Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Prof. Dr.-Ing. Stefan Jähnichen

Portierung, Erweiterung und Integration des ObjectTeams/Java Compilers für die Entwicklungsumgebung Eclipse

Berlin, den 13. Dezember 2003

Diplomarbeit von

Markus Witte
Matrikelnummer: 189585
<markus.witte@first.fraunhofer.de>
<macwitte@cs.tu-berlin.de>

Angefertigt unter der Leitung und Betreuung von
Dr.-Ing. Stephan Herrmann

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den 13. Dezember 2003

_____
Unterschrift

Inhaltsverzeichnis

1 Einleitung....5
2 Konzept....6
2.1 Vorgehensweise....6
2.1.1 Analyse....7
2.1.2 Entwurf....7
2.1.3 Implementierung....7
2.2 Qualitätsziele....7
2.3 Wiederkehrende Arbeitsschritte....8
2.3.1 Qualitätssicherung....8
2.3.2 Dokumentationsstandards....8
2.3.3 Refaktorisierung....8
3 Analyse....9
3.1 Compilerbau....9
3.1.1 Aufbau eines Compilers....9
3.1.2 Parsergeneratoren....10
3.1.3 Grammatik....10
3.2 Aufbau einer Java Class-Datei....14
3.2.1 Constant-Pool....14
3.2.2 Bytecode....14
3.3 ObjectTeams....15
3.3.1 Implicit-Inheritance....16
3.3.2 Translation Polymorphism....17
3.4 Eclipse-Java Plugin....18
3.4.1 Plugin Development Eclipse....18
3.4.1.1 Architektur....19
3.4.1.2 SDK....20
3.4.1.3 PDE....21
3.4.1.4 Extensionpoints....22
3.4.1.5 Beispiel....23
3.4.2 Eclipse Java Plugin....24
3.4.2.1 JDT....25
3.4.2.2 JDT-Core....25
3.4.2.3 JavaBuilder....26
3.4.2.4 IncrementalImageBuilder....27
3.4.2.5 Compiler....29
3.4.3 Eclipse Java Compiler....29
3.4.3.1 Eclipse Grammatik....30
3.4.3.2 Eclipse Scanner....32
3.4.3.3 Eclipse Parser....33
3.4.3.4 Eclipse Datenstrukturen....37
3.4.4 Phasenmodell des Eclipse-Java-Compilers....41
3.4.4.1 Phasenmodell....42
3.4.4.2 Phasenablauf....43
3.4.4.3 Beschreibung der Phasen....44

4 Entwurf....49
4.1 Implicit-Inheritance....49
4.1.1 Copy-Inheritance in ObjectTeams/Java....49
4.1.2 Dynamisches Binden von Rollenklassen....50
4.1.3 Typkonformität von Rollenklassen....52
4.1.4 Bytecodecopy....56
4.1.5 Bytecodetransformation....57
4.1.6 Transformation der Constant-Pool Referenzen....58
4.2 Translation Polymorphism....60
4.2.1 Smart Lifting....60
5 Implementierung....61
5.1 ObjectTeams-Prototyp....61
5.1.1 Vertikaler Prototyp....61
5.1.2 Horizontaler Prototyp....61
5.1.2.1 Grammatikerweiterung....62
5.1.2.2 Präprozessor für optionales replace....67
5.1.3 Parsergenerierung....68
5.1.4 AST-Erweiterung....70
5.2 Portierung....71
5.2.1 Was wurde portiert?....72
5.3 Implementierung....73
5.3.1 Implementierung des Zustandsautomaten....73
5.3.1.1 Zustände des Zustandsautomaten....75
5.3.1.2 Implementierung des Zustandsautomaten....80
5.3.1.3 Implementierung der Zustände....81
5.3.2 Implementierung der Bytecodecopy/transformation....83
5.3.3 Generisches Austauschen von AstNodes....84
5.3.4 Implementierung von Translation Polymorphism....85
5.3.4.1 LiftingEnvironment....85
5.3.4.2 LiftTo-Methode....85
5.3.4.3 Base-Tag-Fake....85
5.3.4.4 Rollenkonstruktoren....85
6 Zusammenfassung....86
6.1 Fazit....86
6.1.1 Erledigte Arbeiten....86
6.1.2 Offene Arbeiten....87
6.1.3 Optimierungen....88
6.2 Resümee der Vorgehensweise....90
6.3 Ausblick....90
7 Anhang....92
7.1 Verwendete Arbeitsmittel....92
7.2 Abbildungsverzeichnis....93
7.3 Quellenverzeichnis....94

1 Einleitung

Ziel

Das Ziel dieser Diplomarbeit ist die Realisierung eines *ObjectTeams/Java-Compilers*, welcher als Basis für die Entwicklung einer vollständigen ObjectTeams Entwicklungsumgebung verwendet werden kann.
Neben den theoretischen Aspekten wird in dieser Diplomarbeit die Vorgehensweise beschrieben, mit der die Portierung, Erweiterung und Integration des ObjectTeams-Compilers durchgeführt wurde.

