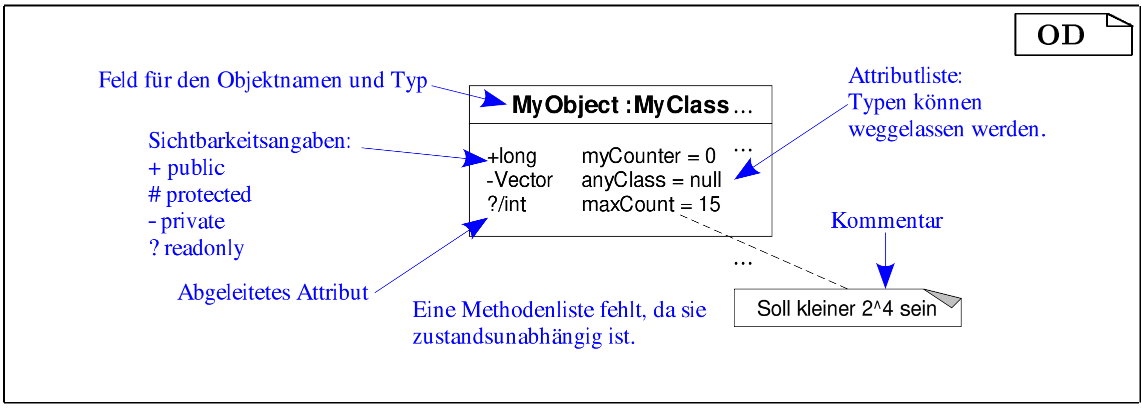


Tabelle 2.12: Muster für ein Objekt im Objektdiagramm

Das zweite Feld einer Objektangabe gibt die Attributbelegungen an. In ihm können den Attributen Werte zugewiesen werden. Dabei kann, wie bei Klassendiagrammen die Angabe zu den jeweiligen Attributen unvollständig sein. Die Wertzuweisung kann sogar fehlen, sollte diese zum Beispiel in späteren Versionen der Sprache durch OCL ersetzt werden. Im Unterschied zum Attributfeld von Klassen sind in dem von Objekten auch die Attribute von Oberklassen enthalten, da ein Objekt die gesamte Hierachie nach oben hin präsentiert.

Bei der Angabe von Attributwerten sind einige Regelungen zu beachten. Klassenattribute müssen in jedem Objekt den gleichen Wert haben, denn sie sind an die Klasse und nicht an das Objekt gebunden. Oft eignet sich hier für eine Wertangabe ein Ausdruck in einer OCL-Bedingung. Ein anderer kritischer Punkt ist die Angabe von Werten bei abgeleiteten Attributen. Gibt man den Wert eines abgeleiteten Attributs an, so müssen die Angaben zu den wertbestimmenden Attributen mit der Ableitungsvorschrift vereinbar sein.

Wie unter 2.1.1 bei den Klassendiagrammen erwähnt, verhalten sich mengenwertige Attribute wie unidirektionale Assoziationen. Die Wertangabe ist in einer Objektdeklaration selbst relativ unübersichtlich. Möcht man Werte von derartigen Attributen angeben, so wird dies genau wie bei der Angabe von Links gestaltet, die im nächsten Teil beschrieben werden.

2.4.2 Links

Hat man die inneren Werte einer Klasse beschrieben, so bleiben noch die Zustände der Assoziationen offen. Ein instanziiertes Element einer Assoziation nennt man einen *Link*. Die meisten Angaben zu Links, wie zum Beispiel Navigationsrichtung und Merkmale, können aus den Assoziationsangaben der Klassendiagramme geholt werden. Werden solche Angaben bei Links gemacht, so müssen sie mit der Assoziationsvorgabe übereinstimmen.

Die Links müssen einer Assoziation im Klassendiagramm zugewiesen werden können. Dazu kann der Name der Assoziation oder, falls eindeutig, der Rollenname innerhalb der Assoziation verwendet werden. Sollte zwischen zwei Klassen nur eine Assoziation existieren, dann ist jeder Link zwischen diesen Klassen eindeutig zuweisbar und der Name kann weggelassen werden. Um Fehler beim Hinzufügen neuer Assoziationen zu vermeiden, sollte jedoch immer ein Name angegeben werden.

Links von abgeleiteten Assoziationen müssen in einem Objektdiagramm nicht speziell markiert sein. Wie bei den Attributen auch, muss die Angabe von solchen Links mit der Ableitungsvorschrift vereinbar sein. Auch hier ist es für das Verständnis hilfreich, den Namen des Links einer abgeleiteten Assoziation mit einem vorangestellten Schrägstrich zu versehen.

Links von qualifizierten Assoziationen fordern Angabe eines Qualifikators. Dies kann auf zwei Weisen geschehen. Einerseits kann der Qualifikator, wie in Abbildung 2.13 direkt bei dem Link stehen. Es kann aber auch der Name eines Attributs in der Klasse am anderen Assoziationsende angegeben werden, dessen Wert im jeweiligen Objekt als Qualifikator fungiert. Ein Sonderfall stellen jene qualifizierten Assoziationen dar, die mit dem Merkmal *ordered* versehen sind. Bei ihnen ist nicht explizit ein Qualifikator angegeben. Da aber im Objektdiagramm eine Reihenfolge ersichtlich sein muss, wird dem Link als Qualifikator eine ganze positive Zahl zugeordnet, die die Position in der Reihenfolge wiedergibt.

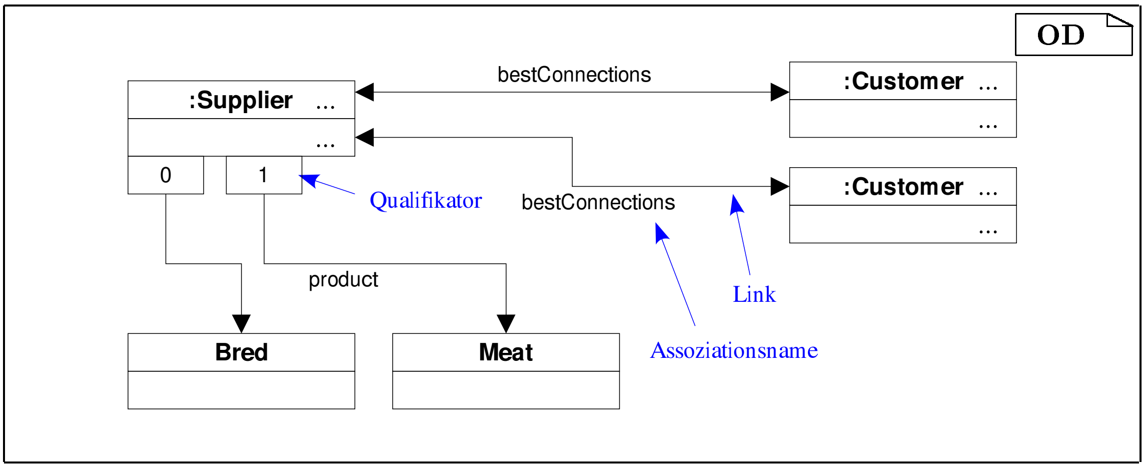


Tabelle 2.13: Links in Objektdiagrammen

Den Links kann im Objektdiagramm ein weiteres Merkmal namens *frozen* angehängt werden, welches unabhängig von den Merkmalen der Assoziation ist. Mit diesem

Merkmal kennzeichnet man einen Link, der solange lebt, wie die beiden Objekte an dessen Enden leben. Dieses Merkmal dient zur vereinfachten Darstellung von OCL-Ausdrücken mit gleicher Aussage.

2.4.3 Einbindung in bisherige Elemente der UML/P

Objektdiagramme stellen keinen neuen Konstruktionsmöglichkeiten her. So ist alles, was mit Objektdiagrammen ausgedrückt werden kann, auch mit OCL ausdrückbar. Trotzdem ist die Verständlichkeit und Lesbarkeit von Objektdiagrammen wesentlich besser als die von OCL-Ausdrücken. Objektdiagramme können also genutzt werden, um die OCL-Ausdrücke zu ersetzen, die einen Systemzustand beschreiben.

Um Objektdiagramme in die OCL einzubinden, müssen diese referenziert werden. Objektdiagramme können deshalb mit einem Namen versehen werden, der im gesamten UML/P-Modell sichtbar ist. Möchte man nun in eine OCL-Aussage ein Objektdiagramm einbinden, dann verwendet mann einen Ausdruck mit dem Schlüsselwort „OD." gefolgt von dem Diagrammnamen. Ein *OCL-Constraint*, welches auf Objektdiagramme zugreift, könnte so aussehen:

```
context int MyClass.myMethod()                                    OCL
  pre: OD.myDiagram1
  post: ( OD.myDiagram2 && ( result == 0 ) )
```

In den Bedingungen muss der beschriebene Zustand des Diagramms „myDiagram1" bzw. „myDiagram2" gelten. Dabei ist zu berücksichtigen, dass sowohl die Objekte, als auch die Links prototypischen Charakter haben. Das heißt, ein Objektdiagramm muss nicht 1:1 in einem laufenden System existieren. Es kann zum Beispiel kein Objekt mit dem Typ eines Interfaces existieren, sehr wohl aber eines mit dem Typ einer implementierenden Klasse. Unter Berücksichtigung einiger Aspekte wie dem eben genannten Interface-Problem, wird ein Zustand beschrieben, in dem wenigstens die direkt angegebenen Eigenschaften der Objekte und Links vorhanden sein müssen. Das Diagramm ist auch gültig, wenn im System Objekte und Links existieren, die nicht in ihm aufgeschrieben sind.

In späteren Versionen des Prototyps bietet es sich an, die OCL auch zur Beschreibung von Eigenschaften in Objektdiagrammen zu nutzen. So könnten mit der OCL zum Beispiel Aussagen über die Menge einer Objektsorte im Verhältniss zu einer anderen gemacht werden. In dieser ersten prototypischen Version der UML/P wird jedoch auf diese Option verzichtet.

Kapitel 3

Das Werkzeug zur Codegenerierung

Wie in der Einleitung erläutert, wird in dieser Arbeit eine prototypische Implementierung für ein Werkzeug gegeben, welches aus UML/P lauffähigen Java-Code generiert. Dieses Werkzeug ist dabei speziell für das Agile Modeling, also dem Einsatz in schnelllebigen Enwicklungsprozessen, geeignet. Um im Verlauf dieser Diplomarbeit Verwechslungen zu vermeiden, wird das Werkzeug im weiteren „Agile-Modeling-Tool“ (*AMTool*) genannt.

In Kapitel 2 wurden diejenigen Elemente der Sprache UML/P vorgestellt, die das AMTool verarbeitet. Dieses Kapitel beschäftigt sich nun mit dessen Implementierung. Es werden zuerst einige besondere Anforderungen an das AMTool erläutert, die bei der Entwicklung auf jeden Fall berücksichtigt werden müssen. Danach wird eine grobe Spezifikation gegeben, die diese erfüllt. Letzlich wird noch der Aspekt der Qualitätssicherung behandelt, der keine direkten Auswirkungen auf die Funktionalität, wohl aber auf die Entwickelbarkeit des AMTools selbst hat.

3.1 Anforderungen an das Werkzeug

Die Anforderungen an das AMTool hängen eng mit dem Umfeld der agilen Entwicklungsprozesse zusammen. So muss zum Beispiel das AMTool, wenn es neu in einen solchen Prozess eingeführt wird, einfach mit den bisher verwendeten Werkzeugen kombinierbar sein. Ebenso muss eine Anpassung der Modelle und des Werkzeugs an neue Gegebenheiten ohne großen Aufwand vollzogen werden können. Im Folgenden werden einige spezielle Anforderungen herausgegriffen und näher erläutert.

3.1.1 Textuelle Eingabe

Das AMTool soll UML/P-Modelle in einer Textdarstellung aus ASCII-Zeichen verarbeiten können. Einige Elemente der UML/P, die OCL und die Coderümpfe, sind ohnehin in einer textuellen Form gegeben. So ist der Aufwand bei der Erstellung des Prototypen geringer, wenn für die graphischen Elemente eine Textnotation entworfen wird, da diese wesentlich einfacher zu bearbeiten ist. In diesem Abschnitt werden nun drei weitere Aspekte betrachtet, die für eine Darstellungsmöglichkeit der Klassendiagramme und Objektdiagramme in Textform sprechen.

Die vorläufige Zielgruppe des AMTools sind Teams, die bereits in Java entwickeln. Oft arbeiten diese mit textbasierten Entwicklungsumgebungen, was zur Folge hat, dass einige Entwickler nicht an das Programmieren mit graphischen Werkzeugen gewöhnt sind. Um ihnen die Umstellung auf die Nutzung von Modellen zu erleichtern, sind diese mit einer eng an Java angelehnten Textnotation beschreibbar.

Eine textbasierte Notation der UML/P lässt sich darüber hinaus gut mit fast allen Werkzeugen kombinieren, die bei der Programmierung von Java verwendet werden. Aus diesem Grund bieten auch die graphisch arbeitenden Werkzeuge fast immer eine Möglichkeit zu einer Speicherung im Textformat an. Der Vorteil ist, dass so viele von Java unabhängige Werkzeuge genutzt werden können, die nur mit ASCII-Zeichenfolgen arbeiten. Ein Beispiel ist *CVS*[1], welches zur Abstimmung des Codes bei paralleler Programmierung dient und in dem der Entwickler im Falle von Problemen von Hand den kritischen Textteil korrigiert.

Ein graphisches Modell ist durch die zweidimensionale Darstellung seines Inhaltes begrenzt. Komplexe Sachverhalte können oft nur sehr schwer oder gar nicht aufgezeichnet werden. Bei textuellen Notationen gibt es diese Beschränkungen nicht. So lassen sich anspruchvollere Modelle auch in Textform gut darstellen. Textnotationen können bei der UML/P inhaltlich zudem alle graphisch beschriebenen Eigenschaften erfassen. Sie verlangen aber von dem Betrachter, dass er sich in einem Text relativ gut orientieren kann. Sie sind bei einfacher strukturierten Modellen bei weitem nicht so anschaulich wie eine Graphik. Gerade bei der Kommunikation mit fachfremden Personen sind Textdarstellungen meistens sogar ungeeignet. Die beiden Darstellungen schließen sich also nicht aus, sondern ergänzen sich gegenseitig.

3.1.2 Manuelle Fehlerbehebung in den Modellen

Wie bereits in der Einleitung erwähnt, gibt es im Bereich modellgestützter Entwicklung Software, die nicht nur Top-Down von den Modellen auf Code abbildet, sondern auch in umgekehrter Weise Modelle aus dem Code konstruiert. Ebenso könnten beim Erstellen eines Modells die anderen Modelle automatisch angepasst werden. Wird zum Beispiel ein neues Attribut in einem Objektdiagramm hinzugefügt, so könnte es automatisch bei allen Klassendiagrammen nachgetragen werden. Dieser Ansatz ist bei kleinen Systemen durchaus nützlich, bringt aber in großen Systemen einen eventuell sehr störenden Effekt mit sich. Wird ein Teil eines großen Systems unter verschieden Aspekten betrachtet, so könnte es sein, dass jeder Entwickler in seinem Modell nur die für ihn relevanten Eigenschaften beschreibt. Ein automatisches Hinzufügen der jeweils einzeln beschriebenen Attribute in allen Klassendiagrammen könnte diese relativ schnell unübersichtlich werden lassen und damit den eigentlichen Sinn der Abstraktion zunichte machen. Schlimmer noch könnte eine automatische

[1] www.cvshome.org, Concurrent Versions System

Änderung eines Typs, die erfolgt, um Kompatiblität zu bewahren, die Aussage eines Modells verfälschen. Aus diesen Gründen wedern die Modelle von dem AMTool nicht verändert.

Werden Inkonsistenzen zwischen parallel entwickelten Modellen nicht automatisch vom Programm behoben, so bietet sich für die Fehlerkorrektur nur eine manuelle Fehlerbehebung an. Die Idee zu der im Prototypen implementierten Variante kommt wiederum von dem im vorigen Abschnitt genannten Programm CVS. Dort können mehrere Programmierer gleichzeitig an einer Datei arbeiten. Hat ein Programmierer seine Änderungen vorgenommen, so findet ein Abgleich mit den Änderungen der anderen Programmierer statt. Kann etwas nicht automatisch abgeglichen werden, so muss der Programmierer seinen Code noch entsprechend von Hand anpassen. Ähnlich können in der UML/P die verschiedenen Modelle und die zugehörigen Methodenrümfe abgeglichen werden. Das Werkzeug prüft bei der Codegenerierung, ob Modelle und Coderümpfe zusammenpassen und versucht, aus ihnen weitgehend automatisch ein lauffähiges Programm zu erstellen. Die Modelle selbst werden dabei nicht beeinflusst. Ist eine automatische Korrektur nicht mehr möglich, so wird eine Fehlermeldung ausgegeben und der Programmierer muss seine Modelle von Hand korrigieren. So kann sichergestellt werden, das die Aussagen der Modelle nicht verfälscht werden.

3.1.3 Hohe Parametrisierbarkeit

In Enwicklungsprozessen wie dem eXtreme Programming kann sich die Spezifikation des zu erstellenden Programms oft ändern. Meistens kann durch eine Änderung der Modelle eine Anpassung an die neue Spezifikation stattfinden. Es gibt jedoch auch Anforderungen an das zu erstellende Progamm, die nicht in den Modellen beschrieben werden können. So könnte zum Beispiel einmal Java 1.1 gefordert werden und ein anderes mal Methodenaufrufe aus Java 1.4 benötigt werden. Derartige Anforderungen muss das AMTool bei dem Generierungsvorgang berücksichtigen. Es muss also an das jeweilige Programm, das erstellt wird, angepasst werden können. Eine solche Anpassung findet mittels Parametrisierung statt, das heißt das Verhalten des AMTools kann größtenteils durch die Angabe von Werten, die die Rahmenbedingungen beschreiben, bestimmt werden. Je mehr parametrisiert werden kann, also nicht in starrem Code abgelegt ist, desto schneller kann das AMTool an neue Gegebenheiten angepasst werden.

3.2 Architektur des Werkzeugs

Wie oben beschrieben, ist eine Anfordung an das Werkzeug, dass es eine Eingabe in Form von Text verarbeiten können muss. Da der generierte Code ebenfalls als Text gegeben ist, findet also eine Übersetzung von einer ASCII-Eingabe zu einer ASCII-Ausgabe statt. Diese Aufgabe entspricht der eines Compilers, womit die eigentliche Architektur des AMTools vorgegeben ist. Ein Compiler erfüllt ebenfalls die zweite Anforderung, dass eine Korrektur der UML/P-Modelle nur manuell von

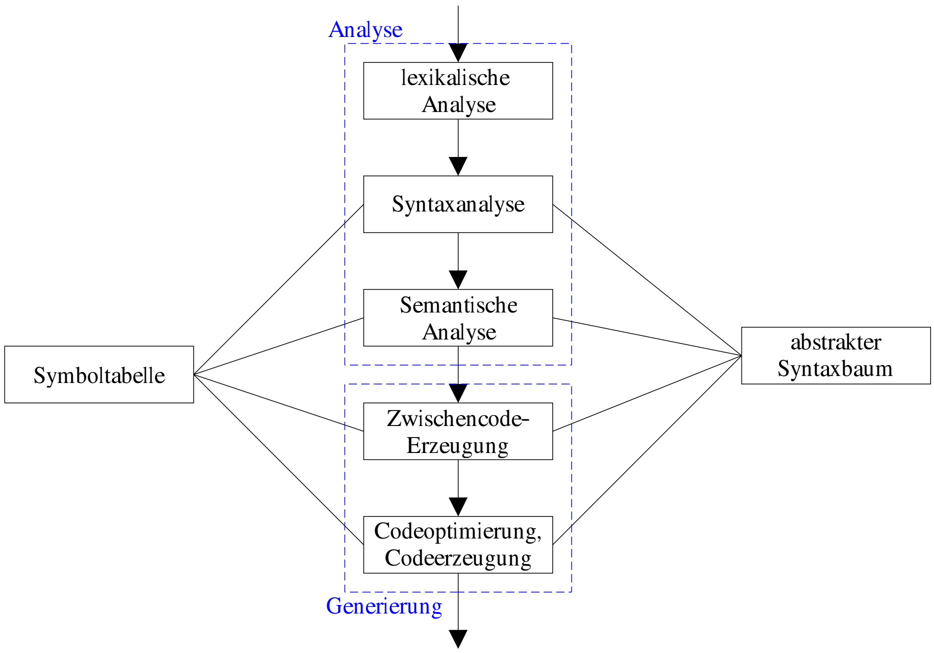


Tabelle 3.1: Die Phasen eines Compilers (siehe [ASU99])

dem Anwender des AMTools stattfinden kann. Auf die UML/P-Modelle wird nur lesend zugegriffen und ist es nicht möglich, aus diesen Programmcode zu generieren, so wird der Anwender mittels einer Fehlermeldung zur Korrektur aufgefordert. Die Anforderung der Parametrisierung ist mit dieser Architektur nicht automatisch gegeben. Ihr kann aber, wie in Abschnitt 3.2.2 beschrieben, durch Hinzufügen von Parametrisierungsmechanismen entgegengekommen werden. Im Folgenden wird nun der Aufbau eines Compilers genauer betrachtet, der die Grundlage für die weiteren Kapitel dieser Arbeit ist.

3.2.1 Aufbau eines Compilers

Der Aufbau eine Compilers ist im Allgemeinen stark an den Ablauf einer Übersetzung von Code gekoppelt. Dieser Ablauf besteht, wie in Abbildung 3.1 zu sehen, im Wesentlichen aus den zwei Phasen *Analyse* und *Generierung*. Die einzelnen Phasen sind nicht zwingend zeitlich von einander getrennt, jedoch Teile von ihnen stehen in kausaler Abhängigkeit.

Die meisten Compiler arbeiten mit Zeichenströmen. Die Eingabe wird als Strom geschickt, dann vom Compiler übersetzt und schließlich wieder in einen Ausgabestrom geschrieben. Gespeichert werden jedoch sowohl die Eingabe, als auch die Ausgabe, in Dateien. Ein Compiler wird daher üblicherweise von Programmen umgeben, die diese Umwandlung von Dateien in einen Strom und umgekehrt vornehmen (siehe Tabelle 3.2). Das Programm, das den Ausgabestrom an ein bestimmtes Medium bindet, nennt man *Linker*. Im Falle einer Java-Codegenerierung wird pro Klasse

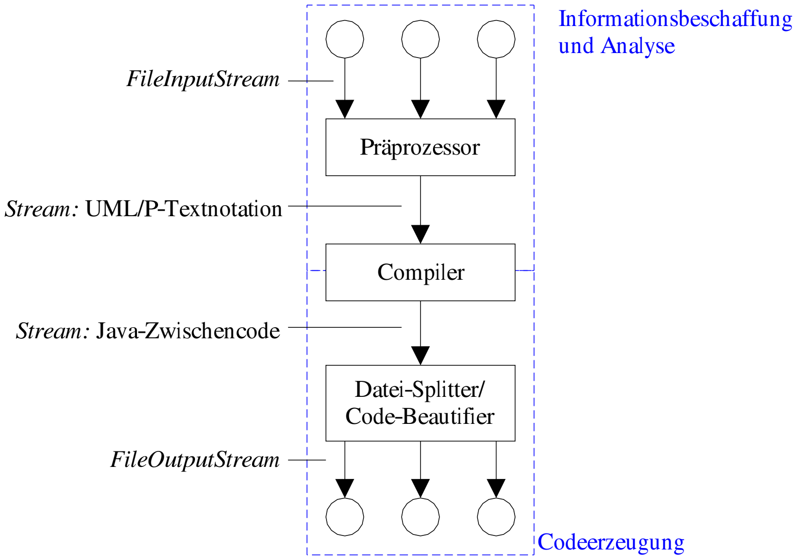


Tabelle 3.2: Die Umgebung des Compilers)

eine Datei im Langzeitspeicher angelegt. Der Vorteil der Umwandlung der Dateien in einen Strom liegt in der Möglichkeit, bei Bedarf mehrere Werkzeuge hintereinanderschalten zu können. Dies geschieht, indem der Ausgabestrom des einen zum Eingabestrom des anderen Werkzeugs gemacht wird. Diese Umwandlungsprogramme sind sehr klein und einfach zu realisieren bzw. bereits vorgefertigt zu beschaffen. Deshalb werden sie im Weiteren nicht näher betrachtet.

Analyse

Die Analyse dient der Aufbereitung von Informationen aus dem Eingabestrom für den Codegenerator. Diese Aufbereitung wird üblicherweise in drei Schritte aufgeteilt: die *lexikalische Analyse*, die *Syntaxanalyse* und die *semantische Analyse*. Ihre jeweilige Bedeutung wird im Folgenden kurz erläutert.

In der lexikalischen Analyse werden den Zeichenfolgen des Eingabestroms Bedeutungen zugewiesen. Es wird dabei wie bei der Verwendung eines herkömmlichen Lexikons vorgegangen. Für jede Zeichenfolge am Anfang des Eingabestroms wird überprüft, ob sie im Nachschlagewerk steht, und wenn das der Fall ist, wird ihr eine, dem Compiler verständliche Erklärung zugewiesen. So kann zum Beispiel zu der Zeichenfolge „143“ die Erklärung „Es handelt sich um eine ganze Zahl“ hinzugefügt werden. Im Vergleich zu einem herkömmlichen Lexikon kann in diesem Fall die Menge der bekannten Zeichenfolgen, wie die der ganzen Zahlen, unendlich sein.

In der Syntaxanalyse wird die Anordnung der Zeichenfolgen im Text untersucht. Mit Hilfe der Erklärungen von der lexikalischen Analyse wird nun versucht, die aufeinanderfolgenden Zeichenfolgen zu gruppieren. So wird zum Beispiel die Eingabe „Bäume haben Blätter!“ als eine zusammengehörende Folge erkannt. Dies geschieht, wie bei herkömmlichen Sprachen auch, mittels einer Grammatik. Die eben genannte

Zeichenfolge wird in die lexikalisch aufbereitete Form „*Substantiv Verb Substantiv Satzzeichen*“ gebracht und kann so mit der Grammatikregel für einen Satz zusammengefasst werden. Eine weitere Regel könnte dann Sätze zusammenfassen, bis schließlich der Text als Ganzes geprüft ist. Der Eingabestrom wird also mit der Syntaxanalyse strukturiert. Eine ausführliche Betrachtung der lexikalischen und der Syntaxanalyse findet in Kapitel 4 statt.

Als Letztes wird bei einer semantischen Analyse die in dem Zeichenstrom enthaltene Information betrachtet. Dabei wird sowohl auf die Ergebnisse der lexikalischen, wie auch auf die Ergebnisse der syntaktischen Analyse zugegriffen. So kann zum Beispiel die Auswertung der Zeichenfolge „ 5*3.0+x “, geprüft werden, ob die Typen vereinbar sind. Die semantische Analyse wird in Kapitel 5 ausführlich behandelt.

Generierung

Die Aufgabe der Generierung ist, aus der aufbereiteten Information des Eingabestroms eine weiterverwertbare Ausgabe zu erzeugen. Im Falle des AMTools wird diese Ausgabe in Form von Dateien gespeichert, die dann von Hochsprachencompilern benutzt werden können. Die Codegenerierung kann, wie die Analyse, in mehrere Phasen zerstückelt werden. Oft wird zuerst ein Zwischencode erstellt, der nicht lauffähig ist und noch nicht formatiert sein muss, aber bereits alle im Code gewünschte Funktionalität enthält. Er enthält außerdem Informationen für weitere Arbeitsschritte. Im Zwischencode des AMTools sind zum Beispiel Markierungen für die Aufspaltung in Dateien enthalten.