Warum Eclipse?

Für die Auswahl einer geeigneten Entwicklungsumgebung, in welche sich das ObjectTeams-Programmiermodell integrieren läßt, gab es folgende Kriterien. Die Entwicklungsumgebung soll eine offene, erweiterbare, gut strukturierte und gut dokumentierte Architektur haben, und zudem nicht durch proprietäre Lizenzbestimmungen eingeschränkt sein. Die frei verfügbare Eclipse-IDE [1] erfüllt diese Kriterien und bietet außerdem noch einen vollständig als Java-Sourcecode vorliegenden inkrementellen Java-Compiler. Dieser kann als externer Batchcompiler oder innerhalb seiner eigentlichen JDT-Entwicklungsumgebung verwendet werden. Aufgrund des Eclipse-Konzeptes ist für den ObjectTeams-Compiler eine ganze Toolpalette denkbar. Angefangen bei einem komfortablen Editor über Wizards, bis hin zu Werkzeugen zum Debuggen und zum Refactoring.

Problematik

Die besondere Problematik besteht darin, dass kein Softwaresystem vollständig neu entwickelt, sondern ein bereits bestehendes System erweitert werden soll. Dieses innerhalb der Softwaretechnik nicht neue, aber keinesfalls leicht lösbare Problem, wird noch dadurch verkompliziert, dass es sich bei dem als Referenzsystem vorhandenen ObjectTeams-Compiler und bei dem Eclipse-Java-Compiler um Systeme handelt, an denen kontinuierlich weiterentwickelt wird.

2 Konzept

Die Integration neuer Funktionalitäten in ein existierendes Softwaresystem, ist eine bestehende Herausforderung der Softwaretechnik. Vorgehensmodelle [2] beschreiben prinzipiell in abstrakter Form, den dafür notwendigen Softwareprozess. Die konkrete Durchführung dieses Prozesses kann sich je nach Größe, Beschaffenheit und Entwicklungsstadium eines Softwaresystems unterscheiden. Keines der bekannten Modelle passt jedoch ideal auf die im Rahmen dieser Diplomarbeit durchzuführende Aufgabe. Am geeignetesten scheint die „Wiederverwendungsorientierte Entwicklung“ [2] zu sein. Existierende Komponenten oder Programmteile werden gesucht, verändert und letztendlich in einer adaptierten Form in das eigene System integriert. Für die Entwicklung eines auf Eclipse basierenden ObjectTeams/Java Compilers würde dies bedeuten, dass Eclipse untersucht, angepasst und in das eigene System integriert wird. Abgesehen von der unterschiedlichen Integrationsrichtung ist eine weitere gravierende Differenz vorhanden: es gibt noch kein eigenes System. Anstelle eines Vorgehensmodells wird eine andere häufig verwendete Strategie angewendet, nämlich ein Prozessmodell zur Durchführung von Softwareänderungen. Diese Strategie stellt das Fundament der dieser Arbeit zugrundeliegenden Vorgehensweise dar.

2.1 Vorgehensweise

Für die Erweiterung des bestehenden Softwaresystems sind Softwareänderungen notwendig. Diese fallen unter die Kategorie „Wartung von Software“ [2] („Wartung zum Hinzufügen oder Ändern von Systemfunktionen“). Wurde das Softwaresystem nicht selber konzipiert, entfällt vor dessen Erweiterung viel Aufwand darauf, Verständnis für das vorhandene System zu entwickeln. Dieses Verständnis kann nur begrenzt aus der Dokumentation des Systems gewonnen werden. Zusätzlich müssen dynamische und statische Artefakte analysiert werden, um die elementaren Abläufe und Zusammenhänge innerhalb des Softwaresystems zu verstehen. Dieses noch lückenhafte Wissen wird verwendet, um erste Implementierungsversuche, in Form eines Prototyps, durchzuführen. Bevor jedoch komplexere Funktionalitäten in das System integriert werden können, muss eine Analyse der Anforderungen für die neu zu erstellenden Programmteile und das daraus resultierende Gesamtsystem durchgeführt werden. Neue Funktionalitäten müssen implementiert werden, und Schnittstellen zwischen neuen und vorhandenen Programmteilen müssen definiert werden. Die Reihenfolge der zu implementierenden Funktionalitäten kann i.d.R. nicht willkürlich gewählt werden. Oft gibt es Abhängigkeiten zwischen unterschiedlichen Funktionalitäten, wodurch die Reihenfolge ihrer Implementierung fest vorgegeben wird. Im Gegensatz dazu wird Softwarequalität durch einen wiederkehrenden Prozess sichergestellt. Durch die Einhaltung von Dokumentations und Programmier-Standards wird eine spätere Wartbarkeit gewährleistet. Eine Testsuite stellt sicher, dass implementierte Funktionalitäten nicht durch spätere Erweiterungen funktionsuntüchtig werden.

2.1.1 Analyse

Das erste Problem ist die Analyse zweier vollkommen unterschiedlicher Softwaresysteme mit relativ hoher Komplexität, in Verbindung mit einem vollkommen neuen softwaretechnischem Hintergrund. Deswegen besteht die Analyse aus mehreren Elementen. Der vorhandene ObjectTeams-Compiler muss analysiert werden, der vorhandene Eclipse-Java-Compiler muss analysiert werden, und der theoretische Hintergrund muss analysiert werden.

2.1.2 Entwurf

Die Hauptaufgabe der Entwurfsphase ist, ein Konzept aufzustellen, wie der Integrationsprozess ablaufen soll. Dazu ist es notwendig, die in der Analyse gewonnenen Erkenntnisse auszuwerten und in ein effizientes System umzusetzen. Entschieden wurde, zum einen ein Prototyping durchzuführen, und zum anderen, dass sukzessiv an thematisch gleichen, aufeinander aufbauenden Problemen gearbeitet wird (*Concern-Modeling* [3]). Der Vorteil liegt darin, sich nur einmal mit einer speziellen Problematik intensiv auseinandersetzen zu müssen und die damit im Zusammenhang stehenden Probleme versucht systemweit zu lösen.

2.1.3 Implementierung

Die erste Stufe innerhalb der Implementierung ist die Entwicklung eines vertikalen Prototypen. Dadurch können erste Erfahrungen mit den bei der konkreten Realisierung auftretenden Problemen und den dafür notwendigen Werkzeugen gemacht werden. In der Praxis enthält dieser Prototyp eine Erweiterung am Compiler die das `within`-Statement mit Hilfe einer Sourcentransformation implementiert und es werden erste Erfahrungen mit den vom Laufzeitsystem benötigten Java-untypischen Bytecodes, konkret dem `team`-Modifier gemacht. Der nachfolgende Teil der Implementierung basiert auf einem aufgestellten Plan. Jener definiert thematisch gleiche Probleme und berücksichtigt deren Beziehungen zueinander. Dieser Plan, welcher einen prinzipiell sukzessiven Prozess definiert, wird - sofern unvorhersehbare Abhängigkeiten auftreten oder Änderungen durchgeführt werden müssen - durch gelegentliche Iterationsschritte verfeinert. Der Ablauf des Implementierungsprozesses wird durch die Einbindung einer nach diesem Plan ausgelegten Testsuite gesteuert.

2.2 Qualitätsziele

Um einer professionellen Softwareentwicklung gerecht zu werden, ist es unerlässlich, Qualitätsziele zu definieren. Neben einem strukturierten Entwicklungsprozess, gehören zu diesen Zielen für das Eclipse-ObjectTeams/Java-Plugin: Verständlichkeit, Modularität, Leistungsfähigkeit und Zuverlässigkeit. Verständlichkeit des Softwaresystems ist für die spätere Softwarewartung relevant und wird durch einheitliche Programmierrichtlinien und durch Dokumentationsstandards erreicht. Modularität der hinzugefügten Komponenten ist in Hinblick auf ein späteres Eclipse-Upgrade wichtig, zusätzlich sollen durch minimale

Invasivität, die Weichen für ein späteres Bootstrapping gestellt werden. Leistungsfähigkeit und Zuverlässigkeit sind die Grundvoraussetzung für die Etablierung eines Compilers auf dem Markt. Alle diese Ziele sind wichtig und müssen durch Konventionen sowie durch qualitätssichernde Maßnahmen während des gesamten Entwicklungsprozesses eingehalten werden.