Eine wesentliche Eigenschaft von Zwischencode ist, dass er leicht zu erzeugen ist und ebenfalls einfach in den endgültigen Code umgewandelt werden kann. So kann er die Komplexität des Generierungsvorgangs verringern. Weitere Bearbeitungsphasen des Zwischencodes beim AMTool sind die Aufteilung in verschiedene Dateien und eine neue Formatierung des Codes hinsichtlich besserer Lesbarkeit.

3.2.2 Parametrisierungsmechanismen

Wie in Abschnitt 3.1.3 erwähnt, ist die Parametrisierbarkeit des AMTools besonders hoch. Die realisierte Parametrisierung kann in zwei Kategorien unterteilt werden. Einerseits kann der Anwender durch Auswahl schnell und problemlos zwischen bereits existierenden Funktionalitäten wählen und andererseits aber auch relativ einfach neue Funktionalitäten erstellen.

Bei der die Auswahl existierender Funktionalität kann ein aufwendiges Übersetzen des AMTools vermieden werden. Die Idee ist, die Funktionalität aus Modulen zusammenzusetzen, welche erst zur Laufzeit an das AMTool gebunden werden. Bei jedem Start des AMTools kann der Anwender mittels eines Aufrufparameters die Modulzusammensetzung wählen, die er gerade braucht. Die Voraussetzung für ein derartiges Vorgehen ist jedoch, dass die Module selbst entweder bereits in compilierter Form vorliegen oder zur Laufzeit erzeugt werden können. Die Module, die zu einer Funktionalität gehören, können in einem *JAR*-Archiv zusammengepackt

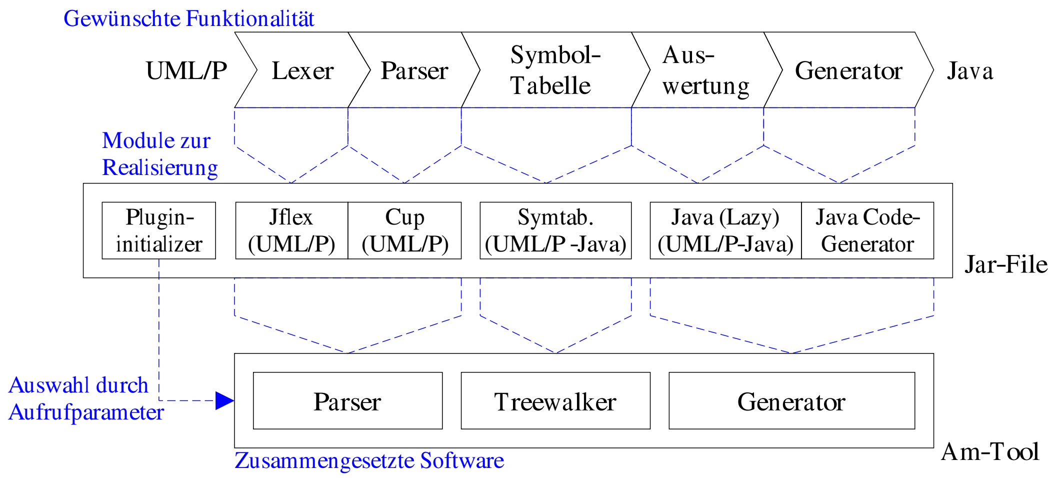


Tabelle 3.3: Das PlugIn-Konzept zur Funktionalitätsauswahl

werden. Dieses kann wie ein *Plug-In* an den Code gebunden werden, indem es in den Klassenpfad aufgenommen wird. Um nun die Funktionalität zu wählen, reicht die Angabe des Namens einer Konfigurationsklasse, die ebenfalls im JAR-Archiv vorhanden ist. Eine graphische Illustration dieser Plug-In-Idee ist in Abbildung 3.3 zu sehen.

Die Parametrisierung zum Erstellen neuer Funktionalitäten ist im Vergleich zur Funktionalitätsauswahl wesentlich aufwendiger. Der Grund ist, dass hier die Module selbst geändert werden müssen. Im Weiteren werden nun drei Möglichkeiten vorgestellt, um Module zu parametrisieren. Eine sehr effiziente Möglichkeit, Module zu parametrisieren, ist die Verwendung von *Skripten* (siehe Tabelle 6.1). Der Vorteil von Skripten ist, dass sie erst zur Laufzeit ausgewertet werden, also eine Änderung sofort wirksam ist. Skripten werden meist im Generator verwendet, in dem die Abbildung der aufbereiteten Information auf den Code stattfindet. Für einfache Aufgaben sind sie wesentlich übersichtlicher und effizienter zu erstellen als Java-Code. Ein Beispiel für eine Parameteränderung ist die Konvention, dass vor jedem Attribut ein Unterstrich stehen soll. Um das AMTool entspechend anzupassen, muss man lediglich in dem Skript, welches das Attribut und die Getter/Setter-Methoden erstellt einen Unterstrich vor die Attributsnamenausgabe setzen und es speichern. Weitere Schritte sind nicht mehr nötig.

Komplexere Aufgaben, wie die Analyse des abstrakten Syntaxbaumes, sind in Java realisiert. Die Funktionalitäten werden, wie in Kapitel 5 zu lesen, in Klassen aufgeteilt. Eine Änderung oder Erweiterung zieht meistens ein Eingreifen in die Methoden dieser Klassen mit sich. Dementsprechend müssen die Dateien erst compiliert werden, damit die Änderungen im AMTool wirksam werden. Ein Übersetzen des gesamten Werkzeugcodes ist jedoch nicht nötig.

Bei Java-Methoden kann eine Parametrisierung auf zwei Wegen stattfinden. Parameter können bei einem Methodenaufruf übergeben werden oder es können Parameterabfragen durch gezielte Aufrufe von Parametermethoden realisiert werden. Bei der

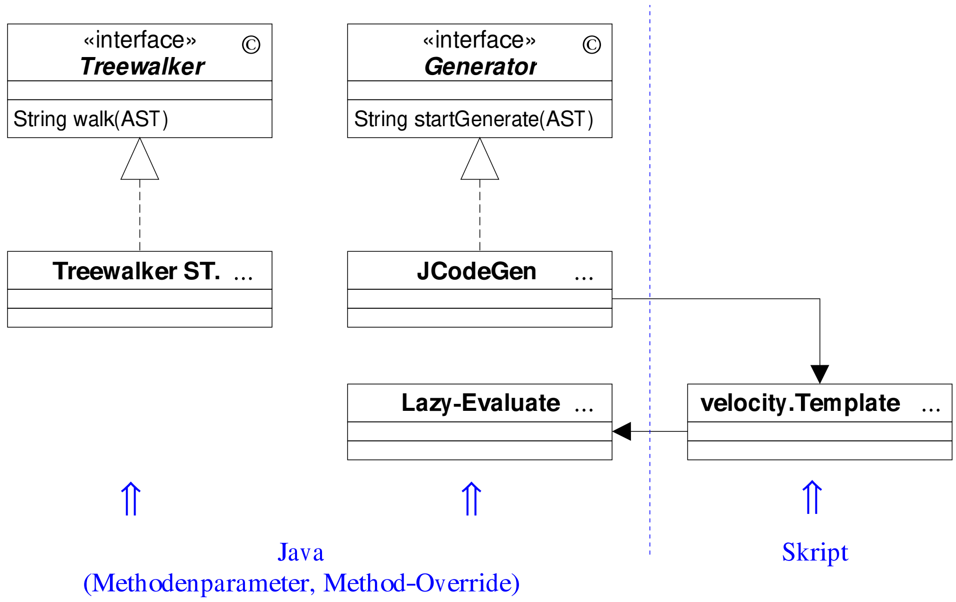


Tabelle 3.4: Der Aufbau der parametrisierbaren Module

ersten Möglichkeit werden die Parameter bereits vor dem Methodenaufruf berechnet und dann der Methode übergeben. Der deutliche Vorteil eines solchen Vorgehens liegt darin, dass die gleiche Methode mit verschiedenen Parametern verwendet werden kann. Dieses Vorgehen bringt jedoch auch zwei Nachteile mit sich. Es müssen bereits alle Parameter berechnet sein, wenn die Methode aufgerufen wird und für die Einführung eines neuen Parameters muss die Signatur einer Methode geändert werden. Ändert sich die Signatur einer Methode, so müssen alle Stellen im Code, bei denen diese Methode aufgerufen wird, korrigiert werden. Bei der zweiten Parametrisierungsmöglichkeit haben neue Parameter keine Auswirkung auf die Signatur. Die Parameter einer Methode bleiben so für den restlichen Code transparent. Um einen Parameter zu ändern, muss in der ensprechenden Parametermethode der Code geändert werden. Überschreibt man die Parametermethoden mittels Vererbung, so können mit geringem Aufwand Module mit veschiedenen Parameterwerten entstehen. Im AMTools werden beide Arten der Parametrisierung verwendet, da jede ihre eigenen Vorzüge hat. Diese beiden Konzepte werden noch einmal in Tabelle 3.5 gegenübergestellt.

Nicht alle Module des AMTools sind parametrisierbar. So ist zum Beispiel eine Parametrisierung der Module zur syntaktischen Analyse oft nur schwer oder gar nicht möglich. Meistens werden hier Generatoren eingesetzt, die fertige Klassen liefern. Dabei muss lediglich die Grammatk angegeben werden, die eine Sprache beschreibt. Eine Grammatikregel alleine ist aber nicht parametrisierbar, sondern es muss immer die Grammatik als Ganzes angegeben werden.

Impl.

```
... /* Umgebung für Methode */ ...
access = getAccessModifier(symbol, getDefaultAccess_Method());
...
... /* Umgebung für Feld */ ...
access = getAccessModifier(symbol, getDefaultAccess_Field());
...

/** Einfache Methode, aber Parameteränderungen betreffen alle Aufrufe
  */
String getAccessModifier(Symbol symbol, String defaultParam) {
   if ( symbol.access == null) return defaultParam;
   return symbol.access;
}
```

Impl.

```
... /* Umgebung für X */ ...
access = getAccessModifier(symbol);
...

/** Methode komplexer, dafür haben Parameteränderungen keine
  *  Auswirkung auf Signatur der Methode
  */
String getAccessModifier(Symbol symbol) {
   if ( symbol.access != null) return symbol.access;
  ... herausfinden der Umgebung und des Defaultwertes ...
   ... defaultParam = getDefaultAccess_Field() ...
  return defaultParam;
}
```

Tabelle 3.5: Aspekte der Methodenparametrisierung

3.3 Qualitätssteigernde Maßnahmen

Neben dem Erstellen von der Funktionalität einer Software spielt auch deren Qualität eine große Rolle. Mit qualitätssichernden Maßnahmen kann auf der einen Seite dem Nutzer ein besseres Programm geliefert werden, und auf der anderen Seite der Code für den Entwickler wesentlich leichter wartbar werden. Gerade das AMTool als universitärer Prototyp muss einfach weiterentwickelbar sein und ein gutes Bild über die implementierte Funktionalität geben, da hier die Bearbeitungszeit der einzelnen Programmierer bei Systementwicklungsprojekten und Diplomarbeiten ein halbes Jahr nicht übersteigen wird. In diesem Anschnitt werden zwei sich ergänzende Maßnahmen vorgestellt, mit denen speziell die Übersichtlichkeit des des AMTools und die Behandlung von Programmierfehlern verbessert werden.

3.3.1 Tests

Bei Entwicklungprozessen wie dem eXtreme Programming werden zur Qualitätssteigerung *Tests* für den Code entwickelt. Ein Test ist ein eigenständiges Progamm, welches testet, ob eine Methode oder ein ganzes Programm die gewünschte Funktionalität hat. Dabei wird im Wesentlichen zwischen drei Arten von Tests unterschieden. Die erste Art von Tests, die sogenannten *System-Tests*, prüft ganze Programme. Mit ihnen wird das Verhalten der Software nach aussen hin getestet. Ein Ebene tiefer sind die sogenannten *Black-Box*-Tests angesiedelt. Sie prüfen Methoden aus einer Black-Box-Sicht, indem sie die Methode lediglich von aussen aufrufen und deren Funktionalität betrachten. Als drittes gibt es noch die *White-Box*-Tests, die zwar genau wie die Black-Box-Tests die Methoden von aussen aufrufen, aber gezielt kritische Stellen innerhalb des Methodencodes prüfen.

Betrachtet man Tests als qualitätssichernde Maßnahme, so bleiben einige Anforderungen offen. So tritt häufig der Fall ein, bei dem für eine Zusicherung einer Eigenschaft im ungünstigsten Fall unendlich viele Tests geschrieben werden müssten. Möchte man zum Beispiel sicherstellen, dass ein Wert in einer Methode für alle Aufrufparameterwerte nicht *null* wird, so müsste man konsequenterweise die Methode auch mit allen Parameterwerten einmal aufrufen und testen. Ist ein Parameter eine natürliche Zahl, so gibt es für ihn unendlich viele Werte, das heißt der Test würde nie enden. Der Kompromiss bei der Verwendung von Tests zu einer Eigenschaft beschränkt sich mit Rücksicht auf die Entwicklungszeit fast immer auf eine kleine Auswahl von prägnanten Fällen. Mit diesen können meistens 90% der Fehler ausgeschlossen werden. Möchte man jedoch eine volle Zusicherung erhalten, so eignet sich in den meisten Fällen das Mittel Tests nicht, sondern es müsste mit *Codereviews* und *Zusicherungsbeweisen* eine Kontrolle stattfinden. Ein derartiger Aufwand lohnt sich in erster Linie bei Systemen mit hohen Sicherheitsanforderungen, wie zum Beispiel in Kraftfahrzeugen.

Tests sind, gerade in Bezug auf den Entwicklungprozess gesehen ein relativ einfach zu entwickelndes und zu verstehendes System. Sie verdoppeln zwar in erster Sicht die benötigte Entwicklungszeit einer Methode, aber die dafür erhaltene Qualitätssteigerung rechtfertigt in den meisten Fällen die Nutzung. So wird auch das AMTool mit Tests ausgestattet, die im Folgenden näher beschrieben werden.

Tests für den Anwender

Aus Sicht des Anwenders bringen Tests im wesentlichen zwei Vorteile. Zum einen sind, wie eben schon erwähnt, in dem Programm weniger Fehler und zum anderen kann der er sicherstellen, dass gewisse Funktionalitäten implementiert sind. Für den Anwender sind hauptsächlich die Tests relevant, die die Software als Ganzes testen. Mit diesen Tests kann relativ gut erkannt werden, welche Funktionalität die Software hat. So kann der Nutzer allein durch Betrachten der Tests oftmals mehr über die Software erfahren, als durch einen Blick in den Code. Neben dem Einblick in die Funktionalität der Software, bekommt der Nutzer mit Tests ein relativ einfaches Mittel, um den Entwicklern die eigenen Wünsche mitzuteilen. So kann der Nutzer des AMTools zum Beispiel bei einer bestimmten Eingabe eine bestimmte Ausgabe fordern und dies gleich in einem Test festhalten lassen. In späteren Versionen des AMTools können vielleicht zu diesem Zwecke Tests automatisch aus *State-Charts* generiert werden, die den Workflow einer Firma abbilden.

Bei einem Compiler, wie dem AMTool, ist die Entwicklung von Unit-Tests relativ einfach. Es muss lediglich eine Mustereingabe erfolgen und das Ergebnis mit einer Musterausgabe verglichen werden. Wie das Ergebnis berechnet wird, ist bei diesen Tests nicht von Interesse. Der Nutzer kann, dank der Ein- und Ausgabe im Textformat, sofort sehen, ob für eine Eingabe die gewünschte Ausgabe erfolgt. Die Entwicklung eines Tests ist dabei relativ einfach. Das hier verwendete Vorgehen ist das automatische Durchsuchen eines Musterverzeichnisses nach Eingabedateien, deren Musterausgabe in einem gleichnamigen Unterverzeichnis gespeichert ist. Zur Erstellung eines Test muss also lediglich eine Ein- und Ausgabedatei erstellt werden.

Tests für Entwickler

Der Entwickler des AMTools benötigt neben den eben genannten Unit-Tests, die ihm zusichern, dass der Code mit dem Wunsch des Nutzers übereinstimmt, zusätzliche Tests. Mit Softwaretests kann die Funktionalitätsprüfung der Software komplett abgedeckt werden. Softwaresysteme sind jedoch meistens so komplex, dass sie aus mehreren Einheiten mit Teilfunktionalitäten zusammengesetzt werden. Möchte nun ein Entwickler an einer Einheit etwas ändern oder einen Fehler im System analysieren, so ist es sehr hilfreich, auch mittels Tests Überprüfungen zu der gewünschten Funktionalität dieser Einheit machen zu können. Die Größenordnung der Einheiten kann, je nach Bedarf und Komplexität des Programms, von Modulen bis hin zu einzelnen Grammatikregeln gehen.

Gerade in komplexen Teilen der Software haben Tests enorme Vorteile. In den meisten Fällen ist davon auszugehen, dass ein Entwickler nicht mehr den Überblick über alle Zusammenhänge und Auswirkungen von Code-Stücken hat. Werden nun mit Tests die Auswirkungen festgehalten, so kann ein Entwickler nach einer Änderung des Codes mit einem einfachen Testlauf überprüfen, ob noch alle gewollten Auswirkungen auftreten oder ungewollte Seiteneffekte ausbleiben. Ein Paradebeispiel ist die Prüfung von Grammatikregeln, die die Eingabe des AMTools beschreiben. Diese Regeln rufen sich gegenseitig auf, bzw. bauen aufeinander auf. Für jede Regel kann mit einem Test zugesichert werden, dass sie die gewünschten Ergebnisse liefert. Muss eine Regel geändert werden, so betrifft das manchmal auch etliche andere Regeln, die ihr Verhalten dann entsprechend wechseln. Wiederum kann man mit einem Testdurchlauf alle Seiteneffekte erkennen und gegebenenfalls korrigieren.

Ein Vorteil einer konsequenten Erstellung von Tests bei der Entwicklung von neuem Code ist die Notwendigkeit, den Code klar zu strukturieren. Tests müssen dazu explizit in einer Programmiersprache formuliert werden. Meist wird mit der Formulierung eines Tests die Funktionalität des getesteten Codes so klar beschrieben, dass diese in der zu testenden Methode gut und verständlich abgefasst werden kann. Der Entwickler wird durch die Tests also gezwungen, sich mit den einzelnen Elementen seines Codes auseinanderzusetzen.

Tests von Teilen der gesamten Software sind fast ausschliesslich Black- und White-Box Tests. So gibt es bereits ein Java-Paket namens *JUnit*, welches einen Rahmen für die Erstellung von Tests auf Methoden liefert. Ein Test verläuft dabei nach einem festen Schema vor. Zuerst wird eine Umgebung aufgebaut, in der die zu testende Methode arbeiten kann. Danach wird diese Methode mit einer Mustereingabe ausgeführt und schließlich werden der Rückgabewert und die Seiteneffekte überprüft. JUnit übernimmt dabei den automatischen Aufruf einer Methode zum Aufbau und einer weiteren zur Ausführung des Tests. Der Inhalt dieser Methoden muss jedoch vom Entwickler selbst geschrieben werden.

Der Aufbau einer Testumgebung ist bei jeder zu testenden Methode verschieden. Gemeinsam ist jedoch allen Tests die Überprüfung der Ausgabe und der Seiteneffekte. In den meisten Fällen läuft ein Überprüfen auf einen Vergleich von Objektstrukturen hinaus. Eine Unterstützung für den Programmierer bietet hier ein Paket namens *vdiff* [Kuk01]. Mit lediglich zwei Methoden können sowohl Seiteneffekte als auch Ausgaben ohne großen Aufwand kontrolliert werden. Möchte man die Ausgabe testen, so erfordert ein Test die Erstellung der Musterausgabe und anschließend den Aufruf der Methode *vdiff.VCompare.compare()* zusammen mit dem Ergebnis der zu testenden Methode. Treten Unterschiede auf, so kann der Rückgabewert *false* sofort bei einer *assert*-Anweisung eine *Exception* auslösen. Bei der Überprüfung von Seiteneffekten ist das Erstellen einer Musterausgabe, bzw. einer Musterumgebung oft sehr aufwendig. Sinnvoller erscheint es hier, nur die Seiteneffekte selbst, und nicht die ganze Umgebung festzuhalten. Das in dem Paket *vdiff* vorgeschlagene Vorgehen sieht vor, zuerst die Umgebung einer Methode zu kopieren, dann die Methode aufzurufen und schließlich die gewünschten Seiteneffekte mit den tatsächlich aufgetretenen zu vergleichen. Auch hier steht wiederum eine Methode *vdiff.VClone.clone()* bereit, die Objektverbände kopiert. Eine Methode *vdiff.Vcompare.verifyCompare()* gibt dann anhand eines Strings, der die Unterschiede beschreibt, an, ob die Auswirkungen in Ordnung sind.

Mit diesen Hilfsmitteln sind Tests erheblich schneller und einfacher zu erstellen. Trotzdem ist der Zeitaufwand relativ hoch (ca. 50% der Programmierzeit). Es bleibt im Ermessen des Programmierers, wieviele Tests zu einer Methode geschrieben werden, damit diese mit einer relativ hohen Wahrscheinlichkeit fehlerfrei funktioniert.

3.3.2 Dokumentation für Reviews und Erweiterungen

Mit Tests wird die Funktionalität von Methoden relativ gut kontrolliert. Es ist jedoch nicht möglich, mit Test Aussagen über die Effizienz oder Übersichtlichkeit einer Implementierung zu machen. Ebenso können Fehler auftreten, die mit Tests nur schwer zu finden sind. Eine verbeitete Technik zum Auffangen dieser Probleme ist die Durchführung von Code-Reviews. Idealerweise sieht sich ein Entwickler, der mit dem geschriebenen Code noch nichts zu tun hatte, diesen an und gibt dann an den eigentlichen Entwickler des Codes Verbesserungs- bzw. Korrekturvorschläge weiter. Die Effizienz der Ausführung eines Reviews hängt sehr stark von der Einarbeitungszeit des jeweiligen Kontrolleurs ab. So ist es nahezu unvermeidbar, den geschriebenen Code entsprechend zu kommentieren bzw. zu dokumentieren.

Neben den Reviews ist es wichtig, neuen Mitarbeitern, die eine Entwicklung fortsetzen wollen, eine schnelle Einarbeitungsphase zu ermöglichen. Genau wie die Code-Reviewer müssen sie schnell bestehende Methoden verstehen können, aber auch einen guten Überblick über die gesamten bereits implementierten Methoden haben, um diese nutzen zu können. Möche man den in Methoden geschriebenen Code anschauen, so ist dieser kompakt in einem Methodenrumpf zusammengefasst. Dort können dann auch Kommentare für die sogenannte White-Box-Sicht der Methode untergebracht werden, die an komplizierten Methodenteilen ein schnelles Verständnis erleichtern. Wieder liegt es im Ermessen des Programmierers, wann sich Code selbst erklärt, ihn eine Kommentierung also eher unübersichtlich macht, und wann eine Kommentierung nötig ist, da mit dem Code allein noch Fragen offen bleiben.

Im Gegensatz zur Sicht in den Code einer Methode selbst, ist es jedoch sehr schwierig, mit einem Blick in diesen die gesamten Funktionalitäten aller Methoden zu überschauen. Java bietet hierzu ein Werkzeug namens *JavaDoc* (Java Documentation Tool) an, das in einem HTML-Dokument die Black-Box-Sichten auf alle Methoden sammelt. Es werden dabei alle Code-Dateien durchforstet und speziell markierte JavaDoc-Kommentare herausgefiltert. Ein JavaDoc-Kommentar beginnt mit einem „/**“ und endet mit einem „*/“. Er wird jeweils vor dem Element positioniert, das er beschreibt. Mit JavaDoc können Pakete, Klassen, Felder und Methoden beschrieben werden. Dank der Hyperlink-Technologie kann sich so ein Entwickler in der generierten Dokumentation schnell orientieren und die Funktionalitäten überblicken. Die nun folgenden Zeilen demonstrieren etwas das Aussehen von gut dokumentiertem Code:

Impl.

```
/** Diese Klasse ist eine Beispielklasse für die Ausarbeitung.
 */
class sample {

  /** Diese Methode berechnet etwas.
   */
  public int doSomething() {
    /* Hier ist eine Erklärung zu einem komplizierten Code-Teil
     */
    ...

    return x;
  }
}
```

Kapitel 4

Lexer und Parser

Der erste Schritte eines Codegenerierungswerkzeuges ist die Erfassung der Modelle. Wie in Kapitel 3 erklärt, kann der hier beschriebene Prototyp vorerst eine Eingabe nur in Form von Text verarbeiten. Da bei der UML einige Modelle, wie Klassendiagramme und Objektdiagramme, jedoch nur graphisch gegeben sind, muss für diese erst eine Textnotation entworfen werden. Die Sprache, die das Werkzeug als Eingabe versteht, also die Notation der UML/P, ist die Menge aller *Wörter*, die in ihr enthalten sind. Ein *Wort* ist eine Beschreibung eines Systems in UML/P, welche als Eingabe verwendet werden kann. Die Menge der Wörter, die in einer Sprache enthalten sind, werden üblicherweise mit einer *Grammatik* beschrieben. Eine Sprache, die die Wörter „aa“, „aaaa“,“aaaaaa“ usw. enthält, kann zum Beispiel durch folgende Grammatik beschrieben werden.

```
A ::= aa;
A ::= Aaa;
```

Ausgehend von einem Ursprungselement (A) kann durch Anwendung der Regeln jedes Wort der eben definierten Sprache erzeugt werden. Möchte man prüfen, ob ein Wort mit der Grammatik erzeugt werden kann, so wendet man die Regeln umgekehrt an und testet, ob man zu dem Ursprungselement gelangt. Die umgekehrte Anwendung der Regeln nennt man eine *Reduktion*, da jedes Wort mit der Grammatik auf das Ursprungselement zurückgeführt wird.

Programmiersprachen sind immer *kontextfreie* Sprachen, das heißt jede Regel der Grammatik hat auf der linken Seite genau ein Element. So wird sichergestellt, dass während einer Reduktion ein Wort nicht länger wird und ein Reduktionsversuch somit in endlicher Zeit endet. Im Folgenden werden nun zwei Konzepte für kontextfreie Sprachen hinsichtlich ihrer Eignung für das AMTool bewertet.

Einige Programme, wie zum Beispiel *Gnome-Dia*[1], können die in ihnen graphisch erstellten Klassendiagramme im *XML-Format* (*eXtensible Markup Language*) speichern. Die Überlegung, nun auch für die textuelle Form der UML/P einen *XML-Dialekt* zu verwenden zieht einige kontroverse Argumente mit sich, die letztendlich

[1] www.lysator.liu.se/˜alla/dia/

in diesem Implementierungsvorschlag gegen XML sprechen. Trotzdem werden die Überlegungen hier erwähnt. Die XML ist, wie der Name schon sagt, eine *Markup*-Sprache, das heißt jedes Sprachelement ist von einem öffnenden und einem schliessenden *Tag* umgeben (siehe Abbildung 4.2). So kann man mit ihr unbeschränkt kontextfreie Sprachen erzeugen, deren Wörter beim Lesen von links nach rechts in linearer Zeit mit einer Grammatik reduziert werden können. Dank ihrer guten Strukturierung eignet sie sich hervorragend zur Speicherung und zum Austausch von Daten. Allerdings enthält sie durch die Tags einen gewaltigen Anteil an Stukturinformation, der den Quelltext dem Menschen sehr überladen erscheinen lässt und so unleserlich macht. Der Mensch ist eher einen fließenden Text gewöhnt, in dem keine Baumstruktur explizit festgehalten wird. Aus diesem Grund wird bei dem AMTool von einer Markup-Sprache als Programmiersprache abgesehen.