2.3 Wiederkehrende Arbeitsschritte

2.3.1 Qualitätssicherung

Um Softwarequalität schon während des Entwicklungsprozesses sicherzustellen, wird ein System zum Verwalten unterschiedlicher Sourcenversionen eingesetzt [4]. Zusätzlich wird eine Junit-Testsuite [5] aufgesetzt, die während der Implementierungsphase zum einen sicherstellen soll, dass durch Änderungen an den Sourcen nicht bereits vorhandene Funktionalität zerstört wird, und zum anderen, dass ein geordneter Ablauf des Implementierungsprozesses durchgeführt wird. Der Aufbau der Testsuite wird daher nach bestimmten inhaltlichen Aspekten geordnet. (Angelehnt an ein *Concern-Model*). Angewendet wird ein einfaches *Whitebox*-Testverfahren [6], welches vordefinierte Programmeingaben auswertet, und mit dem erwarteten Ergebnis vergleicht. Im weiteren Entwicklungsprozess könnte eine *Blackbox*-Testsuite [6] entwickelt werden, die durch ein *Oracle* [6] (z.B. das Compiler-Referenzsystem) Fehler in einem zu testenden System ausfindig macht, oder diesem bescheinigt, eine identische Funktionalität zu haben.

2.3.2 Dokumentationsstandards

Die Dokumentation erfolgt zum einen durch bekannte Hilfmittel wie javadoc [7], aber auch durch die Kennzeichnung von Änderungen innerhalb des original Eclipse-Quellcodes, mit Hilfe von genau definierten Kommentarzeilen.

2.3.3 Refaktorisierung

Refaktorisierung (*Refactoring*) hat neben den offensichtlichen Vorteilen auch Nachteile. Sourcecode kann sehr einfach dateiübergreifend verändert werden. Dadurch wird zwar auf der einen Seite eine saubere Softwarestruktur gewährleistet, aber auf der anderen Seite das Einpflegen von externem Sourcecode erschwert. Dieses Problem trifft den ObjectTeams-Compiler bei jeder Distribution eines neuen Eclipse-Releases. Daher muss dieser nach jedem Eclipse-Releaseaufstieg selber Refaktorisiert werden. Um Fehler während des Refaktorisierungsprozesses aufzudecken ist eine Testsuite hilfreich. Deshalb sollte die Anpassung an neue Eclipse-Releases erst erfolgen, wenn eine ausreichende Testsuite verfügbar ist.

3 Analyse

Dieser Abschnitt beschreibt die im Rahmen dieser Arbeit durchgeführten Analysen. Der Schwerpunkt der Analysen liegt dabei in den umzusetzenden theoretischen Konzepten von ObjectTeams und dem Aufbau des Eclipse-Java-Plugin. Darüber hinaus wird Wissen zusammengetragen, welches zum Verständnis der Materie notwendig ist. Es werden nur Details erläutert, die im Rahmen dieser Arbeit von Bedeutung sind.

3.1 Compilerbau

Dieser Abschnitt vermittelt Basiswissen über Terminologie, Aufbau und Funktionsweise von Compilern. Dabei werden unterschiedliche Compiler-Techniken anhand von Beispielen vorgestellt. Besonderes Augenmerk wird dabei auf alle Aspekte gerichtet, die im Eclipse-Java-Compiler wiederzufinden sind.

Ein Compiler ist ein Programm, welches ein *Wort* (z.B. eine Java-Datei) einlesen kann, das unter Verwendung einer konkreten Grammatik erstellt wurde. Dabei können fehlerhafte Wörter von korrekten unterschieden werden.

3.1.1 Aufbau eines Compilers

In der Abbildung 1 sind in vereinfachter Form die einzelnen Teile eines Compilers abgebildet. Als Eingabe benötigt ein Compiler ein *Wort* (in diesem Beispiel ein Programm), welches in einer durch eine Grammatik genau spezifizierten Sprache geschrieben ist. Als erstes wird aus dem Programm in einer *lexikalischen Analyse* eine *Tokensequenz* erstellt. Dieses geschieht durch einen *Scanner***,** der aufgrund von Trennzeichen oder fest definierten *Identifiern* das Programm in einzelne *Token* zerlegen kann.

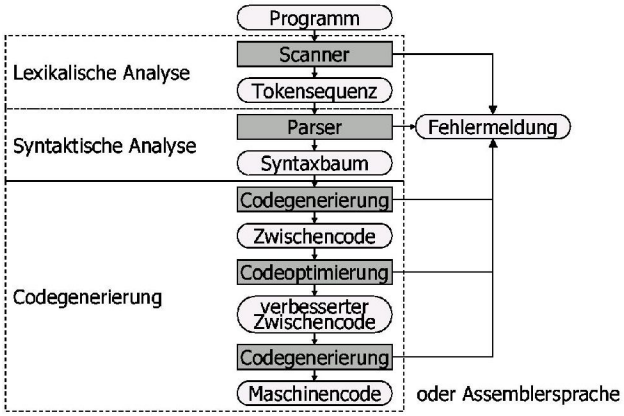


Abbildung 1 Compileraufbau [8]

Diese Tokensequenz wird in einer *syntaktischen Analyse* von einem *Parser***,** falls w ∈ L (G), in einen *Syntaxbaum* transformiert. Basierend auf diesem Syntaxbaum wird dann über einige Zwischenstufen in der Phase der *Codegenerierung*, der eigentliche *Maschinen-* oder *Bytecode* erzeugt.