Sieht man nun von Markup-Sprachen ab, so kann man theoretisch jede beliebige Sprache nutzen, die mit einer kontextfreien Grammatik beschreibbar ist. Eine gute Beschreibungsmöglichkeit für kontextfreie Grammatiken ist die *Backus Naur Form* (BNF). Mit ihr können Grammatiken in einer anschaulichen Form dargestellt werden. Neben *regulären Ausdrücken* sind in der BNF noch Hilfsvariablen, in Form von *Nichtterminalen*, und *Rekursion* gegeben (siehe [Bro98] Teil II, Seite 243 ff.). Um die Lesbarkeit nochmals deutlich zu verbessern, wird hier die BNF um einige Regeln ergänzt. In der *erweiterten BNF* (EBNF) kann beim Wiederholungsoperator nun auch eine Ober- und Untergrenze für die Zahl der Wiederholungen angeben werden und die Angabe eines Seperators zwischen den wiederholten Elementen ist zulässig. Tabelle 4.1 zeigt die Elemente der Sprache auf.

jedes Zeichen x	Terminalzeichen	RA
ϵ	das leere Zeichen	
X | Y	entweder X oder Y	
XY	erst X dann Y	
X^*	X beliebig oft hintereinander	
[X]	Klammern des Audrucks X	
[a-z]	alle Zeichen von a bis z	Erweiterung
X^+	X beliebig oft aber mindestens einmal	(für JFlex)
<A>	Nichtterminale	BNF
<A> := RA(<A>)	Rekursion mittels Nichtterminalen (RA(A) ist regulärer Ausdruck der A enthält)	(enthält RA)
X^+	mindestens einmal X	EBNF
X^{k-l}	k bis l mal X	(enthält BNF)
X_z^*	X beliebig oft mit Seperator z	

Tabelle 4.1: Reguläre Ausdrücke und EBNF

Der Gewinn der Lesbarkeit von Programmiersprachen geht auf Kosten von Problemen bei der Reduktion von Wörtern mit einer Grammatik. Steigt der Zeitbedarf zur Reduktion bei Markup-Sprachen direkt proportional zur Wortlänge, kann er bei beliebigen kontextfreien Grammatiken unter Umständen exponentiell zur Wortlänge

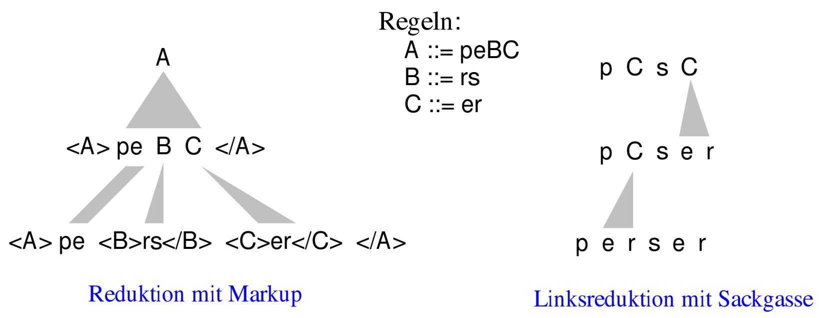


Tabelle 4.2: Lösung der Reduktionsprobleme eines Wortes mittels Markup

steigen. Abbildung 4.2 demonstriert das Problem, dass ein Programm, welches ein Wort von links nach rechts liest, bei der Reduktion in Sackgassen geraten kann und so verschiedene Regeln ausprobieren muss. Effizienter sind Sprachen, deren Grammatiken einen eindeutigen Pfad zum Reduzieren eines Wortes haben. Eine derartige Grammatik ist zum Beispiel eine *LR(k)-Grammatik*. LR(k) steht für *left rightmost*, das heißt, es kann immer mittels *Linksreduktion* reduziert werden, bzw. wenn dies nicht der Fall ist, so kann durch Vorauslesen von k Zeichen eine eindeutige Regel gefunden werden.

Programmiersprachen sind üblicherweise LR(1)-Sprachen oder können durch geeignete Anpassung der Grammatik und einer nachgeschalteten Prüfung der Kontextbedingungen zu LR(1)-Sprachen transformiert werden. Diese bieten einen umfangreichen Schatz an sprachlichen Konstrukten und ihre Wörter können in linearer Komplexität abgeleitet werden. Im weiteren wird die Beschreibung der Eingabe-Grammatik des AMTools in EBNF stattfinden. Die Umsetzung zu einer LR(1)-Grammatik ist nicht immer einfach, aber meistens systematisch mit geringem Aufwand machbar. In Abschnitt 4.6 wird ein Vorgehen für die Standardumsetzung vorgeschlagen und in Abschnitt 4.7 wird auf spezielle Probleme eingegangen. Weitere Hilfen zum Erstellen von LR(1)-Grammatiken finden sich in vielen Büchern und Artikeln, wie z.B. [ASU99].

4.1 Aufbau eines abstrakten Syntaxbaumes

Jedes Programm, das eine Sprache analysiert, baut einen *Syntaxbaum* auf. Ein Syntaxbaum ist eine Baumstruktur, die die Reduktion eines Wortes aufzeichnet und so die Struktur der Grammatik explizit darstellt. Informationen lassen sich wesentlich schneller finden als in einer linearen Notation wie einem Wort. Abbildung 4.3 zeigt auf der linken Seite eine solche Baumstruktur. Ein Knoten des Baumes ist dabei entweder die linke Seite von einer Regel oder, falls es sich um ein Blatt handelt, ein Zeichen. Die Kinder der Knoten sind die Zeichen, die auf der rechten Seite einer Regel stehen.

Um mit dem Syntaxbaum besser systematisch arbeiten zu können, ist es sinnvoll, nicht die tatsächlich vorhandenen Zeichen zu betrachten, sondern nur deren Be-

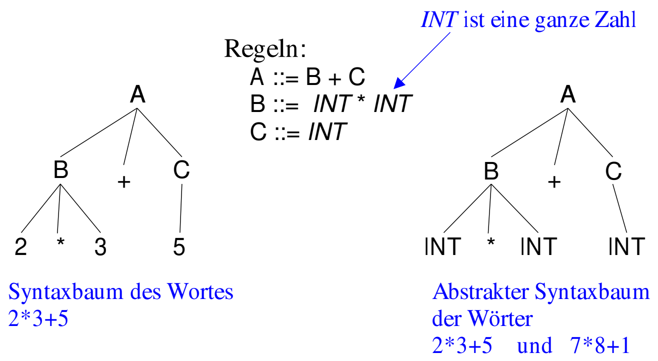


Tabelle 4.3: Beispiel für Syntaxbäume

deutung zu abstrahieren. Ein Baum, der die Zeichen selbst nicht darstellt, heißt abstrakter Syntaxbaum. So sieht zum Beispiel der in Abbildung 4.3 auf der rechten Seite zu sehende abstrakte Syntaxbaum für alle Terme der Form „$INT * INT + INT$" gleich aus. Eine Analyse dieser Terme kann so wesentlich einfacher ausgeführt werden. In diesem Abschnitt wird nun gezeigt, wie der abstrakte Syntaxbaum aus den Wörtern einer Sprache aufgebaut werden kann.

Aufgrund der großen Verbreitung von LR(1)-Sprachen gibt es mittlerweile eine große Menge an Software, die den Aufbau eines abstrakten Syntaxbaumes mittels LR(1)-Regeln unterstützt. Der Aufbau des Baumes wird im allgemeinen in zwei Schritten vorgenommen.

Im ersten Schritt findet eine *lexikalische Analyse* statt. Der eingehende Zeichenstrom wird dabei gelesen und in *Symbole* aufgeteilt. Ein Symbol stellt hier eine Gruppe von Zeichenfolgen mit einer bestimmten Bedeutung dar. Die Zeichenfolgen einer Gruppe haben alle gemeinsame Eigenschaften, anhand derer ihre Zugehörigkeit bestimmt werden kann. Programme, die die lexikalische Analyse durchführen, nennt man *Lexer* oder *Scanner*. Um die zusammengehörenden Zeichenfolgen zu beschreiben, eigenen sich sich *reguläre Ausdrücke* (siehe Tabelle 4.1). Es gibt bereits Programme, sogenannte *Lexer-Generatoren*, die aus einer Reihe von regulären Ausdrücken einen Lexer erstellen. Ein Beispiel für einen Lexer-Generator, der einen in Java geschriebenen Lexer produziert, ist *JFlex*[2]. Der folgende Auszug demonstriert die Vorzüge der Definition einer Zeichenfolge mittels regulärer Ausdrücke.

JFlex

```
// Namen fuer Bezeichner
[0-9][0-9]*[.][0-9]*   { return sym(FLOATNUM,yytext()); }
[0-9][0-9]*      { return sym(INTNUM,yytext()); }
"‘+"’ { return sym(PLUS); }
```

Der Lexer geht Zeile für Zeile durch die regulären Ausdrücke und wenn ein Ausdruck zu der eingelesenen Zeichenfolge passt, so wird der in den Klammern dahinter angegebene Code ausgeführt. Bei der Eingabe „1.5" gibt der Lexer hier zum Beispiel das Symbol „FLOATNUM" und die zugehörige Zeichenfolge „1.5" weiter.

[2] www.jflex.de

Der zweite Schritt nach der lexikalischen Analyse ist die *Syntaxanalyse*. Hier werden die Symbole hierachisch zusammengefasst, wobei durch das Zusammenfassen wiederum ein Symbol entsteht. Das Ergebnis ist der abstrakte Syntaxbaum. Programme mit einer solchen Funktionalität nennt man *Parser*. Die Entwicklung eines Parsers ist relativ aufwendig, aber es gibt Software, die aus einer Menge von Ableitungsregeln einen Parser generiert. Ein klassischer *Parser-Generator*, der für Parser in der Sprache *C* geschrieben ist, ist *Yacc* (Yet Another Compiler Compiler). Es gibt viele ähnlich Parsergeneratoren, die aber alle wie *Yacc* funktionieren. Einer davon, der Java-Parser generiert, ist *Cup*[3]. Das folgende Beispiel veranschaulicht die Definition einer Grammatikregel von Cup.

Cup

```
expression ::= INTNUM:i
        {: RESULT = new Texp(i); :}
      | expression:e1 PLUS expression:e2
        {: RESULT = new Texp(e1,e2); :}
      ;
```

Mit dieser Regel werden alle Summen von ganzen Zahlen akzeptiert. Die Eingabe „1+2+3“ wird vom Lexer zu „INTNUM PLUS INTNUM PLUS INTNUM“ umgewandelt. Dann werden vom Parser die ersten beiden Zahlen zusammengefasst und schließlich die letzte Zahl zur Summe addiert. In den geschweiften Klammern kann der Anwender den Java-Code eintragen, der bei der Ausführung einer Regel angewandt wird. Die Symbole, die in der Regel enthalten sind, können mit Namen versehen werden, auf die der Code zugreifen kann. Ein solcher Name wird hinter das Symbol mit einem vorangestellten Doppelpunkt geschrieben. Das Ergebniss einer Regelanwendung kann in dem Platzhalter „RESULT“ weitergegeben werden.

4.2 Klassendiagramme

Die Definition der Notation für Klassendiagramme in der UML/P muss in ihrer Form für Anwender des Werkzeugs schnell nachvollziehbar sein. Deshalb wird Java, das bereits einige Elemente zur Strukturierung von Sprache enthält, größtenteils übernommen. Die in diesem Abschnitt definierte Notation ist ein Kompromiss aus einer direkten Umsetzung der Klassendiagramme und einer Java-Strukturierung. Tabelle 4.4 zeigt die Syntax der Klassendiagramme.

Wie bei Java erfolgt die Strukturierung in Form von Blöcken, die ineinander geschachtelt werden können. Es wird zuerst ein Schlüsselwort genannt und danach folgt, bei Bedarf, in geschweiften Klammern eine genauere Definition. Es gibt jedoch dabei zwei Ausnahmen, bei denen dieses Konzept nicht eingehalten wird.

Die äußerste Schachtelung ist in Java das Paket. Pakete sind Sammlungen von Klassen, die zur besseren Stukturierung zusammengefasst werden. Diese äußere Schachtelung wird in Java jedoch nicht in Form von Blöcken angegeben, sondern es wird mit einem *package-statement* die Klasse einem Paket zugeordnet. Entsprechend wird

[3] www.cs.princeton.edu/~appel/modern/java/CUP/

auch in der Eingabennotation des Implementierungsvorschlages diese Schachtelung nicht mit Blöcken realisiert, sondern es wird, wie bei einer Java-Referenz auf eine Klasse, das Paket als Pfad vor dem Klassennamen angegeben.

Eine Klassendeklaration ist in mehrere Felder aufgeteilt. Eine exakte Umsetzung der Schachtelung würde eigentlich einen Block „Klasse" fordern, in dem sich dann ein seperater Block „Attribute" und ein Block „Methoden" befinden. Zugunsten der gewohnten Java-Umgebung wird diese innere Schachtelung aufgehoben. So können Attribute und Methoden in einem Block stehen (siehe Tabelle 4.5). Der Enwickler hat so die Freiheit, Felder und die zu ihnen gehörenden Methoden zusammen zu gruppieren oder die Aufteilung „erst Felder, dann Methoden" zu belassen. Das Fehlen dieser Schachtelung fordert allerdings einen Eingriff in die Umsetzung der Methodensignaturen. In der UML ist die Angabe von Parametern bei Methoden optional. Würde in der hier verwendeten Notation einfach die Klammer hinter dem Methodennamen weggelassen werden, um anzudeuten, dass noch keine Parameter spezifiziert sind, wäre die Angabe mit einer Attributdeklaration identisch. Es könnte nicht mehr zwischen Attributen und Methoden unterschieden werden. Aus diesem Grund muss bei Methoden immer ein Klammernpaar stehen, wobei das Unterlassen der Parameterspezifikation, wie in Tabelle 4.7 zu sehen, mit der Zeichenfolge „(...)" ausgedrückt wird.

Die äußerste Schachtelung, die in Java mit Blöcken realisiert wird, ist die Klasse, der eine eigene Datei gewidmet wird. Bei Klassendiagrammen werden mehrere Klassen in einer Datei gespeichert, bzw. es können in einer Datei auch mehrere Diagramme gespeichert werden. Deshalb ist die äußere Schachtelung eine UML/P-Datei. Die Elemente der UML/P sind in ihr als äußerste Block-Schachtelungen enthalten. Klassendiagramme werden dabei mit dem Schlüsselwort „class-diagramm" gekennzeichnet (siehe Tabelle 4.4).

			EBNF
⟨UML/P⟩	::=	{ ⟨ClassDiagram⟩ | ⟨Java⟩ | ⟨OCL⟩ | ⟨ObjectDiagram⟩ }*	
⟨ClassDiagram⟩	::=	`class-diagram` ⟨Identifier⟩ { ⟨CDTypeDeclaration⟩* }	
⟨CDTypeDeclaration⟩			
	::=	⟨CDClassDeclaration⟩	
	|	⟨CDInterfaceDeclaration⟩	
	|	⟨AssociationDeclaration⟩	
	|	⟨SubclassingDeclaration⟩	

Tabelle 4.4: EBNF-Definition Klassendiagramme

Die zusätzlichen Kurzangaben bei Klassendiagrammen können weitgehend übernommen werden. So können die Modifikatoren *private*, *protected*, *public* und *readonly* durch die jeweiligen Kurzformen „-", „#", „+" und „?" ersetzt werden. Die Angabe der Vollständigkeit des Diagramms ("©","...") kann allerdings nicht so einfach übernommen werden. Das zuvor genannte Fehlen eines seperaten Blocks für Methoden und Attribute fordert hier eine explizite Kennzeichnung der Zugehörigkeit. Dies wird, wie in Tabelle 4.5 zu sehen, mittels eines Schlüsselwortes vor der Angabe

```
⟨CDClassDeclaration⟩                                                    EBNF
        ::= ⟨Stereotype⟩* ⟨CDModifier⟩* class ⟨Name⟩
            { extends ⟨Name⟩ }^opt
            { implements ⟨Name⟩,^1−* }^opt { ⟨Completeness⟩ }^opt
            { ⟨CDClassBodyDeclaration⟩* } ⟨Tag⟩*
⟨CDClassBodyDeclaration⟩
        ::= fields ⟨Completeness⟩ ;
        |   methods ⟨Completeness⟩ ;
        |   ⟨Attribute⟩
        |   ⟨MethodSignature⟩

⟨CDInterfaceDeclaration⟩
        ::= ⟨Stereotype⟩* ⟨CDModifier⟩* interface ⟨Name⟩
            { extends ⟨Name⟩ }^opt { ⟨Completeness⟩ }^opt
            { ⟨CDInterfaceBodyPart⟩* } ⟨Tag⟩*
⟨CDInterfaceBodyPart⟩
        ::= fields ⟨Completeness⟩ ;
        |   methods ⟨Completeness⟩ ;
        |   ⟨Attribute⟩
        |   ⟨MethodSignature⟩
```

Tabelle 4.5: EBNF-Definition Klassen

gelöst. Des weiteren ist die Kurzangabe "©" kein ASCII-Zeichen, was die Benutzung in einem internationalen Umfeld einschränken könnte. Es werden daher stattdessen die ausgeschriebenen Schlüsselwörter „complete" und „incomplete" verwendet (siehe Tabelle 4.8).

Elemente, die bereits im Java-Sprachstandard existieren und in die hier verwendete Notation eingebunden wurden, werden in Abschnitt 4.3 näher beleuchtet. Besonders zu beachten ist hier die Unterscheidung zwischen *Name* und *Identifier*. Bei ersterem ist eine Pfadangabe zugelassen, wohingegen zweiterer wirklich nur eine Bezeichnung ist.

```
⟨AssociationDeclaration⟩                                                EBNF
        ::= { assoc | aggreg } ⟨Identifier⟩^opt =
            ⟨Type⟩ { as ⟨Identifier⟩ }^opt { [ ⟨Kardinalitaet⟩ ] }^opt
            { qualified-by ⟨Name⟩ }^opt
            { -- | -->> | <<-->> }
            { qualified-by ⟨Name⟩ }^opt
            { [ ⟨Kardinalitaet⟩ ] }^opt ⟨Type⟩ { as ⟨Identifier⟩ }^opt
            ⟨Tag⟩* ;

⟨SubclassingDeclaration⟩
        ::= extends ⟨Identifier⟩^opt = ⟨Type⟩ -->> ⟨Type⟩ ;
```

Tabelle 4.6: EBNF-Definition Assoziationen

⟨Attribute⟩	::=	⟨Stereotype⟩* ⟨CDModifier⟩* ⟨TypeCardinality⟩^{opt} ⟨VariableDeclarator⟩ ⟨Tag⟩* ;	**EBNF**
⟨MethodSignature⟩	::=	⟨Stereotype⟩* ⟨CDModifier⟩* ⟨TypeVoid⟩^{opt} ⟨Identifier⟩ { ⟨FormalParameters⟩ | (...) } []* ⟨Throws⟩ ⟨Tag⟩* ;	

Tabelle 4.7: EBNF-Definition Attribute und Methodensignaturen (angelehnt an [Rum02])

⟨TypeCardinality⟩	::=	⟨Type⟩ { [⟨Kardinalitaet⟩] }^{opt}	**EBNF**
⟨CDModifier⟩	::=	⟨Modifier⟩ | ⟨Sichtbarkeit⟩ | `readonly`	
⟨Completeness⟩	::=	`complete` | `incomplete`	
⟨Kardinalitaet⟩	::=	`1` | `0..1` | *	
⟨Sichtbarkeit⟩	::=	`+` | `#` | `?` | `-`	
⟨Stereotype⟩}	::=	`<<` ⟨Identifier⟩ `>>`	
⟨Feature⟩	::=	{ ⟨FeatureEntry⟩*, }	
⟨FeatureEntry⟩	::=	⟨Identifier⟩ { `=` ⟨Expression⟩ }^{opt}	

Tabelle 4.8: EBNF-Definition Sonderangaben bei Klassendiagrammen (teilweise übernommen aus [Rum02])

4.3 Coderümpfe

Wie schon in vorhergehenden Kapiteln angesprochen, ist bei der ersten Version des AMTools eine Eingabemöglichkeit für Java-Code in Form von Coderümpfen unumgänglich. Diese Coderümpfe können getrennt den anderen Daten der zugehörigen Klasse abgelegt werden, um die Übersichtlichkeit von z.B. Klassendiagrammen zu gewährleisten.

Die hier angegebenen Coderümpfe erweitern den Java-Standard im wesentlichen in zwei Punkten. Die Blöcke können, wie in Tabelle 4.10 zu sehen, *assert*-Anweisungen enthalten, die auf OCL zurückgreifen. Da das Schlüsselwort „assert“ allerdings im Java-Standard 1.4 bereits vergeben ist, wird hier „ocl“ verwendet. Des weiteren können OCL-Vor- und Nachbedingungen in die Rümpfe integriert werden (siehe Tabelle 4.9). Mit ihnen lässt sich auf eine relativ einfache Weise zur Laufzeit des generierten Code überprüfen, ob eine gewünschte Funktionalität eingehalten wird.

Letztlich gilt es noch zu erwähnen, dass die geschriebenen Coderümpfe automatisch vom AMTool überarbeitet werden, um sie mit den anderen UML/P-Elementen konform in eine Javaklasse einbetten zu können. Es werden zum Beispiel alle direkten Attributzugriffe durch Aufrufe von *Setter*- und *Getter*-Methoden ersetzt, um die Einhaltung der in Klassendiagrammen gesetzten Zugriffsrechte, die über den Java-Standard hinausgehen, zu gewähren.

Im Groben entspricht der Aufbau einer Coderumpf-Einheit dem Aufbau einer Java-Klassendatei. Neben den eben genannten Erweiterungen werden alle Deklarationen gestrichen, die nicht zum Erstellen eines Methodenrumpfes benötigt werden. Es bleiben noch die *import-statements*, die in den Coderümpfen aufgelöst werden müssen, und die Angabe eines Klassennamens, der zur Zuweisung der Methode zu einer Klasse benötigt wird. Durch das Wegfallen des Package-Statements können in einer Datei auch Methoden zu verschiedenen Klassen gespeichert werden. Die zugehörige Klasse wird, wie bei den Klassendiagrammen, über den Pfad vor der Namensangabe ermittelt.

Die Regeln zum Erstellen von *inner classes*, also Klassen, die in einem Methodenrumpf eingebettet sind, wurden komplett aus den Coderümpfen heraus genommen. In Klassendiagrammen muss die gesamte Struktur eines Systems darstellbar sein. Gibt man dem Entwickler die Möglichkeit, Klassen in Methoden zu verstecken, so verhindert man jeglichen Zugriff von aussen auf diese Klasse. Jede inner Class kann in Java auch als frei verfügbare Klasse erstellt werden, der dann auch mit anderen Mitteln der UML/P Eigenschaften zugewiesen werden können.

⟨Java⟩	::=	`java` { ⟨JavaCode⟩ } `;`	**EBNF**
⟨JavaCode⟩	::=	{ `import` ⟨Name⟩ `.*`^{opt} `;` }*	
		{ ⟨Constraint⟩ | ⟨MethodGroupDeclaration⟩ }*	
⟨MethodGroupDeclaration⟩			
	::=	`class` ⟨Name⟩	
		`{` { ⟨MethodRumpfDecl⟩ | ⟨ConstructorRumpfDecl⟩ }* `}`	
⟨MethodRumpfDecl⟩			
	::=	⟨MethodHeader⟩ ⟨OperationConstraintBody⟩ ⟨Block⟩	
⟨ConstructorRumpfDecl⟩			
	::=	⟨ConstructorHeader⟩ ⟨OperationConstraintBody⟩ ⟨Block⟩	

Tabelle 4.9: EBNF-Definition erweiterte Coderümpfe (aus [Rum02])

Der innere Teil der Coderümpfe enstpricht neben den eben erwähnten Abweichungen dem Java-Standard. Die hier in EBNF angegebene Grammatik für Java lässt einiges mehr zu, als der Java-Standard vorgibt. So sind nach der EBNF-Vorgabe zum Beispiel alle Kombinationen von Modifikatoren zugelassen, wohingegen bei Java die Angabe einer Sichtbarkeit eine weitere Sichtbarkeitsangabe ausschließt. Ebenso wird die Einhaltung der Groß- und Kleinschreibung entspechend dem *Coding-Standard* nicht vorgegeben. Die Überprüfung der exakten Einhaltung des Java-Standard kann jedoch nicht vollständig mit der EBNF erfolgen. Der Anteil der syntaktischen Überprüfungen, der von der EBNF in den Java-Code verschoben wird, hängt von dem Aufwand zur Formulierung ab. Oftmals empfielt sich bereits bei etwas komplexeren Überprüfungen das Verschieben in den Programmcode, da der Aufwand in der EBNF sehr schnell steigt.

4.4 OCL

Die OCL/P in diesem ersten Prototypen ist im Vergleich zu der OCL/P aus [Rum02] sehr eingeschränkt. Im aktuellen Zustand (siehe Tabelle 4.14) wird mit ihr lediglich gezeigt, wie sie in das Gesamtkonzept der UML/P eingebunden werden kann. So fehlen momentan noch alle OCL-spezifischen Aussagemöglichkeiten zu logischen

EBNF

⟨Block⟩	::=	{ ⟨Statement⟩* }
⟨Statement⟩	::=	final^{opt} ⟨Type⟩ ⟨VariableDeclarator⟩$^{1-*}_{,}$;
	|	⟨Block⟩
	|	;
	|	⟨Expression⟩ ;
	|	switch (⟨Expression⟩) { ⟨SwitchPart⟩* }
	|	do ⟨Statement⟩ while (⟨Expression⟩) ;
	|	break ⟨Identifier⟩^{opt} ;
	|	continue ⟨Identifier⟩^{opt} ;
	|	return ⟨Expression⟩^{opt} ;
	|	synchronized (⟨Expression⟩) ⟨Block⟩
	|	throw ⟨Expression⟩ ;
	|	try ⟨Block⟩ ⟨CatchClause⟩* { finally ⟨Block⟩ }^{opt}
	|	⟨Identifier⟩ : ⟨Statement⟩
	|	if (⟨Expression⟩) ⟨Statement⟩ { else ⟨Statement⟩ }^{opt}
	|	while (⟨Expression⟩) ⟨Statement⟩
	|	for (⟨ForInit⟩^{opt} ; ⟨Expression⟩^{opt} ; ⟨Expression⟩$^*_{,}$) ⟨Statement⟩
	|	ocl ⟨OCLExpr⟩ { : ⟨OCLExpr⟩ }^{opt} ;
⟨SwitchPart⟩	::=	{ case ⟨Expression⟩ : | default : }$^{1-*}$ ⟨BlockStatement⟩*
⟨CatchClause⟩	::=	catch (⟨Type⟩ ⟨Identifier⟩ []*) ⟨Block⟩
⟨ForInit⟩	::=	⟨Expression⟩$^{1-*}_{,}$ | final^{opt} ⟨Type⟩ ⟨VariableDeclarator⟩$^*_{,}$

Tabelle 4.10: EBNF-Definition Block und Statement (aus [Rum02])

Zusammenhängen. Die Ausdrücke zum Formulieren einer Bedingung sind entweder jene aus Java oder die direkte Angabe eines Objektdiagrammes. Diese Angabe kann jedoch noch nicht in andere Ausdrücke eingebettet werden.

Die OCL kann auf vier Wegen in die UML/P eingebettet werden. Wie in Tabelle 4.14 zu sehen, kann ein eigener OCL-Teil, der unabhängig von den anderen Elementen der UML steht, als äußerste Block-Schachtelung deklariert werden. Hierzu wird das Schlüsselwort „ocl“ verwendet und in dem darauf folgenden Block steht dann eine Reihe von *Constraints*. Möchte man Methodenrümpfe beschreiben, so kann man auch die Constraints direkt in den Java-Teil der UML/P einbinden (siehe Tabelle 4.9). Es können entweder beliebige Constraints eigenständig im Block stehen oder es können, als dritte Einbindungsmöglichkeit, die Vor- und Nachbedingungen direkt vor dem Block einer Methode stehen. Letzlich kann auch die „ocl“-Anweisung in Methodenblöcken die Gültigkeit von OCL-Ausdrücken prüfen.

In der UML können OCL-Angaben direkt in Klassen- und Objektdiagrammen stehen. Die Einbindung der OCL lässt die Diagramme schnell unübersichtlich wirken. Deshalb wird in der UML/P keine OCL in Diagrammen zugelassen. Die Ausdrücke müssen getrennt in einem OCL-Block abgelegt werden. Ein weiterer Grund, der gegen eine Einbettung der OCL in Diagramme spricht, ist, dass die OCL als Text abgelegt wird und weitere Versionen des AMTools vielleicht einmal eine graphische

EBNF

```
⟨Primary⟩            ::= ( ⟨Expression⟩ )
                     |   ⟨Literal⟩
                     |   { ⟨Primary⟩ . }^opt ⟨Identifier⟩ ⟨Arguments⟩^opt
                     |   { ⟨Primary⟩ . }^opt this ⟨Arguments⟩^opt
                     |   { ⟨Primary⟩ . }^opt super ⟨Arguments⟩
                     |   ⟨Primary⟩ [ ⟨Expression⟩ ]
                     |   super . ⟨Identifier⟩ ⟨Arguments⟩^opt
                     |   new ⟨Name⟩ []^1-* { ⟨VariableInitializer⟩*_, ,^opt }
                     |   new ⟨Name⟩ { [ ⟨Expression⟩ ] }^1-* []*
                     |   { ⟨Primary⟩ . }^opt new ⟨Name⟩ ⟨Arguments⟩
                     |   ⟨Primary⟩ . class
⟨PrefixOp⟩           ::= ++ | -- | + | - | ~ | !
                     |   ( ⟨Type⟩ )
⟨PostfixOp⟩          ::= ++ | --
⟨Arguments⟩          ::= ( ⟨Expression⟩*_, )

⟨Expression⟩         ::= ⟨PrefixOp⟩* ⟨Primary⟩ ⟨PostfixOp⟩*
                     |   ⟨Expression⟩ ⟨InfixOp⟩ ⟨Expression⟩
                     |   ⟨Expression⟩ instanceof ⟨Type⟩
                     |   ⟨Expression⟩ ? ⟨Expression⟩ : ⟨Expression⟩
                     |   ⟨Expression⟩ ⟨AssignmentOperator⟩ ⟨Expression⟩
⟨InfixOp⟩            ::= * | / | % | + | - | << | >> | >>>
                     |   < | > | <= | >= | == | !=
                     |   & | ^ | | | && | ||
⟨AssignmentOperator⟩
                     ::= = | *= | /= | %= | += | -= | <<= | >>=
                     |   >>= | &= | ^= | |=
```

Tabelle 4.11: EBNF-Definition Ausdrücke in Java (größtenteils aus [Rum02])

Eingabe für die Diagramme zulassen. Spätestens dann sollten die beiden UML/P-Elemente getrennt werden, da ein Diagramm sonst kaum noch zeichenbar wäre.

Eine Erweiterung des hier verwendeten Teils der OCL/P zu der vollen Version ist mit wenigen Änderungen zu realisieren. Der *OperationConstraintBody* müsste noch um eine „let“-Anweisung ergänzt werden, mit der Hilfsvariablen erzeugt werden können, die in später verwendeten Bedingungen benutzt werden können. Komplett ersetzt werden müssten die OCL-Ausdrücke (OCLExpr). In der OCL/P kann in jedem Teilausdruck wieder auf die gesamte Auswahl der OCL-Ausdrücke zurückgegriffen werden. Diese Ersetzung ist aber wegen des geringen Umfangs des jetzigen OCL-Ausdrucks nicht aufwendig.

4.5 Objektdiagramme

Objektdiagramme werden ähnlich wie Klassendiagramme aufgebaut. Die äußerste Schachtelung, das Diagramm selbst, enthält Objekte und Links. Sie wird mit dem

EBNF

⟨FieldDeclaration⟩	::=	⟨Modifier⟩* ⟨Type⟩ ⟨VariableDeclarator⟩$^{1-*}_{,}$;
⟨VariableDeclarator⟩	::=	⟨Identifier⟩ `[]`* { = ⟨VariableInitializer⟩ }^{opt}
⟨VariableInitializer⟩	::=	⟨Expression⟩
	|	{ ⟨VariableInitializer⟩$^{*}_{,}$,^{opt} }
⟨MethodDeclaration⟩		
	::=	⟨MethodHeader⟩ { ⟨Block⟩ | ; }
⟨MethodHeader⟩	::=	⟨Modifier⟩* ⟨TypeVoid⟩ ⟨Identifier⟩
		⟨FormalParameters⟩ `[]`* ⟨Throws⟩
⟨ConstructorDeclaration⟩		
	::=	⟨ConstructorHeader⟩ ⟨Block⟩
⟨ConstructorHeader⟩		
	::=	⟨Modifier⟩* ⟨Identifier⟩ ⟨FormalParameters⟩ ⟨Throws⟩
⟨FormalParameters⟩	::=	({ `final`^{opt} ⟨Type⟩ ⟨Identifier⟩ `[]`* }$^{*}_{,}$)
⟨Throws⟩	::=	{ `throws` ⟨Type⟩$^{1-*}_{,}$ }^{opt}

Tabelle 4.12: EBNF-Definition Attribute, Methoden und Konstruktoren in Java (aus [Rum02])

Schlüsselwort „object-diagram“ gekennzeichnet. Die Objekte selbst werden mit dem Schlüsselwort „object“ gekennzeichnet. Ihnen kann optional ein Name und ein Typ gegeben werden, und in ihnen kann ein Feld mit Attributen ausgefüllt werden. Die Attributangaben sind identisch mit denen der Klassendiagramme. Ebenso kann auch die Angabe zur Vollständigkeit („complete“, „incomplete“) wie bei den Klassendiagrammen benutzt werden.

Dem Namen, den Ojekte haben, fällt bei einer schriftlichen Form der Objektdiagramme ein besonderes Gewicht zu. Können bei graphischen Versionen namenlose Objekte problemlos referenziert werden, indem zum Beispiel der Pfeil eines Links an der Objektzeichnung endet, geht dies hier nicht. Möchte man einen Link definieren, der zwei Objekte verbindet, so ist es fast immer nötig, die Enden des Links einem Objekt zuzuordnen. Ausser Namen steht in dieser Notation keine Zuordnungsmöglichkeit zur Verfügung. Auch wenn die Namen im generierten Code nie auftauchen, sollte in Objektdiagrammen bei jedem Objekt ein Name stehen.

4.6 Umsetzung der EBNF in LR(1)

Die Umsetzung der meisten EBNF-Regeln in LR(1) ist nach einem festen Vorgehen möglich. In diesem Abschnitt wird zuerst auf den Aufbau einer Cup-Grammatik eingegangen und im weiteren wird dann das Vorgehen veranschaulicht.

Eine Grammatik für Cup kennt, genau wie die EBNF, *Terminale* und *Nichtterminale*. Ein Terminal entspricht dabei einem Symbol, welches direkt vom Lexer übernommen wird. In der hier verwendeten Grammatik wird es durchgehend in großen Buchstaben geschrieben. Nichtterminale sind Symbole, die durch Reduzieren einer Regel entstanden sind. Jedes Nichtterminal muss genau einmal auf der linken Seite

EBNF

⟨TypeVoid⟩	::=	⟨Type⟩ | `void`
⟨Type⟩	::=	⟨BasicType⟩
	|	⟨Name⟩
	|	⟨Type⟩ `[]`
⟨Name⟩	::=	$\langle\text{Identifier}\rangle^{1-*}_{.}$
⟨BasicType⟩	::=	`boolean` | `byte` | `int` | ...
⟨Identifier⟩}	::=	{ `a` | ...| `z` | `A` | ...| `Z`| `_` }⟨Digit⟩*
⟨Digit⟩	::=	`a` | ...| `z` | `A` | ...| `Z` | `0` | ...| `9` | `_`
⟨Number⟩	::=	{ `0` | ...| `9` }$^{1-*}$
⟨Modifier⟩	::=	`public` | `protected` | `private`
	|	`static` | `final` | `abstract`

Tabelle 4.13: EBNF-Definition Bezeichner und Modifikatoren (aus [Rum02])

einer Regel, in der es definiert wird, vorkommen. Nichtterminale werden hier durchgehend klein geschrieben, es sei denn, es handelt sich um eine der Sonderformen, auf die später eingegangen wird.

Jede Regel in Cup hat eine linke Seite, auf der ein Nichtterminal steht. Auf der rechten Seite kann eine Folge von Terminalen oder Nichtterminalen stehen. Der einzige Operator, der bei Cup-Regeln verwendet werden kann, ist der *oder*-Operator („|“). Mit ihm können einem Nichtterminal mehrere Ableitungsregeln zugewiesen werden. Eine Regelangabe könnte folgendermaßen aussehen:

Cup

```
nichtterminal ::= TERMINAL
       | einsymbol anderessymbol
       ;
```

Bei dieser Angabe können nun zwei Regeln angewendet werden. Cup unterscheidet bei diesen beiden Regeln in ihrer Auswirkung. Die obere Regel verkürzt die Eingabe nicht. Wird sie angewandt, so findet lediglich eine Umbenennung des Symbols statt. Bei Cup wird dies eine *shift*-Operation genannt. Bei der zweiten Regel werden mehrere Symbole zu einem neuen Symbol zusammengefasst, die Länge der Eingabe wird also reduziert. Dementsprechend wird die Anwendung einer solchen Regel *reduce*-Operation genannt. Es gibt des weiteren ein paar Sonderformen der Regeln:

Cup

```
nichtterminal ::= /*empty*/
       | einsymbol nichtterminal
       ;
```

Man kann bei Cup sogar das eingegebene Wort verlängern. Dies geschieht mit der *leeren Regel*. Wird das „nichtterminal“ von einer anderen Regel aufgerufen, so verhält sich Cup immer so, als würde dieses existieren und gibt, falls keine andere Regel passt, den Wert von /*empty*/ zurück. Genau wie bei der EBNF sind auch bei Cup-Grammatiken rekursive Regeln möglich. So stellt das oben angegebene „nichtterminal“ eine beliebig lange Liste von „einsymbol“ dar.

EBNF

⟨OCL⟩	::=	`ocl` `{` { ⟨Stereotyp⟩* ⟨Constraint⟩ ⟨Merkmal⟩* `;` }* `}`
⟨Constraint⟩	::=	⟨Invariant⟩ | ⟨OperationConstraint⟩
⟨Invariant⟩	::=	⟨ClassContext⟩^{opt} `inv` ⟨InvariantName⟩^{opt} `:` ⟨OCLExpr⟩$^*_{;}$
⟨OperationConstraint⟩		
	::=	`context` ⟨OperationContext⟩ ⟨OperationConstraintBody⟩
⟨OperationConstraintBody⟩		
	::=	{ `pre` ⟨InvariantName⟩^{opt} `:` ⟨OCLExpr⟩$^*_{;}$ }^{opt}
		{ `post` ⟨InvariantName⟩^{opt} `:` ⟨OCLExpr⟩$^*_{;}$ }^{opt}
⟨ClassContext⟩	::=	{ `context` | `import` }
		{ ⟨ClassOrInterface⟩ ⟨Identifier⟩^{opt} }$^*_{,}$
⟨OperationContext⟩	::=	⟨Stereotyp⟩* ⟨OperationSignature⟩
		⟨Throws⟩ ⟨Merkmal⟩*
⟨OperationSignature⟩		
	::=	⟨MethodSignature⟩ | ⟨ConstructorSignature⟩
⟨MethodSignature⟩	::=	⟨TypeVoid⟩^{opt} { ⟨ClassOrInterface⟩ `.` }^{opt}
		⟨Identifier⟩ ⟨FormalParameters⟩ `[]`*
⟨ConstructorSignature⟩		
	::=	`new` ⟨ClassOrInterface⟩ ⟨FormalParameters⟩
⟨ClassOrInterface⟩	::=	⟨Name⟩
⟨InvariantName⟩	::=	⟨Identifier⟩
⟨OCLExpr⟩	::=	⟨Expression⟩ | `OD.Identifier`

Tabelle 4.14: EBNF-Definition OCL-Bedingungen (siehe [Rum02])

Mit den bisher genannten Cup-Regeln lassen sich nun die weiteren Regeln der EBNF umsetzen. Die einfachste Umsetzung ist die der Angabe $\{\langle x\rangle\ \langle y\rangle\}^{opt}$. Man erstellt ein weiteres Nichtterminal, mit dem die Angabe „^{opt}“ umgesetzt wird. Dieses Nichtterminal trägt den Namen des optionalen Elements mit einem angehängten „Opt“. Eine derartige Regel sieht dann folgendermaßen aus:

Cup

```
nichtterminal ::= ... xyOpt ... ;
xyOpt ::= /*empty*/
      | x y
      ;
```

Ebenso einfach ist die Umsetzung der Regel $\{\langle x\rangle\ \langle y\rangle\}^{1-*}_{\langle sep\rangle}$. Es muss also eine nicht leere Liste der Zeichen x und y entstehen, die durch den Seperator „sep“ getrennt sind. Auch hier erstellt man wiederum ein neues Nichtterminal, in dem diese Angabe umgesetzt wird. Es wird mit der Endung „List“ markiert. Ist in der EBNF kein Seperator angegeben, so kann in der gleich angegebenen Lösung diesen einfach weggelassen werden.

Cup

```
nichtterminal ::= ... xyList ... ;
xyList ::= x y
       | x y sep xyList
      ;
```

EBNF

```
⟨ObjectDiagram⟩   ::= object-diagram ⟨Identifier⟩ { ⟨ODTypeDeclaration⟩* }
⟨ODTypeDeclaration⟩
                  ::= ⟨ODObjectDeclaration⟩ | ⟨LinkDeclaration⟩

⟨ODObjectDeclaration⟩
                  ::= ⟨Stereotype⟩* ⟨CDModifier⟩* object
                      ⟨Identifier⟩^opt { : ⟨Name⟩ }^opt { ⟨Completeness⟩ }^opt
                      { ⟨ODObjectBodyDeclaration⟩* } ⟨Tag⟩*
⟨ODObjectBodyDeclaration⟩
                  ::= ⟨Attribute⟩ | fields ⟨Completeness⟩ ;
⟨LinkDeclaration⟩ ::= link ⟨Identifier⟩^opt = ⟨Identifier⟩ { as ⟨Identifier⟩ }^opt
                      { qualifier ⟨Expression⟩ }^opt
                      { -- | -->> | <<-->> }
                      { qualifier ⟨Expression⟩ }^opt
                      ⟨Identifier⟩ { as ⟨Identifier⟩ }^opt
                      ⟨Tag⟩* ;
```

Tabelle 4.15: EBNF-Definition Objektdiagramme

Bei der Umsetzung eines „*“ wird unterschieden, ob ein Seperator existiert oder nicht. Die Regel { ⟨x⟩ ⟨y⟩ }* kann mit einer einfachen Rekursion realisiert werden. Das Nichtterminal zur Ausführung der Regel wird mit der Endung „Star“ gekennzeichnet.

Cup

```
nichtterminal ::= ... xyStar ... ;
xyStar ::= /* empty */
       | x y xyStar
      ;
```

Ist ein Seperator gewünscht, also die EBNF-Angabe die Form $\{\langle x\rangle\ \langle y\rangle\}^*_{\langle sep\rangle}$ haben, so muss eine etwas aufwendigere Umsetzung stattfinden. Die Idee dabei ist, dass man entweder nichts oder eine nichtleere Liste hat. Dies hat zur Folge, dass in der unten gezeigten Grammatik zwei neue Nichtterminale benötigt werden.

Cup

```
nichtterminal ::= ... xyStar ... ;
xyStar ::= /* empty */
       | xyList
      ;
xyList ::= x y
       | x y sep xyList
      ;
```

Mit den gegebenen Regeln lassen sich alle Sonderformen der EBNF meistens gut realisieren. Es gibt noch einen Sonderfall zu beachten, der nicht mehr direkt an die EBNF gekoppelt ist. Bei Ausdrücken können *Infixoperatoren* vorkommen. Diese haben meistens eine Assoziativität und eine Bindungsstärke an die sie umgebenden Elemente. So wird zum Beispiel bei einer Summe und einem Produkt immer zuerst das Produkt berechenet und danach die Summe, wenn keine Klammern existieren.

Ebenso wird, wenn mehrere Summen hintereinander stehen, immer zuerst die am weitesten links stehende Summe berechnet. In Cup kann dies mit einer *precendence*-Angabe vorgegeben werden.

Cup

```
precendence left PLUS;
precendence left TIMES;

expression ::= NUMBER
            | expression PLUS expression
            | expression TIMES expression
;
```

Die Angabe der Bindungsstärke wird durch die Reihenfolge der *precedence*-Angaben gegeben. Je später eine Angabe erfolgt, desto stärker ist die Bindung. Ist bei so einer Regel keine Präzedenz oder Assoziativität angegeben, meldet Cup fast immer einen *reduce-reduce*-Konflikt, denn es ist nicht schlüssig, ob die linke oder die rechte Seite zusammengefasst werden muss.

4.7 Lösungsvorschläge für Probleme bei der Umsetzung

Wie bereits erwähnt, kann nicht immer eine Umsetzung der ENBF mit den in dem vorigen Abschnitt angegebenen Regeln realisiert werden. Die äussert sich bei Cup mit Fehlermeldungen, die entweder auf *shift-reduce*- oder auf *reduce-reduce*-Konflikte hinweisen. Bei beiden hat der Parser mehrere Regeln zur Auswahl, von denen er sich, trotz der Vorausschau eines Zeichens, beim Lesen von links nach rechts nicht für eine entscheiden kann.

In diesem Abschnitt werden anhand von Beispielen ein paar spezielle Probleme aufgegriffen und deren Lösung vorgestellt. Ziel ist es, den Entwicklern von weiteren Grammatikteilen einen Einblick in die Lösungsmethoden zu geben. Die hier angegebenen Lösungsvorschläge sind jeweils nicht zwingend die einzigen, bzw. die besten für des jeweils exemplarisch gegebene Problem. Trotzdem lässt sich mit ihnen ein Lösungsweg gut aufzeigen. Es wird nun jedesmal das Problem dargestellt und dann eine Lösungsmethode mit der konkreten Lösung angegeben.

4.7.1 Auslagern in Programmiersprache

In Tabelle 4.7 kann bei der Angabe eines Attributs der Typ optional sein. Dieser besteht aus einem Namen (Name), hinter dem eckige Klammernpaare stehen können, die die Tiefe einer Schachtelung in *Arrays* angibt. Hinter dem Typ folgt dann ein Identifikator (Identifier), der den Namen des Attributs angibt, und eventuell wiederum eine Menge von eckigen Klammern zur Angabe von Arrays.

Cup

```
attribute ::= ... typevoidOpt identifier arrayStar ...
  | ...;
...
type ::= name
  | type array
  | ... ;

name ::= identifier
  | ... ;

identifier ::= IDENTIFIER;
array ::= ARRAY;
```

Gibt man nun dem LR(1)-Parser eine Token-Folge wie „IDENTIFIER ARRAY IDENTIFIER“, so liest der Parser zuerst den linken Identifier und weiss durch das Vorausschauen, dass als nächstes ein Token ARRAY kommt. Der Parser muss nun entscheiden, ob er den Identifier zu einem Nichtterminal „name“ shiftet oder ob er den Identifier belässt, um später die Regel für ein Attribut anzuwenden. Das Problem besteht darin, dass zwei Regeln dieselben Terminale als Eingabe akzeptieren und zusätzlich eine Regel der Anfang der anderen sein kann. Der ausschlaggebende Punkt ist in diesem Fall, dass die Typangabe optional ist.

Die hier vorgeschlagene Lösung sieht vor, das Problem, das eigentlich der Parser-Generator lösen sollte, in Java-Code auszulagern. Die Grammatik des Parsers wird nun so modifiziert, dass sie zumindest die in der EBNF gegebenen Sprache akzeptiert. Bei Anwendung der Regel wird dann eine Java-Methode aufgerufen, die den abstrakten Syntaxbaum so aufbaut, als wäre die aus der EBNF gegebene Regel umgesetzt worden. So bleibt ein derartiger Eingriff ausserhalb des Parsers transparent.

Möchte man in diesem Beispiel die Entscheidung, ob nach dem Lesen des ersten Identifikators eine Typangabe oder ein Attributname vorliegt, vom Parser-Generator wegnehmen, so kann man dies durch Einsetzen einer beliebigen Folge von Typdefinitionen tun. Die Folge „identifier arrayStar“ ist in einer Typdefinition enthalten. Die neue Regel zu einem Attribut könnte dann so aussehen:

Cup

```
attribute ::= ... typeStar ... ;
```

Der Umfang der Sprache, der mit dieser Regel akzeptiert wird, ist wesentlich größer geworden. Wird diese Regel ausgeführt, so muss die dabei aufgerufene Java-Routine dieses „typeStar“-Nichtterminal zerlegen und den abstrakten Syntaxbaum so aufbauen, als wäre der Parser aus den zuerst gegebene Regeln gebaut worden.

Das Auslagern eines Problems in Java-Code ist ein Lösungsweg, der quasi immer funktioniert, denn es stehen alle Berechnungsmöglichkeiten von Java zur Verfügung. Trotzdem sollte ein solches Vorgehen weitestgehend vermieden werden, da die Übersichtlichkeit und Wartbarkeit der Grammatik sehr darunter leiden kann. Findet sich aber kein anderer Weg, so ist hiermit eine sichere Variante gewählt.

4.7.2 Präzisierung der Grammatik

Die in Tabelle 4.11 gegebene Grammatik lässt bei Ausdrücken die Möglichkeit des *Typecast* zu. Eine in Klammern gestellte Typangabe kann vor einem beliebigen Ausdruck stehen. Sieht man sich die Grammatik hier nochmal in teilweise vereinfachter Form an, so kann die Folge „LPAR IDENTIFIER RPAR“ sowohl zu einem *Primary* als auch zu einem *Typecast-Operator* reduziert werden.

Cup

```
expression ::= prefixopStar primary postfixopStar
        | ... ;

primary ::= LPAR expression RPAR
        | identfier    // vereinfachte Darstellung
        | ... ;

postfixop ::= PLUSPLUS
        | ... ;

prefixop ::= PLUSPLUS
        | LPAR type RPAR
        | ... ;
type ::= identifier  // Vereinfachte Darstellung
| basetype
| ...;
```

Der Paser, der die eben genannte Zeichenfolge gelesen hat, versucht durch Vorausschau eines Zeichens zu entscheiden, welche Regel er anwenden muss. Steht als folgendes Zeichen das Terminal PLUSPLUS, so könnte es ebenso hinter dem *Typecast* als weiterer Prefixoperator wie auch hinter dem Ausdruck als Postfixoperator stehen. Der Parsergenerator meldet in diesem Falle einen *reduce-reduce*-Konflikt.

Die EBNF-Definition erlaubt meistens mehr sprachliche Konstrukte, als in der tatsächlichen Sprache zulässig sind. Unter Umständen ist es möglich, mit einer Grammatik, die eine gezielte, engere Fassung der Sprache beschreibt, Probleme der Regelzuordnung zu lösen. Es kann also durch mehr Wissen über die Sprache entschieden werden, welche Regel gültig ist.

In diesem speziellen Fall ist es in Java nicht möglich, jeden Typecast mit dem Präfixoperator PLUSPLUS zu kombinieren. Java lässt eine solche Kombination nur mit den Basistypen zu (siehe Tabelle 4.13). Des weiteren findet in der Liste der Präfixoperatoren höchstens ein Typecast statt (siehe [BH98]). Die Umsetzung dieser Eigenschaften findet in mehreren Schritten statt. In einem ersten Schritt wird der Typecast hinauf zu der „Expression“-Regel geholt. Das neue Regelwerk sieht folgendermaßen aus:

Cup

```
expression ::=
          prefixopStar cast prefixopStar primary postfixopStar
        | prefixopStar primary postfixopStar
        | ... ;

prefixop ::= PLUSPLUS
  //      | LPAR type RPAR Regel fehlt hier
        | ... ;

cast ::= LPAR type RPAR;
```

Nun kann vor jedem Ausdruck nur noch ein Typecast stattfinden. Es kann aber immer noch hinter jedem Typecast der Operator PLUSPLUS stehen. In einem zweiten Schritt wird der Typecast in zwei Teile aufgeteilt. Wird zu einem Basistypen gecastet, so kann der Parser dies anhand der Terminale für diese Typen sofort von einem Ausdruck unterscheiden. Wird jedoch etwas anders gecastet, so muss dies aus einem Namen und einer beliebig langen Folge von eckigen Klammern bestehen. Um die Entscheidung eindeutig zu halten, darf es in diesem Falle nicht die Möglichkeit geben, dass hinter dem Cast ein PLUSPLUS steht. Die neue Regel lautet also:

Cup

```
expression ::=
          prefixopStar basecast prefixopStar primary postfixOpStar
        | prefixopStar objectcast prefixopnoplusStar primary postfixOpStar
        | prefixOpStar primary postfixOpStar
        | ...;

basecast ::= LPAR basetype RPAR;
objectcast :: LPAR name arrayStar RPAR;
```

Eine derartige Lösung macht die Grammatik zwar wesentlich schwerer lesbar, aber bei einer entsprechenden Dokumentierung können so Konflikte gelöst werden, ohne dass dazu Java-Code nötig ist. Findet sich eine solche Lösung, ist sie fast immer einer Java-Lösung vorzuziehen, da so für die Regel weiterhin alle Funktionen des Parsers, wie zum Beispiel Fehlererkennung, genutzt werden können.

Kapitel 5

Auswertung des abstrakten Syntaxbaumes

Hat man mittels Lexer und Parser einen abstrakten Syntaxbaum aufgebaut, so müssen die Informationen, die in ihm enthalten sind, für den Generator aufbereitet werden. Die Aufbereitung ist in drei Teilgebiete getrennt: die Überprüfung der Konsistenz der Informationen im Baum, die automatische Ergänzung von Informationen und die Bereitstellung der Informationen in einer für den Generator verwertbaren Darstellung. Diese drei Gebiete sind häufig in ihrer Funktionalität verzahnt, oftmals kann mit der Erfüllung der einen Aufgabe eine andere mit minimalem Mehraufwand gelöst werden.

Wie in Abschnitt 5.2 gezeigt, wird zugunsten der Übersichtlichkeit trotzdem versucht, eine Trennung der Aufgaben vorzunehmen. Dabei werden die jeweils berechneten Werte durch eine Verschachtelung der Module gemeinsam genutzt. Die Idee ist, dass ein Modul immer auf die berechneten Ergebnisse eines anderen Moduls zurückgreifen kann. So muß kein Wert mehrmals berechnet werden, bzw. es ist auch kein redundanter Code nötig. Ein klassisches Beispiel für ein Modul, dessen Werte in Folge benutzt werden, ohne dass etwas neu berechnet werden muss, ist das Hilfsmodul für den Aufbau der *Symboltabelle*. Mit dieser werden die Elemente des abstrakten Syntaxbaumes an *Namensräume* (*Namespaces*) gebunden, anhand derer sich im späteren Verlauf die Module zusätzlich orientieren können (siehe Abschnitt 5.2.1).

5.1 Implementierungsvarianten

Die *semantische Analyse* des Syntaxbaumes kann relativ komplexe und vielfältige Aufgaben umfassen. Deshalb gibt es zur Erstellung der Auswertungsmodule kaum vorgefertigte Generatoren oder Skriptmechanismen, die die Entwicklungsarbeit wesentlich erleichtern würden. Meistens können nur allgemeine Generatoren, wie zum Beispiel das AMTool, bei der Aufwandsbewältigung hilfreich sein. Mit dem Ziel, die *Module* einmal mit dem AMTool aus UML zu generieren, wird hier kein anderes

Hilfsprogramm berücksichtigt.

In dem in dieser Arbeit vorgeschlagenen Lösungsansatz werden zwei Arten von Modulen aufgegriffen, die teilweise verschiedene Anforderungen und Verhaltensweisen haben. Es wird nun dargestellt werden, wie diese Modularten im Allgemeinen funktionieren und für welche Aufgaben sie jeweils geeignet sind. Dabei wird lediglich auf den erstellten Java-Code eingegangen, nicht aber auf die Erstellung selbst.

Das Ausgangsproblem zur Auswertung des abstrakten Syntaxbaumes ist, dass nach dem Parsen einem Programm lediglich die Wurzel des Baumes zur Verfügung steht. Ein Programm, das auf Informationen im Baum zugreifen will, muss sich mindestens einmal von der Wurzel aus zu den einzelnen Knoten im Baum hangeln. Dabei kann es, solange es den Inhalt eines Knoten noch nicht selbst betrachtet hat, zum Navigieren im Baum lediglich auf die Struktur, die mit der Grammatik gegeben ist, zugreifen. Sobald der Inhalt eines Knoten bekannt ist, kann die Navigation mit Indexstrukturen optimiert werden.

5.1.1 Treewalker (Eager-Evaluation)

Eine naheliegende Lösung, einen Baum zu durchschreiten, sind sogenannte *Treewalker*. Sie wandern einmal mit einem Tiefensuche-Algorhytmus durch den gesamten Baum, besuchen also jeden Knoten und stellen dort Berechnungen an. Hat ein Treewalker seine Arbeit getan, so kann ein nächster Treewalker mit einer anderen Funktionalität auf dem Baum angewendet werden.

Der Vorteil von Treewalkern liegt in ihrer Orientierung an der Eingabestruktur. So können Treewalker immer, ohne irgendwelche Informationen über den Inhalt des Baumes zu haben, ihren Gang beginnen. Der Nachteil ist ihr starres Verhalten. Die Werte werden *eager* berechnet, das heißt, dass unabhängig, ob eine Auswertung eines Knoten nötig ist oder nicht, dieser auf jeden Fall besucht wird. Wird also im späteren Verlauf des Programmes nur ein Teil der von diesem Modul extrahierten Information benötigt, so ist Rechenzeit verschwendet worden.

Die eben genannte Entkopplung von Informationsbedarf und Informationsextrahierung fordert eine Speicherung der gewonnenen Daten. Ein Programm sollte nach der Anwendung eines Moduls durch einen simplen Variablenzugriff seine Informationen erhalten, ohne dass weitere Berechnungen stattfinden müssen. Hierbei stellt sich die Frage, wo und wie die Daten untergebracht werden. Eine Möglichkeit wäre, ein eigenes Datenobjekt zu erstellen, in dem die Daten abgelegt werden können. Dieses Objekt könnte an die nächsten Module weitergereicht werden, die dann wiederum durch den Baum laufen und das Datenobjekt nutzen. Die Knoten des abstrakten Syntaxbaumes würden so von den Treewalkern unabhängig bleiben, dass heißt, sie müssten keine Felder zur Speicherung der Daten enthalten. Allerdings besteht in den meisten Fällen ein enger Zusammenhang der Daten mit den Elementen des Baumes. Der Aufwand, die einzelnen Daten den Baumelementen zuzuweisen, rechtfertigt also oft die Speicherung der Daten direkt in den Elementen des Baumes. Bei weiteren Treewalkerdurchläufen können diese gespeicherten Informationen unmittelbar genutzt werden.

Möchte man mehrere Treewalker zur Lösung von Aufgaben einsetzen, so müssen diese kombiniert werden. Treewalker werden normalerweise einer Reihenfolge nach aufgerufen, da die Gefahr einer Endlosschleife bei einem gegenseitigen Aufruf der Walker relativ hoch ist. Benötigt ein Treewalker eine Information von einem in der Reihenfolge weiter hinten liegenden, so müssen mehrere Durchläufe der Reihenfolge gestartet werden. Da sich jeder Treewalker dann wiederum einmal durch den ganzen Baum hangelt, steigt der Rechenaufwand massiv. Im ungünstigsten Fall, in dem jedes Modul von seinem Nachfolgemodul abhängig wäre, müssten bei n Modulen n Durchläufe der Modulfolge stattfinden, was einer n-fachen Berechnungsdauer entspräche.

Die Verwendung von Treewalkern macht meistens nur Sinn, wenn drei Bedingungen erfüllt sind. Erstens sollte noch keine andere Navigationsmöglichkeit im Baum vorliegen, also das Programm ohnehin einmal durch den Baum laufen müssen, um den Inhalt einmal zu sehen. Zum anderen sollte die Aufgabe, die das Modul zu erfüllen hat, eine strukturorientierte und nicht inhaltsorientierte Aufgabe sein, denn bei inhaltsorientierten Aufgaben ergibt sich oben genanntes Performanzproblem. Und drittens sollte standardmäßig ein Großteil der durchlaufenen Knoten etwas zum Ergebnis beitragen. Ein Beispiel für eine strukturorientierte Aufgabe ist der in Abschnitt 5.2.1 beschriebene Symboltabellenaufbau.

5.1.2 Lazy-Evaluation

Für inhaltsorientierte Aufgaben empfiehlt sich ein anderer Lösungansatz. Der einfachste Weg, unnötige Berechnungen zu verhindern, ist, die Werte erst dann zu berechnen, wenn sie definitiv benötigt werden. Im Vergleich zu einem Treewalker, der seine Ergebnisse immer *eager* im Speicher ablegt, erfolgt bei der *Lazy-Evaluation* die Informationsbeschaffung immer über den Aufruf einer Methode, welche erst dann das Ergebnis berechnet. Braucht eine Methode für die Berechnung ihrer Werte auch welche aus anderen Modulen, so kann sie diese mit einem gezielten Methodenaufruf ermitteln. Die Module sind somit zeitlich und kausal nicht mehr getrennt. Lediglich die Methoden in den Modulen stehen, bedingt durch die gegenseitigen Aufrufe, in einer kausalen Ordnung.

Die Berechnung zur Informationsbeschaffung wird durch Aufrufe des Generators ausgelöst. Da Informationen mehrfach angefragt werden können, empfiehlt es sich, einen *Cache-Mechanismus* einzubauen, der nach einer einmaligen Berechnung das Ergebnis speichert. Dieser Mechanismus kann an allen Stellen im AMTool verwendet werden, an denen Informationen des abstrakten Syntaxbaumes ausgewertet werden. Die Konstruktion einer Cache-Lösung sieht von der Idee her folgendermaßen aus:

Java

```
Object MyObject = null;
...
  public Object getMyObject() {
    if (Object==null) {
      return calcMyObject;
    }
    return calcMyObject();
  }

  private void calcMyObject() {
   ... getOtherObject();
   return ... ;
  }
```

Jede berechnende *calc*-Methode wird damit nur ein einziges Mal aufgerufen werden. Werden die Module größer und unübersichtlicher, steigt die Gefahr, dass sich kausale Schleifen bilden. Tritt eine derartige Schleife auf, bleibt das Programm in der Schleife hängen und meldet einen *Stack-Overflow*. Eine zur Fehleranalyse hilfreiche Lösung ist der Einbau einer Abfrage, die keinen rekursiven Aufruf einer calc-Methode zulässt. Die in dem Prototypen verwendeten Lösungen weichen von diesem Muster meistens ab, funktionieren aber alle nach dieser Kernidee. So ist es zum Beispiel bei Basistypen oft nicht möglich, anhand des Wertes zu erkennen, ob eine Berechnung bereits stattgefunden hat, da alle Werte als berechnetes Ergebniss zugelassen sind. In diesen Fällen wird eine boolsche Hilfsvariable zugezogen, die den Berechnungsstatus angibt.

Der Zugriff des Generators auf die Informationen funktioniert bei Treewalkern relativ einfach. Die Informationen werden einfach aus dem Datenobjekt geholt, egal von welchem Modul sie stammen. Bei Lazy-Auswertung müssen jedoch die Methoden der einzelnen Module aufgerufen werden. So müsste der Code-Generator immer wieder verschiedene Modulschnittstellen für verschiedene Daten ansprechen. Eine Möglichkeit, dieses Problem zu beheben, ist die Informationsbeschaffung über nur eine Schnittstelle. Diese wäre über die die Einführung eines Schnittstellenobjektes, welches die Schnittstellen der Module nach aussen hin zusammenfasst. Ähnlich wie ein Datenobjekt, ist dieses jedoch relativ aufwendig zu erstellen, und es müsste für jede Modulkombination ein eigenes Objekt erstellt werden. Ein weitaus einfacherer Weg, die Schnittstellen zusammenzufassen, ist es, den Java-Mechanismus der Vererbung zu nutzen. Dabei übernimmt das erbende Modul die Schnittstelle des darüberliegenden und fügt seine Methoden hinzu. Der Generator muss so nur noch auf das unterste Modul der Hierachie zugreifen und bekommt dort all seine Informationen.

Die Kombination der Module durch Vererbung hat Auswirkungen auf die Parametrisierung. Die Parametrisierung (siehe Abschnitt 3.2.2) der Module kann durch Erweitern des untersten Moduls in der Vererbunghierachie kompakt in einer Klasse vorgenommen werden, vorausgesetzt, die Parametermethoden unterscheiden sich namentlich. Schwierig hingegen werden größere Änderungen an Modulen, die in der

Vererbunghierachie weiter oben stehen. Wären die Module nur über Referenzen aneinander gebunden, so könnten höherstehende Module durch von ihnen erbende Module ersetzt werden. Sind die Module aber bereits durch Vererbung gebunden, so bleibt bei einer Moduländerung in einer höheren Hierachiestufe nur der Austausch der gesamten Modulreihe.

Die Nutzung der Lazy-Evaluation, also die Berechnung von Information auf Anfrage, hat zwei Folgen. Zum einen wird die Latenzzeit bei einer Informationsanfage größer, da die Werte erst berechnet werden müssen, und zum anderen muss das aufrufende Programm wissen, welche Werte benötigt werden. Erstere Folge hat im Falle des AMTools keine sichtbare Auswirkung, da es nicht interaktiv ist und es so keine Rolle spielt, wann innerhalb des Laufes etwas berechnet wird. Die zweite Folge stellt den Generator eine neue Aufgabe. Er muss von sich aus wissen, welche Werte verfügbar sind. Dazu müsste er mindestens einmal den Baum komplett abschreiten, was eigentlich eine klassische Aufgabe eines Treewalkers ist. Im AMTool wird dem Generator mittels eines Treewalkers eine Basis an Navigationsinformationen zur Verfügung gestellt, anhand derer er dann die gewünschten Informationen effizient von den Lazy-Evaluation-Modulen holen kann.

5.2 Beschreibung der Module

Dieser Abschnitt behandelt eine Modulkombination, die Informationen zur Generierung von Java-Code bereitstellt. Dabei wird berücksichtigt, dass die Module so gestaltet und verbunden werden, dass sie mit relativ geringem Aufwand auch zur Generierung anderer Sprachen genutzt werden können. Eine einfache Anwendung wäre zum Beispiel eine, die die Modelle zum Zwecke eines Vergleichs in eine einheitliche Form bringt, also aus UML/P wieder UML/P generiert. Dafür kann man die ersten zwei der in diesem Kapitel beschriebenen Module verwenden.

Zur Auswertung des abstrakten Syntaxbaumes werden sowohl Treewalker- als auch Lazy-Evaluation-Module genutzt. In dem AMTools werden generell zuerst Treewalker angewendet, die ihre Ergebnisse im Baum speichern. Für die Java-Generierung wird genau ein Treewalker verwendet. Danach wird der Generator aufgerufen, der sowohl auf die im Baum stehenden Daten, als auch auf die Lazy-Module zugreift.

5.2.1 Aufbau der Symboltabelle

Grundlage für den Compilerbau sind Programmiersprachen. In diesen ist es üblich, dass ein Element immer mit einem gleichen Namen bezeichnet wird. Der Name, der das jeweilige Element beschreibt, wird als Symbol für das Element verwendet. Zu jedem Symbol können Informationen gespeichert werden, die sich auf das bezeichnete Element beziehen. So kann zum Beispiel bei einem Symbol für ein Attribut der zugehörige Typ gespeichert werden. Die Symbole werden in einer Tabelle abgelegt, aus der sich andere Methoden Informationen zu den beschriebenen Elementen holen können. Eine solche Tabelle heißt *Symboltabelle*.

Bei dem AMTool hat die Symboltabelle einen weiteren Verwendungszweck. Die Lazy-Module benötigen oft Informationen zu den in der UML/P beschriebenen Elementen. Diese Informationen können aber über den ganzen abstrakten Syntaxbaum verteilt sein. Abhilfe schafft hier eine Indexstruktur, die auf alle Knoten im Baum verweist, die das gleiche Element beschreiben. Diese Indexstrukturen kann man in den Symbolen ablegen. Mit ihnen können sowohl der Generator als auch die Lazy-Module effizient auf die Information im Baum zugreifen. Das AMTool baut mit diesem Modul lediglich die Symboltabelle mit dieser Indexstruktur auf, um den Lazy-Modulen eine weitere Navigationsmöglichkeit im Baum zu bieten.

Bei den alten funktionalen Programmiersprachen konnte ein Name nur ein Element repräsentieren. Heute können bei fast allen Sprachen gleiche Namen, in Abhängigkeit des Kontextes, für verschiedene Elemente stehen. So kann der Name „xyz“ einmal für eine Stringvariable in einer Klasse stehen und an anderer Stelle in einer Methode einen Integer-Wert in einer Schleife darstellen. Der Kontext, der bei gleichen Namen bestimmt, zu welchem Element sie gehören, ist der sogenannte Namensraum. So wird zum Beispiel bei Java für jede neue Klasse ein neuer Namensraum erstellt. Um eindeutig zu sein, muss sich ein Symbol aus einem Namen und dem zugehörigen Namensraum zusammensetzen.

Die Angabe des Namensraumes in der Symboltabelle kann auf zwei Weisen erfolgen. Entweder ist die Symboltabelle flach, d.h. es gibt nur eine Tabelle, in der alles steht, oder sie ist eine Kombination von mehreren Tabellen. Bei ersterer Lösung setzt sich der Name eines Symbols aus dem des Elements und des Namensraums zusammen. Taucht im Quellcode ein Name auf, so wird ermittelt, in welchem Namensraum er sich befindet, und dieser wird vorne dem Name hinzugefügt. Das Ergebnis ist eine relativ große Tabelle, bei der jeder Namensraum einen eigenen Namen haben muss.

Bei der zweiten Lösung hat jeder Namensraum eine eigene Tabelle für die Namen der Elemente. Die Symbole sind so auf mehrere Tabellen verteilt, was die Tabellen kleiner und somit besser handhabbar werden lässt. Die Tabellen der Namensräume stehen meist in einer baumförmigen Hierachie zueinander, über die navigiert werden kann. Es ist bei dieser Tabellenstruktur im Vergleich zur flachen Struktur sehr einfach, in den Namensräumen nach oben zu navigieren, da man von der jeweiligen Tabelle lediglich zur jeweiligen *Parent*-Tabelle springen muss. Für das AMTool stellt das einen Vorteil dar, da in Java durch die Schachtelung der Blöcke ein Großteil der Navigation nach oben stattfindet. Tabelle 5.1 stellt ein solches Hierachiekonzept dar. Ein weiterer Vorteil dieser Lösung ist, dass es nicht immer nötig ist, einem Namensraum einen eigenen Namen zu geben. Nur wenn von anderen Tabellen auf einen Namensraum zugegriffen werden soll, so muss dieser, um referenziert werden zu können, einen Namen haben. Blöcke, die keinen Namen haben, wie zum Beispiel bei einer „for“-Schleife in einer Methode, können ohne großen Aufwand in der Namensraumhierachie eingeordnet werden. Ein solcher Block-Namensraum kann und soll nicht von aussen referenziert werden, da er eine gekapselte Einheit ist.

Je nach späterem Verwendungszweck kann die Namensraumhierachie bereits an die Zielsprache angepasst werden. Werden zum Beispiel zwei Klassendiagramme verglichen, so muss der Namensraum „Diagramm“ existieren. Ist andererseits Java das

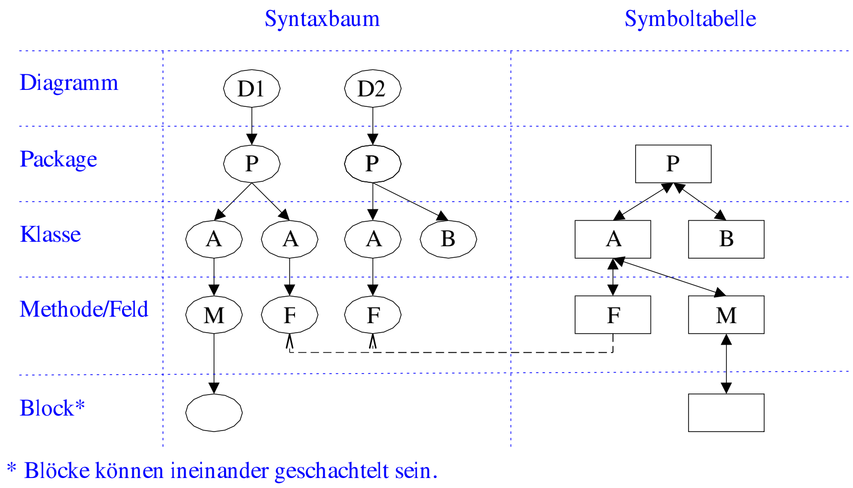


Tabelle 5.1: Die Symboltabelle als Index für Elemente

Ziel, so sollte der Namensraum „Diagramm" nicht auftauchen, da sich sonst die Module die Information aus den verschiedenen Diagrammen zusammensuchen müssten.

Der Aufbau der Symboltabelle ist mit seinen Anforderungen und Eigenschaften prädestiniert für ein Treewalker-Modul. Es müssen nahezu alle Knoten unabhängig von ihrem Inhalt betrachtet werden, da die Module, die die Symboltabelle nutzen, darauf aufbauen, dass diese den abstrakten Syntaxbaum vollständig erfasst. Die Struktur des Syntaxbaumes wird den Namensräumen zugeordnet. Die einzigen Informationen, die zum Lösen dieser Aufgabe benötigt werden, sind der abstrakte Syntaxbaum und die Namensraumhierachie.

Einige Compiler nutzen keinen Treewalker zum Symboltabellenaufbau, sondern inkludieren diese Aufgabe bereits in den Parser, der, bedingt durch den Aufbau des abstrakten Syntaxbaumes, ebenfalls alle Knoten besucht. Es sprechen, in dem speziellen Falle des AMTools, einige Punkt gegen die Integration in den Parser. Im Gegensatz zu einem Treewalker geht der Parser nicht top-down, sondern bottom-up durch den Syntaxbaum. Da im unteren Bereich des Baumes oft noch nicht klar ersichtlich ist, zu welchem Namensraum ein Name gehört, bekommt dieser einen eigenen Namensraum, der irgenwann mit einem anderen verschmolzen wird. Dies ist wesentlich aufwendiger als eine von vornherein klare Zuweisung der Namensräume. Des weiteren kann bei einem Treewalker die Reihenfolge der Abarbeitung der Elemente festgelegt werden, was auch wieder einige Vereinfachungen im Code mit sich ziehen kann. Letztlich gilt es noch zu erwähnen, dass man den Entwickler nicht mit einer überladenen Darstellung der Grammatik belasten soll, indem dort Berechnungen anstellt werden.

5.2.2 Überprüfen der Konsistenz der UML/P-Modelle

Beschreibt man eine Software mit UML/P, so können Angaben zu bestimmten Elementen in verschiedenen Modellen gemacht werden. Zum Beispiel kann ein Entwickler bei einer Business-Anwendung den Einkauf und ein anderer den Verkauf modellieren. Auch wenn die Modelle jeweils in sich konsistent sind, so muss für die Codegenerierung eines gemeinsamen Systems trotzdem ein Abgleich zwischen den Modellen stattfinden. Dabei werden sowohl komplementäre Angaben zusammengefasst, als auch differierende Angaben weitestmöglich aufeinander abgestimmt. Ein Generator, der von diesem *Konsistenz-Modul*(KM) Informationen bezieht, bekommt sie entweder konsistent zu den Modellen passend oder das Modul meldet einen Fehler.

Dabei wird zwischen *weichen* und *harten* Fehlern unterschieden. Ein weicher Fehler kann von dem Modul so korrigiert werden, dass der generierte Code zwar nicht mehr mit allen Modellen übereinstimmt, aber die Abweichungen nur einen kleinen Teil der Modelle betreffen. In den meisten Fällen findet eine solche Korrektur mittels Parameterfunktionen statt, die einen *Default*-Wert bestimmen. Ein typisches Beispiel ist die später beschriebene Korrektur verschiedener Sichtbarkeitsangaben zu einem Attribut. Harte Fehler hingegen führen dazu, dass ein Modell nicht mehr übersetzt werden kann. So ist es zum Beispiel unmöglich, einem Attribut zwei nicht zueinander kompatible Typen zuzuweisen. Bei nicht korrigierbaren Fehlern liefert das Modul den Wert *null*.

Es werden nicht immer alle Informationen des KM benötigt. Es liegt somit nahe, die Implementierung in Form eines Lazy-Moduls zu realisieren. Die Schnittstelle des Moduls sind Methoden, die als Übergabeparameter ein Symbol einer Symboltabelle haben. Anhand dieses Symbols, das als Index genutzt wird, werden dann die Knoten des abstrakten Syntaxbaumes nach Informationen durchsucht. Das Ergebnis der Berechnungen wird aus Optimierungsgründen im abstrakten Syntaxbaum gespeichert. Um es nicht in jedem zu einem Element gehörenden Knoten speichern zu müssen, wird hier die Konvention eingehalten, berechnete Ergebnisse nur in dem Knoten abzulegen, der in der Indexliste an erster Stelle steht.

Im Folgenden werden ein paar Funktionalitäten des Moduls vorgestellt und näher betrachtet.

Modifikatoren

Existieren mehrere UML/P-Modelle, die eine Klasse, ein Feld oder eine Methode beschreiben, so kann schnell der Fall eintreten, dass die Modifikatoren nicht bei jeder Beschreibung komplett angegeben sind. Ziel des KM ist es, bei unterschiedlichen Angaben einen Kompromiss zu finden, der die Modelle weitestgehend einhält. Bei kontroversen Angaben können die Modelle nicht sauber übersetzt werden, aber es können bei Modifikatoren immer Korrekturen durchgeführt werden, die mit den Modellen zum größten Teil vereinbar sind. Bei diesen Korrekturen gibt das KM eine Warnung aus, und verhält sich, als wäre überall eine konsistente Angabe gegeben.

Die Modifikatoren lassen sich in zwei Gruppen unterteilen. Die einfacher zu handhabende Gruppe sind diejenigen, bei denen das Element mit einer 0-1-Semantik eine Eigenschaft hat oder nicht. Zu diesen Modifikatoren gehören *abstract*, *final* und *static*. Wird im Verlauf der Auswertung festgestellt, dass ein solcher Modifikator

	Value-Getter	*Default-Parameter*
static	getStaticModifier(Field) getStaticModifier(Method)	defaultIsStatic(Field) defaultIsStatic(Method)
final	getFinalModifier(Field)	defaultIsFinal(Field)
public *protected* *readonly* *private*	getAccessModifier(Field) getAccessModifier(Method) getAccessModifier(Class)	defaultAccessPriority(Field) defaultAccessPriority(Method) defaultAccessPriority(Class)

Tabelle 5.2: Die Methoden zur Modifikatorermittlung

nicht an allen Angaben zu einem Element steht, so wird eine Methode aufgerufen, die einen Default-Wert berechnet. In den meisten Fällen sind diese Methoden trivial und dienen lediglich zu Parametrisierung des Moduls. Bei der zweiten Gruppe der Modifikatoren existiert eine mehrwertige Abstufung. Zu ihnen gehören *public*, *protected*, *readonly* und *private*. Auch hier wird im Falle kontroverser Angaben eine Methode zur Default-Berechnung aufgerufen. Es ist jedoch sinnvoller, statt einem einzigen Default-Wert eine tendenzielle Richtung vorzugeben, bei der, zum Beispiel im Falle der Sichtbarkeiten, die großzügigere Angabe gewählt wird. Tabelle 5.2 gibt die Methoden und ihre Parametrisierungsmöglichkeiten an.

Typen

Genau wie bei Modifikatoren können auch bei Typen in Modellen verschiedene Angaben stehen. Die Ermittlung eines gemeinsamen Typs, der mit allen Modellen vereinbar ist, erweist sich jedoch als sehr aufwendig. Eine vollständige Typisierung müsste neben der Prüfung auch neue Typen erstellen können. So könnte ein Element in einem Modell mit dem Interface „Iterator“ beschrieben werden und in einem anderen mit dem Interface „Set“. Die Erstellung eines neuen Typs, der alle Angaben erfüllt, stößt schnell an die Grenze des Machbaren.

Etwas einfacher, aber immer noch sehr aufwendig, ist die Suche nach existierenden Typen, die alle Vorgaben erfüllen. Dazu müsste man bei jeder Klasse und jedem Interface die Vererbungs- bzw. Implementierungshierachie nach oben hin durchgehend und mit den Anforderungen der Modelle vergleichen. Eine Klasse, die alle entsprechenden Typen vereint, kann dann ohne Einschränkungen verwendet werden. Allerdings dauert eine derartige Suche aufgrund der großen Menge an Klassen, die Java hat, relativ lange. Außerdem werden in den meisten Fällen in den Modellen neue Typen beschrieben, die nicht mit den bereits existenten Klassen übereinstimmen, das heißt, das hier beschriebene Vorgehen würde oft nichts helfen.

Übrig bleibt eine sehr einfache Variante der Typüberprüfung, die in dem KM realisiert ist. Werden in Modellen verschiedene Typen genannt, so muss in einem der Modelle ein Typ sein, der zu den Typen der anderen Modelle kompatibel ist. Man sucht quasi den kleinsten gemeinsamen Nenner. Die Ermittlung dieses Typs beschränkt sich also auf die Auswahl der in den Modellen genannten Typen. Zwischen den gegebenen Typen wird die Beziehung über Interfaces und Erweiterung (Vererbung) geprüft. Wird ein Typ gefunden, der mit allen Modellen vereinbar ist, so kann er ohne Einschränkungen benutzt werden. Werden als Typ Klassen angegeben, die nur binär vorhanden sind, findet die Untersuchung mittels der *java.lang.reflect*-API statt. Diese erkennt jedoch nur Klassen, die *public* sind. Klassen, die eine eingeschränke Sichtbarkeit haben, werden nicht in die Typüberprüfung eingebunden.

Diese letzte einfache Art der Typüberprüfung kann ebenfalls zur Kontrolle und Ermittlung des Typs von Ausdrücken dienen. Es kann zum Beispiel kontrolliert werden, ob bei einer Zuweisung die linke Seite mit der rechten Seite vereinbar ist.

Sammeln von Information

Neben den eben genannten Prüfungen gibt es einige Methoden, die Informationen lediglich sammeln, und dabei davon ausgehen, dass diese konsistent sind. So sind zum Beispiel Stereotypen und Merkmale vom Sprachstandard relativ unabhängig und können frei definiert werden. Ihre Wirkung und ihre Zusammenhänge sind dem Modul nicht bekannt. So könnte eine Klasse mit dem Stereotyp „Customerdata" immer ein Merkmal fordern, in dem der Ersteller der jeweiligen Klasse registriert ist. Diese Zusammenhänge werden aber in der UML/P nicht explizit ausgedrückt und dementsprechend nicht berücksichtigt. Das Modul sammelt einfach alle Angaben ein und stellt sie dem Generator unbearbeitet zur Verfügung.

Zusätzlich zu der konsistenten Information zu einzelnen Elementen benötigt der Generator auch Listen, die Elemente mit bestimmten Eigenschaften aufführen. Beim Zusammenstellen der Listen dürfen aber nur die Elemente genommen werden, bei denen die gewünschte Eigenschaft mit allen Modellen vereinbar ist. Beispielsweise könnten alle Attribute gesucht werden, die die Kardinalität [1] haben. In diesem Fall müsste bei jedem Feld mit der Angabe [1] geprüft werden, ob es in einem anderen Modell *null* sein kann.

5.2.3 Behandlung unvollständiger Modelle

Bei allen Vorteilen, die die Abstraktion eines Gesamtbildes in Modellen hat, so hat sie folgenden Nachteil. Für eine Generierung von Java ist ein UML/P-Modell meist unterspezifiziert. Dieses Modul schließt nun die Informationslücke zwischen Modell und Implementierung. Es baut auf dem vorhergehenden KM auf. Es greift immer dann ein, wenn die gelieferte Information zwar konsistent, aber unvollständig ist. Die Ermittlung einer perfekten Implementierung ist nicht immer möglich und so wird versucht, eine Lösung zu finden, die zumindest die Vorgaben aus den Modellen erfüllt. Ähnlich dem KM werden auch hier einige Default-Werte über Parametrisierungsmethoden dem Modul vorgegeben.

Technisch gesehen erweitert dieses Modul mittels Vererbung das KM. Alle Methoden des KM, die bereits vollständige Informationen liefern, werden einfach übernommen. Diejenigen Methoden, bei denen noch Informationen ergänzen werden müssen, werden von diesem Modul überschrieben. Ein Zugriff auf die Methode des KM ist dann

	Value-Getter	*Default-Parameter*
initializer (für finals)	getInitializer(Field)	defaultNumberInitializer(Field) defaultObjectInitializer(Field) defaultBooleanInitializer(Field)
method-body	getBodyBlock(Method)	defaultNumberMethodBody(Method) defaultBooleanMethodBody(Method) defaultObjectMethodBody(Method)
type	getType(Field)	defaultType(Field)
methods	getMethods(Class)	

Tabelle 5.3: Die Methoden zur Ergänzung von Information

immer noch über das Schlüsselwort *super* möglich. Genau wie das KM gibt dieses Modul eine Warnung bei Ergänzungen aus, bzw. eine Fehlermeldung, wenn eine Vervollständigung nicht möglich ist.

Das Fehlen einer Typangabe im Modell ist ein Musterbeispiel für den Einsatz dieses Moduls. Es wird zuerst die Methode des KM aufgerufen und wenn diese kein Ergebnis liefert, so wird die Ermittlung des Typs aus dem Umfeld versucht. Findet zum Beispiel bei einem Attribut eine Zuweisung statt, so kann man davon ausgehen, dass das Attribut den Typ der rechten Seite der Zuweisung hat. Bei mehreren Zuweisungen muss, ähnlich der Typprüfung, derjenige Typ ausgewählt werden, der immer zuweisbar ist. Wie schon bei dem KM erwähnt, ist der Aufwand zur Kreation neuer Typen sehr hoch und wird deshalb nicht berücksichtigt.

Kapitel 6

Der Generator

Die Aufgabe des Codegenerators ist es, aus der aufbereiteten Information des abstrakten Syntaxbaums einer UML/P-Datei Code zu generieren (siehe Tabelle 6.1). Je stärker das AMTool in die Entwicklung einer Software eingebunden ist, desto besser muss es handhabbar und an den Prozess anpassbar sein. Besonders der Generator muss Anforderungen an Aspekte wie Lebendigkeit, Wartbarkeit und Erweiterbarkeit erfüllen (siehe 6.2).

In diesem Kapitel werden nun im ersten Teil verschiedene Implementierungsvarianten verglichen. Der zweite Teil des Kapitels enthält eine Reihe von Vorschlägen zur Umsetzung verschiedener Teile der Modelle in Java. Eine komplette Beschreibung aller Umsetzungsmöglichkeiten würde den Rahmen dieser Arbeit sprengen, denn der Anwender des AMTools kann mit der Parametrisierung frei bestimmen, wie der zu generierende Code an seine Umgebung angepasst werden soll.

Codegenerierung ist der Vorgang zum Erzeugen von Code aus ausführbaren, konstruktiven Modellen.

Codegenerator ist ein Programm, das ein konstruktives Modell einer höheren Programmiersprache in eine Implementierung transformiert. Der generierte Code gehört entweder zum Produktionssystem oder zu der Menge der Tests.

Skript ist eine konstruktive Steuerung der Codegenerierung. Skripte parametrisieren den Codegenerator und erlauben damit plattform- und aufgabenspezifische Codegenerierung.

Template ist eine spezielle Form eines Skripts. Ein Template beschreibt Codemuster, die bei der Generierung durch konkrete Elemente des Modells ersetzt werden. Dabei wird typischerweise ein Makro-Ersetzungsmechanismus eingesetzt.

Tabelle 6.1: Begriffsdefinitionen zum Codegenerator ([Rum02])

Lebendigkeit : Änderungen am Modell sollten sofort im generierten Code wiedergegeben werden. Änderungen am Generierungsprozess sollten ebenfalls ohne ein neues Compilieren und starten des Generators möglich sein. Die Anpassbarkeit des Generators kann dabei von der Änderung bei Texteinrückungen bis hin zum Kreieren neues Codes für UML-Ausdücke, die nicht im Standardprodukt enthalten sind, reichen.

Wartbarkeit : Der Prozess der Codegenerierung selbst sollte bereits einen guten Eindruck davon geben, wie der erstellte Code am Ende aussehen wird. Das erleichtert die Entwicklung neuer generierter Programmstücke und erhöht die Anpassbarkeit und Lebensdauer der existierenden Programmstücke.

Erweiterbarkeit : Das Einführen neuer generierter Programmstücke sollte einfach und schmerzlos sein. Der Anwender sollte immer die Codegenerierung an seine speziellen Bedürfnisse anpassen können. Dabei sollte so weit möglich die Notwendigkeit eines Neustarts bzw. einer neuen Compilierung des Generators unterbunden werden.

Tabelle 6.2: Anforderungen an die Nutzbarkeit eines Codegenerators (Inhalt entnommen aus [SVB02])

6.1 Implementierungsvarianten

Wird der Codegenerator aus Java-Code erstellt, so werden die in Tabelle 6.2 genannten Anforderungen an die Lebendigkeit und Erweiterbarkeit nur schlecht erfüllt. Denn wenn an dem Generator etwas geändert werden soll, so muß man nach dem Schreiben der Klassen diese erst compilieren und dann in das Programm einbinden. Ein derartiger Aufwand wäre eigentlich nur bei sehr tiefen Eingriffen in die Funktionalität zu rechtfertigen. Ebeso ist es mit Java nahezu unmöglich, dem Aspekt der Wartbarkeit entgegenzukommen, da die Generierung mit Java-Befehlen stattfindet, die das Bild des zu generierenden Codes zerstückeln.

Ein Weg, diesen Anforderungen entgegenzukommen, liegt in der Verwendung von *Skripten*. Sie werden erst zur Laufzeit ausgewertet und haben somit im Gegensatz zu festem Programmcode den Vorteil, dass jede Änderung im Quelltext sofort das Verhalten des Werkzeugs beeinflusst. Die Anwender können so schnell und unkompliziert den Generator an ihre Bedürfnisse anpassen. Im Folgenden werden nun einige Skript-Möglichkeiten beleuchtet, die zur Generierung von Java-Code benutzt werden können. Dabei wird immer bewertet, inwiefern die jeweilige Möglichkeit die Anforderungen an den damit erstellten Generator erfüllt.

6.1.1 XSLT

Die *eXtensible Stylesheet Language Transformation*(siehe [Eck00]) ist ein XML-Dialekt, der immer größere Beachtung findet. Das haupstächliche Ziel dieser Sprache ist es, XML-Dokumente in einen anderen XML-Dialekt zu transformieren. So können zum Beispiel XML-Dokumente relativ einfach in HTML-Dokumente umgeformt werden. Sie ist keineswegs auf XML beschränkt und kann auch zur Generierung von Java verwendet werden. Auf Basis von XSLT ist eine Änderung der Generierungsvorschriften zur Laufzeit möglich und eine neue Übersetzung des AMTools nicht mehr nötig.

Die Wartbarkeit erfordert, dass der Anwender bereits bei der Erstellung der Übersetzungsvorschriften einen relativ guten Überblick darüber hat, wie das Ergebnis aussieht. Genau hier hat XSLT jedoch eine große Schwachstelle. Die XSLT-Befehle sind auf die Baumstruktur des zu transformierenden XML-Dokuments ausgerichtet. Dementsprechend ist der Aufbau des XSLT-Skripts nicht wie erfordert an dem zu generierenden Code orientiert, sondern an dem abstrakten Syntaxbaum, der für diese Verwendung in XML gespeichert werden könnte. XSLT-Vorschriften werden so für den Anwender des AMTools sehr unübersichtlich. Wie schlecht XSLT für den Menschen lesbar ist, zeigt folgendes Beispiel, welches aus dem unten gegebenen XML-File den String „private int myName;“ generiert.

XML

```
...
<attribute>
  <visibility> public </visibility>
  <type> int </type>
  <identifier> myName </identifier>
</attribute>
...
```

XSLT

```
...
<xsl:template match="attribute">
  <xsl:text>private </xsl:text>
  <xsl:apply-templates/>
  <xsl:text>;</xsl:text>
</xsl:template>

<xsl:template match="visibility">
</xsl:template>

<xsl:template match="type">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="identifier">
  <xsl:apply-templates/>
</xsl:template>
...
```

Ein weiterer Schwachpunkt von XSLT ist, dass es nicht mit den Java-Modulen kommunizieren kann, also nicht während der Transformation gezielt einzelne Informationen anfordern kann, sondern von vornherein alle Informationen zu den Modellen haben muss. Dies setzt eine vollständige Auswertung der Modelle voraus, deren Ergebnisse in dem zu transformierenden XML-Dokument abgelegt werden müssen. Der Generator benötigt meistens nicht alle Informationen einer vollständigen Auswertung. Die dann unnötig durchgeführten Berechnungen verringern die Performanz des AMTools.

6.1.2 Templates

Templates (siehe Tabelle 6.1), die ebenfalls wie XSLT hauptsächlich zur Erstellung von Dokumenten verwendet werden, geben Textrümpfe vor, die dann mit Informationen aus *Kontext-Objekten* angereichert werden, indem deren Methoden aufgerufen werden. Die Ermittlung der Modellinformation erfolgt nicht, wie bei XSLT, bereits beim Erstellen eines Eingabedokuments, sondern der Templatemechanismus holt Daten zu den Modellen „just-in-time“ bei Bedarf. Somit fällt die Generierung eines Eingabe-Dokuments für das Skript weg.

Templates sind des weiteren nicht an eine Baumstruktur gebunden. Die Navigation durch den abstrakten Syntaxbaum wird von den Java-Methoden des Kontext-Objektes übernommen. Man muss die Templates nicht auf die Struktur der Modelle ausrichten, sondern kann sie an dem zu generierenden Code orientieren. Die Wartbarkeit des Generators ist also gewährleistet. Im weiteren werden einige Templatemechanismen betrachtet, mit denen der Codegenerator erstellt werden kann.

Eigenkreation

Die beste Möglichkeit, den Anforderungen an den Codegenerator bei der Templateerstellung entgegenzukommen, ist sicherlich der Entwurf eines eigenen Templatemechanismus. So kann man optimal auf eine gute Lesbarkeit und Entwicklungseffizienz hinsteuern. Ebenso können die Aufgabenbereiche der Analyse der Modelle und der Generierung von Code klar getrennt werden. Eine Änderung der Parameter im Codegenerator hat damit keine Auswirkungen auf die restlichen Module des AMTools.

Der Aufwand zum Erstellen eines solchen Mechanismus ist jedoch relativ groß im Verhältnis zu den gewonnenen Vorteilen. Bei einer Überprüfung der Alternativen ist es deshalb vorzuziehen, einen vorgefertigten Templatemechanismus zu verwenden und eventuelle Abstriche in Kauf zu nehmen.

JavaServer Pages (JSP)

JSP, eine Mechanismus der *Apache-Gruppe*[1], erfüllt alle generellen Anforderungen an einen Templatemechanismus. Es bietet Textrümpfe, die mit Informationen ergänzt werden können. Allerdings erfolgen diese Ergänzungen in Form von eingebetteten Java-Code. In einem JSP-Template kann dementsprechend an beliebigen Stellen Java-Code stehen, mit dem in einem Template jede Berechnung durchgeführt werden könnte. Die Templates sind jedoch wegen der Modularisierung des AMTools ausschließlich für die Parametrisierung des Codegenerators gedacht, nicht aber für die Aufbereitung der Informationen aus den Modellen. Die Verlagerung von Aufgaben aus den Auswertungsmodulen in den Generator würde die Architektur des AMTools zerstören.

Ein weitere Nachteil von JSP ist, dass auch Skript-Funktionen, wie Schleifen oder Fallunterscheidungen nur in Java realisiert werden können. Bei der Generierung von

[1] www.apache.org

Java ergibt sich nun das Problem, dass die Java-Befehle verschiedene Bedeutungen haben können. So gehört zum Beispiel ein if-Schlüsselwort einmal zu einer Abfrage, die im generierten Code steht, und ein anderes mal gehört es zu einer steuernden Abfrage des Templates. Deshalb sollte von der Verwendung von JSP abgesehen werden.

Folgendes Beispiel soll die Doppeldeutigkeit der Java-Befehle illustrieren.

JSP

```
<%
  if (strVector.size() == 0 ) {
%>
  //There are no variables to check!
<%
  } else {
    for (Enumeration e = strVector.elements(); e.hasMoreElements();) {
      String str = (String)e.next();
%>
  if (x.isInstanceOf(<%= str %>)) {
    return <%= str %>.getClass().getName();
  }
<%
    }
  }
%>
```

Velocity

Velocity ist ein Template-Mechanismus, der ebenfalls zur Apache-Gruppe gehört. Er scheint im Vergleich zu den anderen am besten für die Erstellung des Codegenerators geeignet zu sein. Er hat hinsichtlich der Java-Generierung den wesentlichen Vorteil, im Skript Elemente zur Fallunterscheidungen und Schleifenbildung zu haben (siehe [VDT99]). So kommt innerhalb der Templates kein Java-Code für derartige Aufgaben zum Vorschein und der Anwender kann sich besser orientieren. Des weiteren können Informationen aus dem Quellcode des AMTools lediglich über Methodenaufrufe geholt werden. So ist bereits durch die Verwendung von Velocity ausgeschlossen, dass aus Versehen Aufgaben der Analyse-Module im Generator stattfinden. Das zuvor gezeigte Beispiel für ein JSP-Template sieht in Velocity folgendermaßen aus.

VM

```
#if( $strVector.size == 0 )
// There are no variables to check!
#else
 #foreach( $str in $strVector )
if (x.isInstanceOf( $str )) {
    return ${str}.getClass().getName();
}
 #end
#end
```

Ein weiterer Vorteil von Veloctiy ist ein *Macro-Mechanismus*, der bei der Entwicklung von Templates viele redundante Angaben verhindern kann. Die Templates lassen sich so besser warten und werden allein schon durch das Zusammenfassen von Teilen in einem Macro deutlich übersichtlicher. Folgendes Macro dient zum Beispiel zum Erstellen einer Felddeklaration.

VM

```
#macro (fielddecl $field)
$!field.access $!field.static $!field.final $field.type $field.name
 #if ($field.initializer)
 = $field.initializer
 #end
#end
```

Sollte nun irgendwo in einem Template der Bedarf nach einer Felddeklaration bestehen, so kann der Ersteller des Templates über die *Macro-Library* von Velocity problemlos auf dieses Macro zugreifen. In folgendem Beispiel werden die Deklarationen der Felder einer Klasse der Reihe nach ausgegeben.

VM

```
#foreach ( $foofield in $foo.fields )
 #fielddecl($foofield) ;
#end
```

6.2 Umsetzung der UML/P

In diesem Abschnitt werden einige Möglichkeiten betrachtet, wie die Elemente der UML/P in Java umgesetzt werden können. Wie zu Beginn des Kapitels erwähnt, werden hier lediglich Vorschläge gemacht. Wie die Modelle des AMTools tatsächlich umgesetzt werden, hängt davon ab, wie der zu generierende Code eingesetzt werden soll, und kann mit den Velocity-Skripten bestimmt werden. Eine ausführliche Behandlung der Umsetzungsmöglichkeiten der UML/P findet sich in [Rum02].

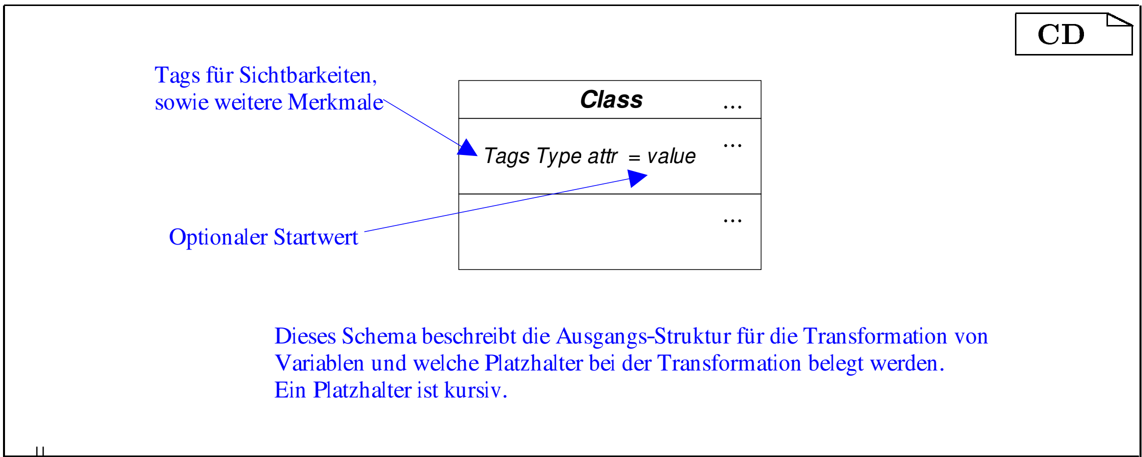
Ausgangspunkt ist die Betrachtungsweise der im Modell enthaltenen Informationen. Diese können entweder als *konstruktiv* oder als *deskriptiv* angesehen werden. Wird eine Information als konstruktiv betrachtet, so kann sie vollständig in dem generierten Java-Code wiedergespiegelt werden. Betrachtet man eine Information als deskriptiv, so kann man sie nicht komplett in Java-Code umwandeln. In diesen Fällen versucht man meistens, mittels Tests (siehe Abschnitt 3.3.1) für den generierten Code zu der Aussage zu gelangen, dass die Information mit einer gewissen Wahrscheinlichkeit umgesetzt ist. Das letztendliche Ziel des AMTools ist eine konstruktive Umsetzung der in einem UML/P-Modell enthaltenen Informationen. Allerdings ist oftmals der Aufwand für eine konstruktive Umsetzung so hoch, dass sie in Konflikt mit der Entwickelbarkeit und Anwendbarkeit des AMTools gerät. Wie so oft liegt es bei Grenzfällen im Ermessen des Enwicklers, wann eine konstruktive Umsetzung sinnvoll ist, und wann der Aufwand dem damit gewonnenen Nutzen nicht mehr entspricht.

6.2.1 Übersetzung von Klassendiagrammen

Klassendiagramme, die eine Basis der UML/P sind, können zum größten Teil konstruktiv auf Java-Code abgebildet werden. Besonders die Struktur der Software kann komplett in *Klassenrahmen* umgesetzt werden. Probleme gibt es jedoch zum Beispiel bei der Kardinalitätsangabe [1]. In Ermangelung einer effizienten konstruktiven Umsetzung wird diese Angabe als eine deskriptive Information betrachtet.

Attribute

Einfache Attribute werden im Allgemeinen nach dem in Tabelle 6.3 dargestellten Schema umgesetzt. Eine Besonderheit bei der Umsetzung ist das Voranstellen eines Unterstrichs vor den Attributnamen im generierten Code. Damit sind die so generierten Attribute gekennzeichnet. Entwickler, die an anderen Teilen einer Software arbeiten und nicht das AMTool verwenden, werden so auf die Verwendung von Getter- und Setter-Methoden hingewiesen.

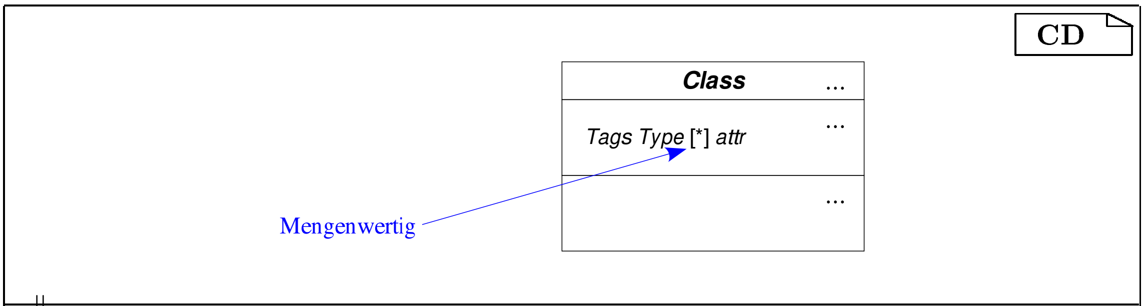


Java

```
class Class {

  private Type _attr = value;

  tags' synchronized Type getAttr () {
    return ( _attr );
  }

  tags" synchronized Type setAttr ( Type attr ) {
    return ( _attr = attr );
  }
}
```

Tabelle 6.3: Umsetzung eines Standardattributs

Einen Zusatz zu den in Java enthaltenen Attributangaben stellt die Kardinalitätsangabe in Klassendiagrammen dar. So kann zum Beispiel ein Attribut vom Typ Integer mengenwertig sein, d.h. dieses Attribut stellt eine Menge von Integer-Objekten dar. Attribute, die mit dem Zeichen [*] als Menge deklariert sind, bedürfen einer anderen Übersetzung als der in Tabelle 6.3. Wie in Tabelle 6.4 zu sehen, müssen diese mengenwertigen Attribute mittels einer *Collection* oder Ähnlichem realisiert werden. Da das Java-Attribut nun vom Typ Collection (oder ...) ist, müssen die Setter-Methoden sicherstellen, dass immer der richtige Typ in diesem Attribut gespeichert wird. Sollte das mengenwertige Attribut das Merkmal *ordered* haben, so wird statt dem *Set* ein Objekt mit dem Interface *List* verwendet.



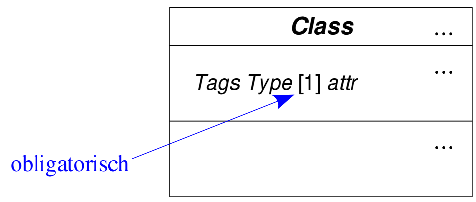
Java

```
class Class {
  private java.util.Set _attr;

  tags' synchronized java.util.Set getAttr () {
    return ( _attr );
  }

  tags" synchronized void addAttr ( Type attr ) {
    _attr.add( attr );
  }

  tags" synchronized void removeAttr ( Type attr ) {
    _attr.remove( attr );
  }
}
```

Tabelle 6.4: Umsetzung eines mengenwertigen Attributs

Ein wesentlich schwierigerer Fall ist der Umgang mit Attributen der Kardinalität [1]. Für eine konstruktive Umsetzung dieser Angabe müsste man für alle möglichen Abläufe sicherstellen, dass in keinem dieses Attribut mit *null* belegt wird. Wenn man, wie oben entschieden, die Information als deskriptiv betrachtet, kann man eine Umsetzung wie in Tabelle 6.5 wählen. Bei dieser Umsetzung wird zwar nicht sichergestellt, dass in dem generierten Programm der Wert null nicht auftritt, aber sollte die Setter-Methode bei Tests den Wert null erhalten, so schlägt zumindest der Test fehl. Die Überprüfung der Getter-Methode soll sicherstellen, dass auch die Anfangsbelegung des Attributs ungleich null ist.

CD



⇓

Java

```
class Class {
  private Type _attr = value;

   tags' synchronized Type get Attr () {
    assert ( attr != null);
    return ( _attr );
   }

   tags" synchronized Type set Attr ( Type attr ) {
    assert ( attr != null);
    return ( _attr = attr );
   }
}
```

Tabelle 6.5: Umsetzung eines Attributs mit Kardinalität [1]

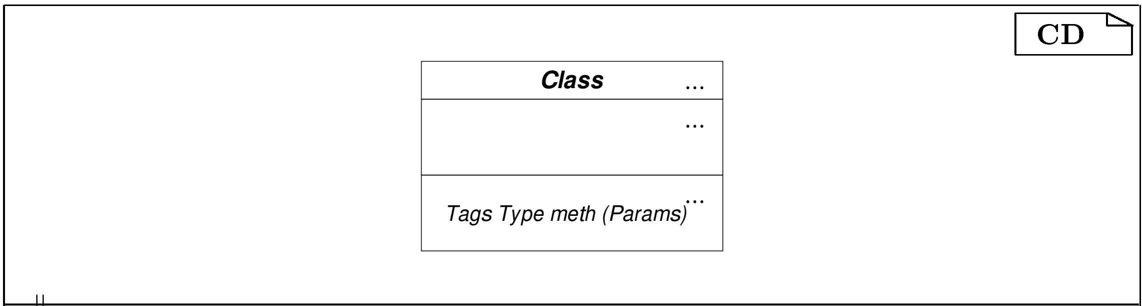
Methoden

In Klassendiagrammen wird von einer Methode nur die Signatur angegeben. So kann aus dem Diagramm selbst keine vollständige Methode generiert werden. Die Information zu dem Block, der die Funktionalität der Methode beschreibt, muss entweder aus Coderümpfen oder der OCL ermittelt werden. Deshalb wird bei Methoden nur ein Rahmen aufgebaut, der noch mit einem Block gefüllt werden muss. Der Aufbau eines Methodenrahmens ist in Tabelle 6.6 zu sehen.

Methoden, die ausschließlich zum Zugriff auf Attribute dienen, also Getter- und Setter-Methoden, sollten nicht eigens im Klassendiagramm aufgeführt werden. Sie werden automatisch mit den Attributen generiert und können in den Coderümpfen referenziert werden. Möchte man Zugriffsmethoden mit anderen Namen und einer veränderten Funktionalität definieren, so kann dies mit Methodendeklarationen geschehen. Oft ist es jedoch besser, für solche Methoden das jeweilige Attribut mit einem Merkmal oder Stereotypen zu versehen und dann über ein verändertes Skript des Generators die Methode direkt bei den Attributen zu erstellen.

Assoziationen

Assoziationen können in zwei Gruppen gespalten werden. Es gibt die unidirektionalen und die bidirektionalen Assoziationen. Unidirektionale Assoziationen können



Java

```
class Class {

  tags Type meth ( Params ) {
  ... body ...
  }
}
```

Tabelle 6.6: Umsetzung einer Methode

dabei wie Attribute einer Klasse behandelt werden. Der Typ des Attributes entspricht der Klasse am anderen Ende der Assoziation. Bei der Namensgebung des Attributes wird vorzugsweise der Rollenname innerhalb der Assoziation verwendet. Sollte dieser nicht vorhanden sein, wird alternativ der Name der Assoziation benutzt. Im Falle, dass beide Namen nicht zur Verfügung stehen, kann der Typ der gegenüberliegenden Klasse als Name verwendet werden.

Bidirektionale Assoziationen fordern beim Hinzufügen und Entfernen eines Links, dass beide Objekte an dessen Enden geändert werden. Mit dem Einbau eines Methodenaufrufs in dem jeweiligen Attribut, wie in Tabelle 6.7 zu sehen, kann zum Beispiel sichergestellt werden, dass die Attribute, die zu einer Assoziation gehören, immer konsistet sind. Mittels des Unterstrichs soll sichergestellt werden, dass die lokalen Methoden für einen Assoziationszugriff nur von den zugehörigen Assoziationsmethoden genutzt werden.

Die Umsetzung der Kardinalitäten verläuft analog zu den Kardinalitäten bei Attributen. Auch hier werden mittels Assert-Anweisungen lediglich Tests unterstützt.

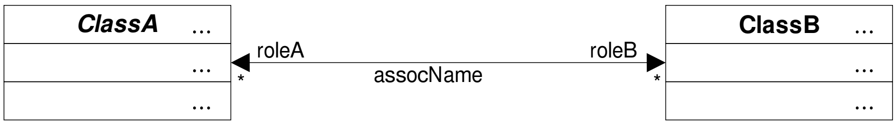
6.2.2 OCL

OCL-Contraints werden in den meisten Fällen nur als deskiptive Information angesehen, da oftmals, um zum Beispiel eine Invariante sicherzustellen, der gesamte generierte Code analysiert werden müsste. Hier werden ansatzweise zwei Vorschläge gegeben, wie verschiedene OCL-Constraints umgesetzt werden können.

Invarianten

Invarianten ist eine Eigenschaft, die in der gesamten Laufzeit des Systems gültig sein muss. Ein OCL-Constraint, welches eine Invariante beschreibt, kann bei diesem

CD



Java

```
class ClassA {
  private java.util.Set _roleB;

  public synchronized java.util.Set getRoleB () {
    return ( _roleB );
  }
  public synchronized void addRoleB ( ClassB roleB ) {
    _roleB.add( roleB );
    roleB._addLocalRoleA ( this );
  }
  public synchronized void removeRoleB ( ClassB roleB ) {
    _roleB.remove( roleB );
    roleB._removeLocalRoleA ( this );
  }

  public synchronized void _addLocalRoleB ( ClassB roleB ) {
    _roleB.add( roleB );
  }
  public synchronized void _removeLocalRoleB ( ClassB roleB ) {
    _roleB.remove( roleB );
  }
}
```

Tabelle 6.7: Umsetzung einer Assoziation

Prototypen in zwei Teile aufgespalten werden. Eine Kontextdeklaration bestimmt, welche Klassen referenziert werden können, und ein Java-Ausdruck beschreibt die Eigenschaft, die gelten soll. Der Vorschlag für eine Umsetzung von OCL soll hier anhand des in Kapitel 2 gegebenen Constraints geschehen.

OCL

```
context XY xy inv:
  !xy.a.equals(5);
```

Die Invariante bezieht sich immer auf den Zustand des Systems, also auf die Attribute von Klassen. Sie müsste also theoretisch bei jeder Änderung des Systemzustands geprüft werden. Um diesen Aufwand einzuschränken, werden lediglich die Attribute der in dem Kontext gegebenen Klasse kontrolliert. So wird in diesem Beispiel nur die Klasse XY betrachtet. Eine Zustandsänderung kann hier nur mit Setter- und Getter-Methoden herbeigeführt werden. Mit entspechenden Assert-Aufrufen in diesen Methoden kann so in Testläufen sichergestellt werden, dass die Invariante wahr ist. Die Umsetzung des oben genannten Constraints in einer Klasse XY könnte also folgendermaßen aussehen.

Java

```
class XY {

  private int _a;

  public int getA () {
    assert ( !this._a.equals(5) );
    return _a;
  }

  public int setA ( int a ) {
    assert ( !this._a.equals(5) );
    return _a = a;
  }
}
```

Der angegebenen Ausdrücke müssen in drei Punkten abgeändert werden, um sie in eine Java-Klasse einbetten zu können. Da durch die Angabe des Kontextes die Prüfung der Invariante, die mit diesem Ausdruck gegeben ist, bereits in die Klasse XY integriert ist, muss diese Klasse nicht mehr mit einem Namen referenziert werden. Die vorkommenden Referenzen auf „xy." können jedoch, statt gelöscht zu werden, auch mit dem aussagekräftigeren Ausdruck „this." ersetzt werden. Der zweite Punkt, der geändert werden muss, ist der Unterstrich vor dem Attributnamen. Bei Ausdrücken in anderen Methoden wird der Unterstrich durch die Verwendung von Getter- und Setter-Methoden transparent gehalten, aber in diesen Methoden selbst muss direkt referenziert werden, da sonst rekursive Endlosschleifen auftreten würden. Damit ist auch der dritte Punkt, das Herausnehmen aller Zugriffsmethoden für diese Klasse, erklärt.

Vor- und Nachbedingungen

Mit Vor- und Nachbedingungen wird die Funktionalität von Methoden beschrieben. Dementsprechend wird als Kontext eine Methode angegeben. Im Vergleich zu den Invarianten, deren Assert-Anweisungen in den Zugriffsmethoden abgelegt wurden, werden hier diese Anweisungen in den Methodenrumpf gelegt. Wie im vorigen Abschnitt soll eine Umsetzung auch hier mit einem Beispiel aus Kapitel 2 verdeutlicht werden.

OCL

```
context float MyNumber.squareRoot(float n1)
  pre: n1 > 0
  post: (result * result) == n1
```

Im Gegensatz zu den Invarianten müssen die Java-Ausdrücke in den Bedingungen nicht modifiziert werden, da nur Namen auftreten können, die in der Methode bekannt sind. Die Vor- und die Nachbedingung können also direkt eingesetzt werden.

Java

```
class MyNumber {

  float squareRoot ( float n1 ) {

    boolean oclPre1 = ( n1 > 0 );

    ... body /* ohne return */ ...

    float result = /* Return-Wert der Methode */
    if (oclPre1) assert ( ( result * result ) == n1 );
    return result;
  }
}
```

Der Aufwand, die Vorbedingung im Code einzubetten, ist gering, da sie immer an vorderste Stelle in den Methoden-Block geschrieben werden kann. Um die Nachbedingung einzubetten, muss jedoch der Block modifiziert werden. Bei jedem auftretenden „return“ muss zuvor geprüft werden, ob alle Bedingungen eingehalten werden. Dies erfolgt durch ein Aufspalten der Return-Anweisung, in der der Rückgabewert mit einer Hilfsvariable untersucht wird, die dann von der Methode zurückgegeben wird.

Mit dem alleinigen Einbau der Assert-Anweisungen ist jedoch der Code noch nicht getestet worden. Der Aufbau von einer Umgebung, in der die Methode mit verschiedenen Parameterwerten getestet werden kann, ist ein sehr schwieriges und aufwendiges Unterfangen. Deshalb muss in diesem Umsetzungsvorschlag ein Test von Hand geschrieben werden, der wie folgt aussehen könnte:

Java

```
...
void testSquareRoot () {
  MyNumber n = new MyNumber();
  n.squareRoot(0);
  n.squareRoot(-1);
  n.squareRoot(5);
}
...
```

6.2.3 Coderümpfe

Coderümpfe können nahezu komplett in Java übernommen werden. Es müssen in der aktuellen Version lediglich zwei Eingriffe stattfinden. Zum einen müssen die Anweisungen mit dem Schlüsselwort „ocl“ in Java-Assert-Anweisungen umgewandelt werden. Bei dem momentanen Umfang der OCL muss dazu nur das Schlüsselwort gewechselt werden, da die Ausdrücke selbst bereits in Java geschrieben sind. Der andere Eingriff in die Code-Rümpfe betrifft alle Ausdrücke. So darf kein direkter

Zugriff auf ein Attribut erfolgen, sondern es müssen immer Getter- und Setter-Methoden verwendet werden. Um dies sicherzustellen muss jeder Ausdruck „x.“, wobei x eine beliebige Variable darstellt, durch „getX().“ ersetzt werden. Bei der Referenz x selbst, sollte hinter ihr kein Punkt stehen, muss zwischen linker und rechter Seite einer Zuweisung unterschieden werden. Auf der rechten Seite kann sie weiterhin in eine Getter-Methode umgewandelt werden. Auf der linken Seite muss sie hingegen durch eine Setter-Methode ersetzt werden, die die rechte Seite als Parameter hat. Tabelle 6.8 soll diese Umwandlung nochmals verdeutlichen.

z.y.x = a.b.c;	**Java**
⇓ getZ().getY().setX(getA().getB().getC());	**Java**

Tabelle 6.8: Umwandlung von direkten Attributzugriffen

Kapitel 7

Zusammenfassung und Ausblick

Zusammenfassung

In der Arbeit wurde ein Werkzeug, das aus UML/P-Modellen Java-Code generiert, konzipiert und prototypisch umgesetzt. Die UML/P ist eine in [Rum02] beschriebene Teilmenge der UML, die speziell für die Generierung von Java modifiziert und um Coderümpfe ergänzt wurde. Um eine optimale Verträglichkeit mit anderen Werkzeugen zu gewährleisten, wurde für die graphischen Elemente der UML/P (Klassendiagramme und Objektdiagramme) eine Textnotation entworfen.

Das Werkzeug hat, wie in der Graphik in Tabelle 7.1 zu sehen, die Architektur eines Compilers, der aus mehreren Modulen besteht. Die Verwendung einer Compilerstruktur kommt der Anforderung, aus der Sprache UML/P die Sprache Java zu generieren, entgegen. Um eine flexible Anpassung des Werkzeugs an die Gegebenheiten eines agilen Enwicklungsprozesses zu ermöglichen, sind die Module stark parametrisierbar. Um die Qualität des Werkzeuges zu sichern, werden System-Tests für das System als Ganzes und JUnit-Tests für die Methoden verwendet.

Die lexikalische und die syntaktische Analyse der UML/P übernehmen ein Lexer und ein Parser. Zur Herstellung dieser Module werden die Hilfswerkzeuge JFlex und Cup herangezogen. Die Auswertung des Syntaxbaumes erfolgt mit Hilfe von Modulen, die eine Symboltabelle erstellen (Treewalker), die Konsistenz der Modelle prüfen und bei Bedarf fehlende Informationen ergänzen (Lazy-Evaluation Module). Ein mit Velocity aus Templates erstellter Generator erzeugt schließlich den Java-Code.

Ausblick

In einer Implementierung wurde die Spezifikation teilweise umgesetzt, um sie zu prüfen und zu bewerten (Proof of Concepts). Die Prüfung wurde anhand der Klassendiagramme durchgeführt. Tabelle 7.2 zeigt den aktuellen Entwicklungstand des Werkzeugs. Eine Prüfung der Einsatztauglichkeit des Prototypen ist im jetzigen Zustand nicht möglich, da das Werkzeug Coderümpfe noch nicht umsetzen, also kein lauffähiges Programm erzeugen kann.

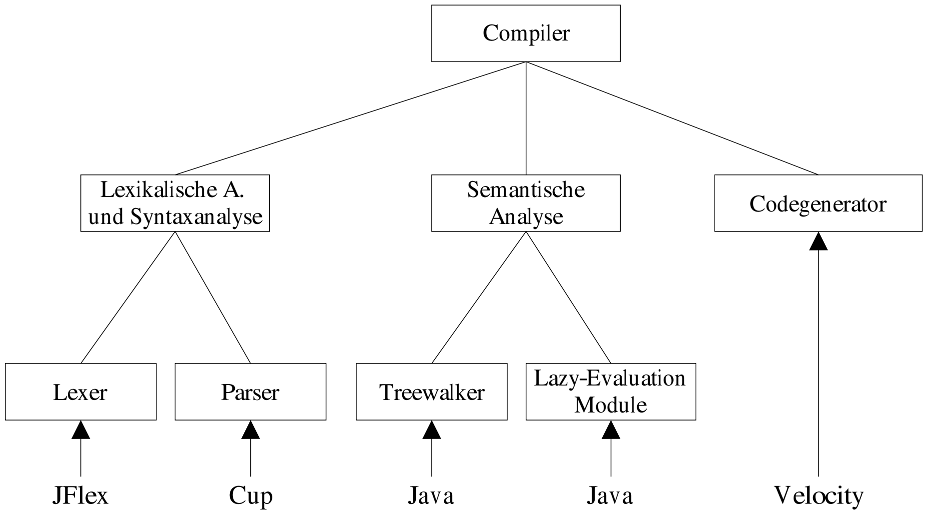


Tabelle 7.1: Der Aufbau der Werkzeugs

Der Einbau der Coderümpfe in die Implementierung und die dann mögliche Prüfung der Einsatztauglichkeit wird in einem Systementwicklungsprojekt am Lehrstuhl Professor Broy durchgeführt werden. Aufbauend auf dieser Diplomarbeit können zudem der komplette Umfang der OCL/P aus [Rum02] und andere Elemente der UML, wie zum Beispiel State-Charts, auf ihre Umsetzbarkeit in Java-Code untersucht werden.

Nächste Entwicklungsschritte des Prototypen wären die Implementierung der OCL/P und der Objektdiagramme. Des weiteren könnte man eine graphische Eingabemöglichkeit für die Modelle erstellen. Eine interessante Aufgabe wäre auch, den Prototypen selbst aus der UML/P zu erstellen. So könnte sich das Werkzeug teilweise selbst generieren, bzw. mit dem Bootstrapping-Verfahren ([BH98] S.175) weiterentwickelt werden.

	Klassendiagr.	*Coderümpfe*	*OCL/P*	*Objektdiagr.*
JFlex Ausdrücke	90%	90%		
Cup-Grammatik	90%	90%		
Abstrakter Syntaxbaum	90%	90%		
Symboltabelle	90%			
Auswertung des Baumes	90%			
Generierung	90%			

Tabelle 7.2: Entwicklungsstand des Prototypen

Anhang A

Technische Referenz

A.1 Der abstrakte Syntaxbaum

In Kapitel 4 wurde beschrieben, wie mittels einer Grammatik aus einer UMP/P-Datei ein abstrakter Syntaxbaum entsteht. Dort wurde jedoch noch nichts über die konkreten Java-Befehle geschrieben, die bei einer Regelausführung mit Cup den abstrakten Syntaxbaum im Speicher aufbauen. Dieser Abschnitt soll nun zeigen, wie dieser Aufbau im AMTool realisiert ist. Dazu werden zuerst die Knoten des Baumes und die Beziehungen zwischen ihnen betrachtet und dann die Erstellung der Knoten selbst.

A.1.1 Knoten des abstrakten Syntaxbaumes

Ein Knoten im abstrakten Syntaxbaum wird durch ein Objekt im Speicher repräsentiert. Ein neuer Knoten entsteht nur bei einer Regelanwendung und so wird meistens für die Nichtterminale auf der linken Seite der Regeln eine eigene Klasse eingerichtet.

EBNF

$\langle\text{CatchClause}\rangle \quad ::= \underline{\texttt{catch}}\ \underline{\texttt{(}}\ \langle\text{Type}\rangle\ \langle\text{Identifier}\rangle\ \underline{\texttt{[]}}^{*}\ \underline{\texttt{)}}\ \langle\text{Block}\rangle\underline{\texttt{;}}$

Die Namen der Klassen, die einen Knoten beschreiben, beginnen einheitlich mit einem „T“, an dem dann der kleingeschriebene Name des jeweiligen Nichtterminals angehängt wird. So heißt zum Beispiel die Klasse für den Knoten, der mit der oben gegebenen Regel entsteht, „Tcatchclause“. Der Knoten hat drei feste Kinder (Type, Identifier und Block) und ein Terminal, dass beliebig oft erscheinen kann. Die Kinder werden als Attribute in den Klassen der Knoten abgelegt. In den Knoten des Baumes können ebenfalls Attribute abgelegt werden, die bei der Auswertung als Zwischenspeicher oder Cache dienen.

Impl.

```
class Tcatchclause extends AST {
   public Ttype type = null;
   public Tidentifier identifier = null;
   public Tblock block = null;

   public int arrayDepth = 0;
   ...
}
```

Jede Klasse, die einen Knoten beschreibt, erweitert die Klasse AST, die Attribute enthält,die in allen Knoten gebraucht werden können. Eine besondere Behandlung hat das Terminal „[]“ erhalten. Hier muss lediglich gespeichert werden, wie oft es bei der Regelanwendung vorkommt, da es selbst keine weiteren Informationen enthält. Enthalten wiederholte Terminale oder Nichtterminale jedoch Informationen, so werden sie in einem typisierten Vector abgelegt. Die Typisierung geschieht momentan deskriptiv, da mit der Setter-Methode der Vector als ganzes geliefert wird.

EBNF

$\langle$Block$\rangle$ ::= $\underline{\{}$ $\langle$Statement$\rangle^*$ $\underline{\}}$

Impl.

```
class Tblock extends AST {
   public Vector statements = null;

   ... setStatements ( Vector statements ) {
    ...
   }
   ...
}
```

Auffallend ist hier, dass die Attribute alle *public* sind, obwohl es Setter- und Getter-Methoden gibt. Der Grund ist, dass zu Testzwecken diese Attribute über das Paket *java.lang.reflect* erkannt werden müssen. Generell sollten aber weiterhin die Zugriffsmethoden benutzt werden. Auf sie wird im nächsten Abschnitt genauer eingegangen.

Die Behandlung der Knoten, die aus Regeln mit mehreren Alternativen bestehen, geschieht je nach Bedarf auf zwei verschiedene Weisen. So kann entweder der Mechanismus der Vererbung genutzt werden, oder es können die Alternativen als Kinder abgelegt werden. Der erste Weg macht nur Sinn wenn in den Regeln keine weiteren Terminale oder Nichtterminale zusammengefasst werden. Dies ist zum Beispiel bei der Regel für die CDTypeDeclaration der Fall:

EBNF

$\langle$CDTypeDeclaration$\rangle$
::= $\langle$CDClassDeclaration$\rangle$ | $\langle$AssocciationDeclaration$\rangle$ |
$\langle$CDInterfacedeclaration$\rangle$ | $\langle$CDSubclassingDeclaration$\rangle$

Impl.

```
class Tcdtypedeclaration extends AST {}
class TCDClassDeclaration extends Tcdtypedeclaration {
  ...
}
class TAssociationDeclaration extends Tcdtypedeclaration {
  ...
}
...
```

Mit dieser Vorgehensweise können die einzelnen Elemente in einem typisierten Vector aufgenommen werden, da sie alle den gemeinsamen Typ der Oberklasse haben. Sie macht den Aufbau des abstrakten Syntaxbaumes relativ einfach, erschwert aber durch den nicht explizit gegebenen Typen eines Knotens die Auswertung. Die zweite Möglichkeit, die angewandt werden sollte, sobald in den Regeln etwas zusammengefasst wird, sieht verschiedene Attribute für die jeweiligen Alternativen vor.

EBNF

$$\begin{array}{lll}\langle\text{VariableInitializer}\rangle ::= & \langle\text{Expression}\rangle \\ \mid & \underline{\{}\ \langle\text{VariableInitializer}\rangle^{*}_{\underline{,}}\ \underline{,}^{opt}\ \underline{\}}\end{array}$$

Impl.

```
class Tvariableinitializer extends AST {
  public Texpression expression = null;
  public Vector variableInitializers = null;
  ...
}
```

Hier wird entweder das Attribut *expression* oder das Attribut *variableinitializers* belegt, je nach dem, welche Regel angewandt wurde. Tabelle A.1 soll die Unterschiede nochmals verdeutlichen.

A.1.2 Herstellen der Knoten

Dieser Abschnitt beschreibt, wie der abstrakte Syntaxbaum aus den Cup-Regeln heraus erstellt wird. Die Grammatik liegt dem AMTool in Form von Cup-Regeln, die LR(1)-konform sind, vor (siehe Kapitel B). Bei einer Regel von Cup kann unter jeder Alternative Java-Code eingebettet werden. Dieser hat die Aufgabe, das Objekt, welches das Nichtterminal auf der linken Seite der Regel repräsentiert, in einem Platzhalter namens RESULT abzulegen. Die Aufgabe wird an dem Beispiel des Nichtteminals „type“ verdeutlicht.

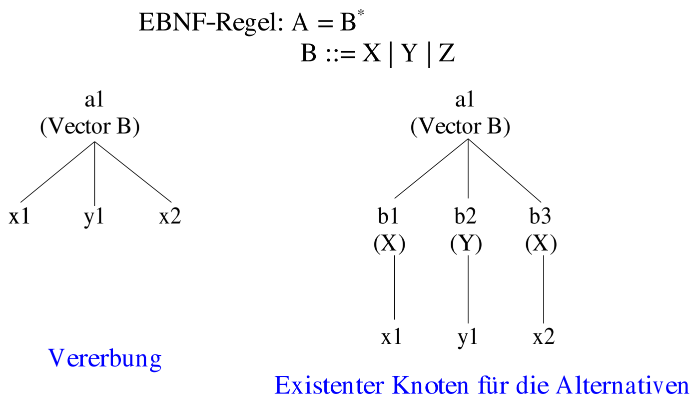


Tabelle A.1: Verschiedene Möglichkeiten zur Umsetzung einer „Oder“-Regel

Cup

```
type ::= name:n
            {: RESULT = getF().newTtype(n); :}
        | basetype:t
            {: RESULT = t; :}
        | type:t ARRAYDEPTH
            {: RESULT = t.addArrayDepth(); :}
        ;
```

Zur Berechnung des Objekts für den Platzhalter kann auf die Objekte der Terminale und Nichtterminale auf der rechten Seite der Regel zugegriffen werden, indem diese jeweils mit einem Label benannt werden, das mit einem Doppelpunkt markiert ist. Es gibt drei verschiedene Konzepte, wie mit dem eingebetteten Code der Platzhalter gefüllt werden kann. Diese werden im Weiteren anhand der „type“-Regel vorgestellt.

Bei der Alternative „basetype:t“ muss für das Nichterminal „type“ kein neues Objekt erstellt werden. Ein Basistyp ist bereits ein Typ und kann so durch ein Objekt der gleichen Klasse wie „Type“ repräsentiert werden. Des weiteren finden keine Änderungen an dem Objekt statt. Es kann also einfach durchgereicht werden, indem es dem Platzhalter zugewiesen wird.

Auch bei der Zuweisung „type:t ARRAYDETPH“ muss kein neues Objekt erzeugt werden, da das existierende Objekt lediglich mit Information angereichert wird. Um diese Anreicherung möglichst einfach zu gestalten, wurden die Zugriffsmethoden der Knoten-Objekte leicht verändert. Die schreibenden Methoden geben als Rückgabewert nicht das geschriebene Attribut wieder, sondern das Knoten-Objekt selbst. So wird, wie oben zu sehen, mit dem Aufruf der Zugriffsfunktion das Knoten-Objekt gleich weitergereicht. Die Zugriffsfunktionen haben je nach Verwendung entweder das Präfix „add“, wenn etwas zu einem Attribut hinzugefügt wird, oder „set“, wenn ein Attribut neu gesetzt wird.

Impl.

```
public class Ttype extends AST {
  public int arrayDepth = 0;
  ...
  public Ttype addArrayDepth() {
    arrayDepth++;
    return this;
  }
  ...
}
```

Die dritte Variante des eingebetteten Codes beinhaltet nun die Erstellung eines neuen Objekts. So bekommt die oben gezeigte Regel mit dem Nichtterminal „name“ ein Objekt der Klasse *Tname*. Die Aufgabe, neue Objekte zu erstellen, wird bei dem AMTool in eine *Factory* (ASTFactory.java) ausgelagert. Der Vorteil eines solchen Vorgehens ist, das die Funktionalität der Neuerstellung einer Klasse kompakt überschaubar und nicht in allen Knoten verteilt ist. Zusätzlich kann so die Funktionalität der Erstellung der Knoten durch einen Austausch der Factory einfach geändert werden. Die Methode „getF()“ liefert die Factory, von der dann die Methode „newTtype(Tname name)“ aufgerufen wird. Im Falle dieser Regel wird zum Beispiel das übergenebe Objekt nicht als Attribut direkt angehängt, sondern die Attribute des Objekts werden übernommen. Dies hat hier den Vorteil, dass auch Basistypen ohne Mehraufwand wie komplexe Typen behandelt werden können.

Impl.

```
class ASTFactory {
  ...
  /* aus type ::= name */
  public Ttype newTtype(Tname name) {
    type = new Ttype();
    type.identifier = name.identifier;
    type.path = name.path;
    return type;
  }
  /* aus basetype ::= DOUBLE | FLOAT | ... */
  public Ttype newTtype(String identifier) {
    type = new Type();
    type.identifier = identifier;
    type.path = new Vector();
    return type;
  }
}
```

Eine Besonderheit liegt bei den Regeln mit der Endung „Star“ oder „List“ vor. Der Wiederholungsoperator der EBNF wird in einer LR(1)-Grammatik mittels eines rekursiven Regelaufrufs realisiert (siehe Abschnitt 4.6). Der Aufbau einer Liste geht also in einzelenen Schritten, wobei bei jeder Regelanwendung ein Objekt angehängt wird. Die fertige Liste wird als Vector an den jeweiligen Knoten angehängt. Um nun im eingebetteten Java-Code weiterhin eine „add"-Methode verwenden zu können, die die Liste selbst weitergibt, wird ein Objekt *Tobjectlist* zum Listenaufbau verwendet. Dieses wird, nachdem die Liste dem jeweiligen Knoten als Vector übergeben wurde, wieder verworfen. Eine „Star"-Regel sieht dann folgendermaßen aus:

Cup

```
modifierStar ::= /*empty*/
                    {: RESULT = getF().newTobjectlist(); :}
               | modifier:m modifierStar:l
                    {: RESULT = l.addObject(m); :}
               ;
```

A.2 Das Test-Framework

Um die Qualitätssicherung des AMTools mit Tests zu vereinfachen, existiert ein Testframework, mit dem sowohl JUnit-Tests als System-Tests einfach erstellt und ausgeführt werden können. Bei den Systemtests wird das AMTool mit einer Mustereingabe aufgerufen und dann der generierte Code mit einer Musterlösung verglichen. Die JUnit-Tests hingegen rufen direkt die jeweils zu testende Methode auf und prüfen deren Funktionalität (siehe Kapitel 3).

A.2.1 Ausführen der Tests

Die Klassen der Testumgebung für das AMTool sind in dem Paket *test* abgelegt. Hinzugezogen wird bei den Tests das Paket *JUnit*, welches sich in einem gleichnamigen JAR-File befindet und das Paket *vdiff*. Der Testumgebung muss mittels einer Konfigurationsdatei mitgeteilt werden, was sie testen soll. In dieser Datei (test/config.txt) werden drei Parameter abgelegt, die wie folgt aussehen können.

```
patternDirName=/c:/eigene~1/progra~1/amt/test/patterns
tempDirName=/c:/eigene~1/progra~1/amt/test/temp
testDirName=/c:/eigene~1/progra~1/amt/amtool
```

Der Parameter *patternDirName* gibt an, in welchem Verzeichnis sich die Mustereingaben für die System-Tests befinden. Da sich bereits bei der Änderung eines Parameters die Ausgabe des AMTools ändern kann, können so verschiedene Mustereingaben und Ausgaben in beliebigen Verzeichnissen abgelegt werden. Die System-Tests vergleichen ihre Musterlösung mit der Ausgabe des AMTools. Dazu muss diese Ausgabe erst in einem temporären Verzeichnis in Form von Java-Dateien gespeichert werden. Der Parameter *tempDirName* gibt an, wo sich dieses Verzeichniss befindet, bzw. wo es angelegt werden soll. Der Parameter *testDirName* betrifft die JUnit-Tests. Die Testumgebung überprüft bei einem Aufruf alle Klassen, die sich in dem hier angegebenen Verzeichnis befinden. Es werden automatisch alle Klassen mit der Endung

„Test“ an JUnit weitergegeben. Momentan werden Unterverzeichnisse noch nicht berücksichtigt.

Ist die Konfigurationsdatei eingerichtet, so können die Tests wahlweise mit einem der folgenden Befehlen gestartet werden.

```
make dotest
java test.Test
```

Zuerst werden die JUnit-Tests durchgeführt. Hierbei erscheint auf der Ausgabe für jeden bestandenen Test ein Punkt. Schlägt ein Test fehl, so wird entweder ein „F“ (Failed) oder ein „E“ (Error) ausgegeben und es folgt eine Beschreibung des Fehlers. In einem zweiten Schritt werden dann die System-Tests durchgeführt. Sie rufen die Klasse *amtool.Main* mit den jeweiligen Mustereingaben auf. Schlägt hier ein Test fehl, so wird die Prozedur abgebrochen und es kann in dem temporären Verzeichnis die falsche Ausgabe betrachtet werden.

A.2.2 Erstellen von System-Tests

Um Systemtests zu erstellen, benötigt man zuerst ein Verzeichnis, in dem die Mustereingaben abgelegt werden. Dieses kann entweder bereits existieren, oder es kann für eine neue Testreihe angelegt werden. In diesem Verzeichnis wird dann eine Mustereingabe in Form von einer Datei mit der Endung „.am“ (z.B. „Muster.am“) abgelegt. Es muss dann ein gleichnamiges Unterverzeichnis angelegt werden, das die Endung „.out“ hat (z.B. Muster.out/) und in dem die gewünschte Ausgabe gespeichert wird. Der Test an sich ist somit fertiggestellt, denn die „.am“-Datei wird automatisch erkannt und in die Testfolge eingebunden.

A.2.3 Erstellen von JUnit-Tests

Um für eine Klasse (z.B. Muster.java) JUnit-Tests zu entwickeln, muss zuerst eine Klasse mit der Endung „Test“ erzeugt werden, die in dem gleichen Verzeichnis wie die zu testende Klasse liegt(z.B. MusterTest.java). Die Testklasse muss die Klasse *junit.framework.TestCase* erweitern. Wie die Tests in der Klasse aufgebaut werden, wird anhand des folgenden Beispiels beschrieben.

Impl.

```
class Muster {
  public Object musterMethode ( String x ) {
    ...
  }
}

class MusterTest extends junit.framework.TestCase {

  Muster muster = null;
  Object musterEingabe = null;
  Object musterLoesung = null;
  ...

  protected void setUp () {
    muster = new Muster();
    musterEingabe = ...;
    musterLoesung = ...;
    ...
  }

  public void testMusterMethode () {
    Object testResult = muster.musterMethode(musterEingabe);
    assertTrue(testResult.equals(musterLoesung));
  }
  ...
}
```

JUnit geht nach einem festen Schema vor. Zuerst werden alle Methoden gesucht, die mit dem Wort „test“ beginnen. Danach wird für jede dieser Methoden eine Umgebung für die zu testende Methode aufgebaut, indem die Methode *setUp* aufgerufen wird. Dann wird die „test“-Methode gestartet, die den Test durchführt. Man muss also einerseits in der Methode *setUp* die Umgebung aufbauen und in der jeweiligen Testmethode den Test schreiben. Mehr ist von dem Framework nicht vorgegeben.

Unterstützen kann man diese Tests mit dem Paket *vdiff* von Konstantin Kukuschkin [Kuk01]. Der einfache Aufruf einer equals-Methode reicht nicht aus, um ganze Objektstrukturen zu vergleichen. Mit der Methode vdiff.VCompare.compare können derartige Vergleiche gemacht werden(siehe Abschnitt 3.3.1). Diese Methode liefert einen Vector mit den Unterschieden in Form von Strings.

Impl.

```
...
public void testMusterMethode () {
  vdiff.VCompare vcompare = new vdiff.VCompare();
    Object testResult = muster.musterMethode(musterEingabe);
    assertTrue(vcompare.compare(testResult,musterLoesung).size() == 0);
}
...
```

A.3 Aufruf und Struktur des Werkzeugs

Das AMTool wird mittels eines Kommandozeilenaufrufs mit drei Parametern gestartet, der wie folgt aussieht:

`java amtool.Main` *source destination konfiguration*

Der Parameter *source* bestimmt die Datei, in der die UML/P-Modelle abgelegt sind. Der aus diesen Modellen generierte Java-Code wird dann in dem Verzeichnis *destination* abgelegt. Das AMTool muss seinen Gererierungsprozess an das jeweilige Programm anpassen können. Wie in Abschnitt 3.2.2 beschrieben, geschieht die Auswahl existierender Funktionalität mit der Angabe einer Konfigurationsklasse, die das AMTool initialisiert. So wird mit dem dritten Parameter der Name der Konfigurationsklasse angegeben. Ein konkreter Aufruf könnte also folgendermaßen aussehen:

```
java amtool.Main MyModel.am generated/ amtool.JavaInitializer
```

Jede Konfigurationsklasse muss das Interface *amtool.Initializer* implementieren. Im Weiteren wird die Konfigurationsklasse näher erläutert und die Struktur des AMTools erklärt, falls neue Module bei der Parametrisierung erstellt werden sollen.

A.3.1 Die Konfigurationsklasse

Die Beschreibung der Konfigurationsklassen lässt sich anhand des Interfaces *Initializer* gut durchführen. So sollen in diesem Abschnitt die folgenden vier in dem Interface deklarierten Methoden näher beleuchtet werden.

Impl.

```
interface Initializer {

  public UMLPParser getParser();
  public TreeWalker[] getWalkers();
  public Generator getCodeGen();

  public void initPrinter();
}
```

Zuerst wird der Parameter initPrinter() betrachtet. Die Printer-Klasse dient zur Ausgabe von Fehlermeldungen, Warnungen und Debug-Informationen. Hier kann man entscheiden, in welcher Sprache die Ausgabe stattfindet. Dazu muss man lediglich einmal die jeweilige Printer-Klasse erzeugen. Im jetzigen Zustand des AMTools sind jedoch nur die englischen Ausgaben gegeben. Dementsprechend sieht in der Konfigurationsklasse *JavaInitializer* diese Methode folgendermaßen aus:

Impl.

```
class JavaInitializer implements Initializer {
  ...
  public void initPrinter() {
    new PrinterEnglish();
  }
  ...
}
```

Die anderen Parameter bestimmen den eigentlichen Codegenrierungsvorgang. Das AMTool holt sich mit der Methode getParser() den Parser, der aus dem Zeichenstrom den abstrakten Syntaxbaum aufbaut. Das Interface *UMLPParser* schreibt dabei nur eine Methode parse(java.io.Reader) vor, mit der der Aufbau realisiert ist. Es findet hier keine Trennung von Lexer und Parser statt, da manche Parser, wie zum Beispiel ANTLR[1], die lexikalische Analyse und die Syntaxanalyse in einer Klasse zusammengelegt haben. Im Falle des AMTools werden Cup und JFlex in der Klasse *CupParser* zusammengefügt. Die Auswahl des Moduls zum parsen umfasst also in der Konfigurationsklasse *JavaInitializer* lediglich das erzeugen eines Objekts der Klasse *CupParser*.

Impl.

```
class JavaInitializer implements Initializer {
  ...
  public UMLPParser getParser() {
    CupParser cp = new CupParser();
    return cp;
  }
  ...
}
```

Ist der eingehende Strom syntaktisch analysiert worden, so werden nun auf dem enstandenen Baum die Treewalker angewendet. Das AMTool holt aus der Konfigurationsklasse ein Array mit den Treewalkern, die ausgeführt werden müssen. Jeder Treewalker muss eine Methode walk(AST root) haben, mit der der Gang begonnen werden kann. Das AMTool benutzt einen Treewalker zum Symboltabellenaufbau, wobei die Bezeichner dabei nur mit einem Index auf die sie deklarierenden Knoten des AST versehen werden. In der Konfigurationsklasse wird so ein Array der Länge 1 mit dem TreeWalker *AMTTWsymtab* zurückgegeben.

[1] www.antlr.org

Impl.

```
class JavaInitializer implements Initializer {
  ...
  public TreeWalker[] getWalkers() {
    TreeWalker[] result = new TreeWalker[1];

    /* Symboltabelle setzen */
    result[0] = new AMTTWsymtab(AMTTWsymtab.JAVA);
    return result;
  }
  ...
}
```

Schließlich holt sich das AMTool aus der Konfigurationsklasse den Generator, der aus dem abstrakten Syntaxbaum Java-Code generiert. In der Methode getGenerator() wird der Generator mit den zugehörigen Lazy-Modulen zur Auswertung des Baumes aufgebaut. Über das Interface *Generator* ist vorgeschrieben, eine Methode startgenerate(AST root) zur Verfügung zu stellen. Bei dieser Implementierung des AMTools wird der Generator zusammen mit dem Modul *AMTEvalComplete* und dem Start-Template „templates/JGenTemplate.vm “ verwendet. Die entsprechende Methode in der Konfigurationsklasse sieht folgendermaßen aus:

Impl.

```
class JavaInitializer implements Initializer {
  ...
  public Generator getGenerator() {
    AmtEvalComplete ec = new AmtEvalComplete();
    Generator cg = new JCodeGen("‘templates/JGenTemplate.vm”’,ec);
    return  cg;
  }
  ...
}
```

A.3.2 Die Struktur

Um sich in dem jetzt bestehenden Code schnell zurechtzufinden, zählt dieser Anschnitt die relevanten Dateien und Verzeichnisse des AMTools auf. Die in dieser Arbeit selbst geschriebenen Programmdateien des AMTools befinden sich in dem Paket *amtool*. Bis auf die Testumgebung ist hier der gesamte Code inklusive AMScanner.lex und AMParser.cup zu finden. Die Templates befinden sich in dem Verzeichnis *templates/*. Wie einige andere Verzeichnisse, ist dieses nicht vorgeschrieben. Es kann mittels der Parameter in der Konfigurationsdatei beliebig umbenannt oder verschoben werden. Die Verzeichnisstruktur wird in Tabelle A.2 dargestellt, wobei die kursiv markierten Äste nicht fest vorgeschrieben sind.

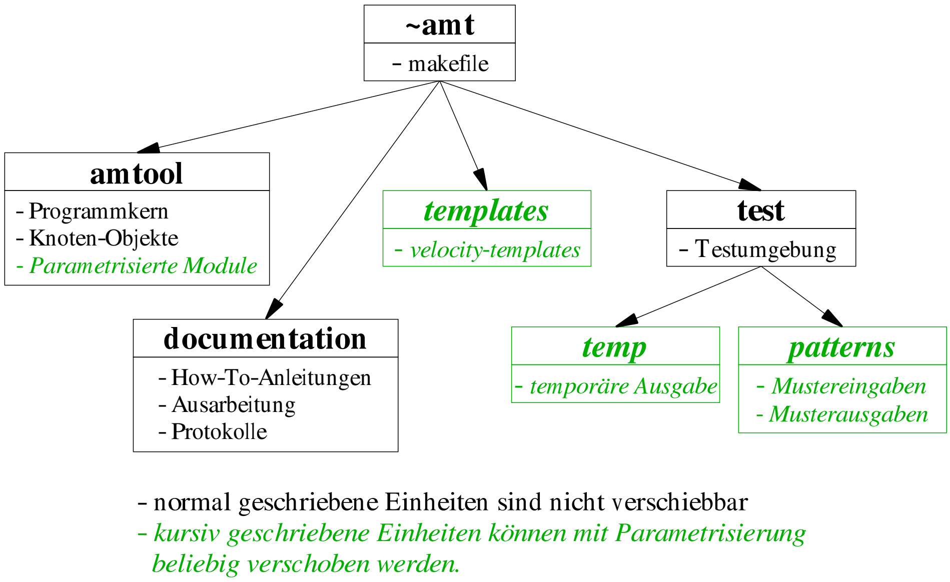


Tabelle A.2: Die Verzeichnisstruktur des AMTools

Gerade für die Einarbeitung ist das Verzeichnis *documentation/* relevant. Dort findet man neben dieser Ausarbeitung auch *How-To*-Dateien, die bestimmte Vorgänge beschreiben. So wird zum Beispiel die Installation des AMTools auf verschiedenen Plattformen erklärt.

Anhang B

LR(1)-Grammatik

Die Präzedenzen der Terminale von der Grammatik zu UML/P.

Cup

```
precedence nonassoc  ID;
precedence nonassoc  INTNUM, LIT, FNUM;
precedence right     OPENSBRACKET, CLOSESBRACKET;
precedence left      SEMI;
precedence left      ELSE;
precedence right     EQUALS, STAREQUALS, SLASHEQUALS, PERCENTEQUALS,
                     PLUSEQUALS, MINUSEQUALS, DOUBLELANGLEEQUALS,
                     DOUBLERANGLEEQUALS, ANDEQUALS, CIRCONFLEXEQUALS,
                     OREQUALS;
precedence left      QMARK, COLON;
precedence left      LANGLE, RANGLE, SMALLEREQUAL, BIGGEREQUAL,
                     DOUBLEEQUAL, NOTEQUAL;
precedence left      AND, CIRCONFLEX, OR, CONDITIONALAND, CONDITIONALOR;
precedence left      PLUS, MINUS;
precedence left      STAR, SLASH, PERCENT, DOUBLELANGLE, DOUBLERANGLE,
                     TRIPLERANGLE;
precedence left      INSTANCEOF;
precedence right     DOUBLEPLUS, DOUBLEMINUS;
precedence left      POINT, ARRAYDEPTH;
```

Umsetung von Tabelle 4.4.

Cup

```
AMFile        ::= umlpStar
                  ;
umlpStar      ::= /* empty */
                  | classdiagram umlpStar
                  | javapart umlpStar
                  ;

classdiagram  ::= CLASSDIAGRAM ident OPENBLOCK typedeclStar CLOSEBLOCK
                  ;
typedeclStar  ::= /*empty*/
                  | typedecl typedeclStar
```

```
                   ;
typedecl      ::= classdecl | interfacedecl
                  | assocdecl | subclassingdecl
                  ;
```

Umsetung von Tabelle 4.5.

Cup

```
classdecl     ::= stereotypeStar cdmodifierStar
                  CLASS name:i extendsOpt implementsOpt completenessOpt
                  OPENBLOCK classbodydeclStar CLOSEBLOCK featureStar
                  ;
classbodydeclStar
              ::= /*empty*/
                  | classbodydecl classbodydeclStar
                  ;
classbodydecl ::= FIELDS completeness SEMI | METHODS completeness SEMI
                  | attribute | methodsignature
                  ;

interfacedecl ::= stereotypeStar cdmodifierStar
                  INTERFACE name iextendsOpt completenessOpt
                  OPENBLOCK interfacebodydeclStar CLOSEBLOCK featureStar          ;
interfacebodydeclStar
              ::= /*empty*/
                  | interfacebodypart interfacebodydeclStar
                  ;
interfacebodypart
              ::= FIELDS completeness SEMI | METHODS completeness SEMI
                  | attribute | methodsignature
                  ;

implementsOpt ::= /*empty*/
                  | IMPLEMENTS namelist
                  ;
iextendsOpt   ::= /*empty*/
                  | EXTENDS namelist
                  ;
extendsOpt    ::= /*empty*/
                  | EXTENDS name
                  ;
```

Umsetung von Tabelle 4.6.

Cup

```
assocdecl     ::= assocsort identOpt EQUALS
                  type asidentOpt
                  cardinalityOpt assocdir cardinalityOpt
                  type asidentOpt featureStar SEMI
                  ;
assocsort     ::= ASSOCIATION | AGGREGATION
                ;
```

```
asidentOpt    ::= /*empty*/
                  | AS ident
                  ;
assocdir      ::= BIDIRECT | UNIDIRECT | DOUBLEMINUS
                  ;
subclassingdecl
              ::= EXTENDS identOpt EQUALS type UNIDIRECT type SEMI
                  ;
```

Umsetung von Tabelle 4.7. Hierbei ist die in Abschnitt 4.7.1 erwähnte Auslagerung in Java zu beachten.

Cup

```
attribute     ::= stereotypeStar cdmodifierStar typevoidcardStar
                  varinitOpt featureStar SEMI
                  ;
methodsignature
              ::= stereotypeStar cdmodifierStar typevoidcardStar
                  cdformparams squarebracketStar throwsOpt featureStar SEMI
                  ;
cdformparams  ::= LPAR POINT POINT POINT RPAR
                  | formparams
                  ;
```

Umsetung von Tabelle 4.8.

Cup

```
typevoidcardStar
              ::= /*empty*/
                  | typevoid  cardinalityOpt typevoidcardStar
                  ;

cdmodifierStar
              ::= /*empty*/
                  | cdmodifier cdmodifierStar
                  ;
cdmodifier    ::= modifier | READONLY | visibility
                  ;

completenessOpt
              ::= /*empty*/
                  | completeness
                  ;
completeness  ::= COMPLETE | INCOMPLETE
                  ;
cardinalityOpt::= /*empty*/
                  | OPENSBRACKET cardinality CLOSESBRACKET
                  ;
cardinality   ::= number
                  | number POINT POINT number
                  | STAR
                  ;
```

```
visibility    ::= PLUS | RHOMBUS | QMARK | MINUS
                  ;

stereotypeStar::= /*empty*/
                  | stereotype stereotypeStar
                  ;
stereotype    ::= DOUBLELANGLE ident DOUBLERANGLE
                  ;
featureStar   ::= /*empty*/
                  | feature featureStar
                  ;
feature       ::= OPENBLOCK featureentrylist CLOSEBLOCK
                  | OPENBLOCK CLOSEBLOCK
                  ;
featureentrylist
              ::= featureentry
                  | featureentry COMMA featureentrylist
                  ;
featureentry  ::= ident
                  | ident EQUALS expression
                  ;
```

Umsetung von Tabelle 4.9.

Cup

```
javapart      ::= JAVA OPENBLOCK
                  importStar constraintormgroupStar
                  CLOSEBLOCK
                  ;

importStar    ::= /*empty*/
                  | importstatement importStar
                  ;
importstatement
              ::= IMPORT name POINTSTAR SEMI
                  | IMPORT name SEMI
                  ;

constraintormgroupStar
              ::= /*empty*/
                  | methodgroupdeclaration constraintormgroupStar
                  | constraint constraintormgroupStar
                  ;
methodgroupdeclaration
              ::= CLASS name OPENBLOCK methodorconstuctStar CLOSEBLOCK
                  ;
methodorconstuctStar
              ::= /*empty*/
                  | methodrumpfdecl methodorconstuctStar
                  | constructorrumpfdecl methodorconstuctStar
                  ;
```

```
methodrumpfdecl
              ::= methodheader operationconstraintbody block
                  ;
constructorrumpfdecl
              ::= constructorheader operationconstraintbody block
                  ;
```

Umsetung von Tabelle 4.10.

Cup

```
block         ::= OPENBLOCK statementStar CLOSEBLOCK
                  ;
statementStar ::= /*empty*/
                  | statement statementStar
                  ;
statement     ::= FINAL type variabledeclaratorList SEMI
                  | type variabledeclaratorList SEMI
                  | block
                  | SEMI
                  | expression SEMI
                  | SWITCH LPAR expression RPAR
                    OPENBLOCK switchpartStar CLOSEBLOCK
                  | DO statement WHILE LPAR expression RPAR SEMI
                  | BREAK identOpt SEMI
                  | CONTINUE identOpt SEMI
                  | RETURN SEMI
                  | RETURN expression SEMI
                  | SYNCHRONIZED LPAR expression RPAR block
                  | THROW expression SEMI
                  | TRY block catchclauseStar
                  | TRY block catchclauseStar FINALLY block
                  | ident COLON statement
                  | IF LPAR expression RPAR statement
                  | IF LPAR expression RPAR statement ELSE statement
                  | WHILE LPAR expression RPAR statement
                  | FOR LPAR forinitOpt SEMI expressionOpt SEMI
                    expressionStar RPAR statement
                ;
switchpartStar::= /*empty*/
                  | switchpartStar switchpart
                  ;
switchpart    ::= case statementStar
                  ;
case          ::= CASE expression COLON
                  | DEFAULT COLON
                  ;
catchclauseStar
              ::= /*empty*/
                  | catchclause catchclauseStar
                  ;
catchclause   ::= CATCH LPAR type ident squarebracketStar RPAR block
```

```
                    ;
forinitOpt     ::= /*empty*/
                    | forinit
                    ;
forinit        ::= type variabledeclaratorStar
                    | FINAL type variabledeclaratorStar
                    | expressionList
                    ;
```

Umsetung von Tabelle 4.11. Bei der Umsetzung des Typecast-Operators ist hier die Sonderbehandlung von Abschnitt 4.7.2 zu beachten.

Cup

```
expressionOpt ::= /*empty*/
                    | expression
                    ;
expressionStar::= /*empty*/
                    | expressionList
                    ;
expressionList::= expression:e
                    | expression COMMA expressionStar
                    ;
expression     ::= prefixopStar primary postfixopStar
                    | prefixopStar LPAR basictype squarebracketStar RPAR
                      prefixopStar primary postfixopStar
                    | prefixopStar LPAR name squarebracketStar RPAR
                      prefixopNonminusplusStar primary postfixopStar
                    | prefixopStar LPAR name squarebracketStar RPAR
                      postfixopStar
                    | expression infixop expression
                    | expression INSTANCEOF type
                    | expression QMARK expression COLON expression
                    | expression assignmentoperator expression
                    ;

primarypointOpt
               ::= /*empty*/
                    | primary POINT
                    ;
primary        ::= primaryfin | primaryrec
                    ;
primaryfin     ::= LPAR expression RPAR
                    | literal
                    ;
primaryrec     ::= primarypointOpt ident argumentsOpt
                    | primarypointOpt THIS argumentsOpt
                    | primarypointOpt SUPER arguments
                    | primary OPENSBRACKET expression CLOSESBRACKET
                    | primarypointOpt SUPER POINT ident argumentsOpt
                      //primarypointOpt aus LR(1)-Gr{"u}nden
                    | primarypointOpt NEW name squarebracketStar
```

```
                  OPENBLOCK varinitlist CLOSEBLOCK
                | primarypointOpt NEW name
                  squaredexpList squarebracketStar
                | primarypointOpt NEW name arguments
                | primary POINT CLASS
                ;

squaredexpList::= OPENSBRACKET expression CLOSESBRACKET
                | squaredexpList OPENSBRACKET expression CLOSESBRACKET
                ;
argumentsOpt  ::= /*empty*/
                | arguments
                ;
arguments     ::= LPAR expressionStar RPAR
                ;

literal       ::= number | float | qliteral | TRUE | FALSE
                ;

prefixopStar  ::= /*empty*/
                | prefixop prefixopStar
                ;
prefixop      ::= DOUBLEPLUS | DOUBLEMINUS | PLUS
                | MINUS | prefixopNonminusplus
                ;
prefixopNonminusplus
              ::= TILDE | EXMARK
                ;
prefixopNonminusplusStar
              ::= /*empty*/
                | prefixopNonminusplus prefixopNonminusplusStar
                ;

postfixopStar ::= /*empty*/
                | postfixop postfixopStar
                ;

postfixop     ::= DOUBLEPLUS | DOUBLEMINUS
                ;
infixop       ::= STAR | SLASH | PERCENT | PLUS
                | MINUS | DOUBLELANGLE | DOUBLERANGLE
                | TRIPLERANGLE | LANGLE | RANGLE
                | SMALLEREQUAL | BIGGEREQUAL | DOUBLEEQUAL
                | NOTEQUAL | AND | CIRCONFLEX | OR
                | CONDITIONALAND | CONDITIONALOR
                ;
assignmentoperator
              ::= EQUALS | STAREQUALS
                | SLASHEQUALS | PERCENTEQUALS | PLUSEQUALS
```

```
                  | MINUSEQUALS | DOUBLELANGLEEQUALS
                  | DOUBLERANGLEEQUALS | ANDEQUALS
                  | CIRCONFLEXEQUALS | OREQUALS
                  ;
```

Umsetung von Tabelle 4.12.

Cup

```
variabledeclaratorStar
              ::= /*empty*/
                  | variabledeclaratorList
                  ;
variabledeclaratorList
              ::= variabledeclarator
                  | variabledeclarator COMMA variabledeclaratorList
                  ;
variabledeclarator
              ::= ident squarebracketStar
                  | ident squarebracketStar EQUALS varinit
                  ;
varinitOpt    ::= /*empty*/
                  | EQUALS varinit
                  ;
varinit       ::= expression
                  | OPENBLOCK varinitlist CLOSEBLOCK
                  ;
varinitlist   ::= varinit
                  | varinitlist COMMA varinit
                  ;

methodheader  ::= modifierStar typevoid ident formparams
                  squarebracketStar throwsOpt
                  ;
constructorheader
              ::= modifierStar ident formparams throwsOpt
                  ;

formparams    ::= LPAR formparamStar RPAR
                  ;
formparamStar ::= /*empty*/
                  | formparam
                  | formparam COMMA formparamStar
                  ;
formparam     ::= type ident squarebracketStar
                  | FINAL type ident squarebracketStar
                  ;
throwsOpt     ::= /*empty*/
                  | THROWS namelist
                  ;
```

Umsetung von Tabelle 4.13.

Cup

```
typevoid      ::= type | VOID
                  ;
type          ::= name | basictype | type ARRAYDEPTH
                  ;
basictype     ::= BOOLEAN | BYTE | CHAR | DOUBLE | FLOAT
                  | INT | LONG | SHORT
                  ;
namelist      ::= name
                  | name COMMA namelist
                  ;
name          ::= ident
                  | identlist ident
                  ;
identlist     ::= ident POINT
                  | identlist ident POINT
                  ;
identOpt      ::= /*empty*/
                  | ident
                  ;
ident         ::= ID
                  ;
number        ::= INTNUM
                  ;
float         ::= FNUM
                  ;
qliteral      ::= LIT
                  ;
squarebracketStar
              ::= /*empty*/
                  | ARRAYDEPTH squarebracketStar
                  ;

modifierStar  ::= /*empty*/
                  | modifier modifierStar
                  ;
modifier      ::= PUBLIC | PROTECTED | PRIVATE
                  | STATIC | ABSTRACT | FINAL
                  ;
```

Tabellenverzeichnis

Literaturverzeichnis

[Amb02] Scott W. Ambler. *Agile Modelling: Effective Practices for EXtreme Programming and the Unified Process.* John Wiley & Sons Inc, 2002.

[ASU99] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullmann. *Compilerbau - 2.Auflage.* Oldenbourg Wissenschaftsverlag GmbH, 1999.

[Bec00] Kent Beck. *Extreme Programming - Das Manifest.* Addison Wesley, 2000.

[BH98] Bernhard Bauer, Riitta Höllerer. *Übersetzung objektorientierter Programmiersprachen.* Springer-Verlag, 1998.

[Bre02] Matthias Brenner. *Modellcharakter - Generativer Entwicklungsprozess mit UML.* iX, 3/2002.

[Bro98] Manfred Broy. *Informatik - Eine grundlegende Einführung.* Springer-Verlag, 1998.

[Eck00] Robert Eckstein. *XML - kurz und gut.* O'Reilly Verlag, 2000.

[Fla02] David Flanagan. *Java in a Nutshell.* O'Reilly Verlag, 2002.

[GJS96] James Gosling, Bill Joy, Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[Kuk01] Konstantin Kukushkin. *Testumgebung für Agile Modeling.* TU-München, 2001.

[Oes01] Bernd Oestereich. *Die UML - Kurzreferenz für die Praxis.* Oldenbourg Wissenschaftsverlag GmbH, 2001.

[Rot02] Christel Rotter. *Durchmodelliert - UML-basiertes Entwicklungswerkzeug Bold for Delphi.* iX, 5/2002.

[Rum02] Bernhard Rumpe. *Agile Softwareentwicklung mit der UML.* TU-München, 2002, Fassung vom 1.6.2002.

[SVB02] Thorsten Sturm, Jesco von Voss and Marco Boger. *Generating Code for UML with Velocity Templates.* Gentleware AG, 2002.

[VDT99] Velocity Documentation Team. *Velocity User Guide.* Apache Software Foundation, 1999.

[Ver00] Gerhard Versteegen. *Projektmanagement mit dem Rational Unified Process.* Springer-Verlag, 2000.

Index