Um bei der Entwicklung von Compilern Zeit zu sparen, können einige, bei allen Compilern immer wiederkehrenden Programmteile automatisch generiert werden. Dafür werden sogenannte Programmcode-Generatoren verwendet.

Scannergeneratoren (z.B LEX) erzeugen mit Hilfe von *Regulären Ausdrücken* einen Scanner, *Parsergeneratoren* (z.B. YACC) erzeugen mit Hilfe einer Grammatik einen Parser.

3.1.2 Parsergeneratoren

Das automatische generieren eines Parsers aus einer Grammatik bringt einige Vorteile mit sich. So ist es aufgrund des großen Zustandsraumes schwer, einen per Hand implementierten Parser zu testen. Fehlerquellen lauern überall und machen sich oft erst nach längerer Laufzeit bemerkbar (ganz abgesehen von direkter und indirekter Rekursion beim Parsieren). Natürlich ist es ebenso schwer einen automatisch generierten Parser zu testen, aber die verwendeten theoretischen Modelle die hinter einem Parsergenerator stehen, sind so weit ausgereift und auch bewiesen, dass bei einer korrekten Grammatik auch ein Parser erzeugt werden kann, der so funktioniert wie es von ihm erwartet wird. Fehlerquellen werden meist schon vom Parsergenerator erkannt und ggf. eliminiert. Es gibt unzählige Parsergeneratoren auf dem Markt, von denen etliche (z.B. Yacc, Bison und Jikespg) frei verfügbar sind.

Die Verwendung eines Parsergenerators ist denkbar einfach. Hat man erst eine passende Grammatik, kann das Erzeugen des Parsers z.B. durch den Aufruf jikespg Grammatik.g geschehen. Die vom Parsergenerator automatisch generierten Dateien bieten jetzt die Funktionalität, aus einer Tokensequenz, automatisch einen Syntaxbaum zu erzeugen. Parsergeneratoren können unterschiedliche Parser erzeugen. Die häufigsten verwendeten Parser sind LL(1), LR(1) oder LALR(1) Parser.

Außerdem:
Der Eclipse-Java-LALR(1)-Parser wurde mit Jikespg (siehe [9]) generiert.
Die Java Grammatik liegt den Eclipse-Sourcen bei.

3.1.3 Grammatik

Im folgenden Abschnitt soll ein Überblick über die unterschiedlichen Grammatiken gegeben werden, ohne dabei auf syntaktische Besonderheiten einzelner Parsergeneratoren einzugehen.

Eine Grammatik ist definiert als
Grammatik = { Nonterminals , Terminals , Produktionsregeln , Startsymbol }.
Beispiel: Grammatik = { {S}, {a,b,e}, { 1. S -> SaSb, 2. S -> e }, S }

Parser verwenden unterschiedliche Techniken zum Parsieren eines Wortes. Die am häufigsten verwendeten Techniken sind:

Top-Down	LL(1)
Bottom-Up	LR(1)/LALR(1)

Was diese Abkürzungen genau bedeuten, soll anhand eines Parsierungsbaumes für die obige Beispielgrammatik und das Wort **ab** erläutert werden.

Top-Down Parser fangen in einem Suchbaum mit dem *Startsymbol* an, um das gesuchte Wort zu erzeugen. Dazu werden systematisch alle möglichen *Produktionsregeln* auf die im Wort enthaltenen *Nonterminalen Symbole* angewendet. Hier im Beispiel können auf das Startsymbol S beide Produktionsregeln angewendet werden. Dieses wird rekursiv für jeden Knoten wiederholt, bis der gesamte Suchbaum durchlaufen ist. Knoten mit Wörtern, aus denen in keinem Fall mehr das gesuchte Wort erzeugt werden kann, werden natürlich aus dem Suchraum entfernt. (Man überprüft übrigens aus Performancegründen nicht das ganze Wort, sondern lediglich das nächste (n) *Terminalen Symbole*). LL(1) ist solch ein Parsertechnik. LL(1) bedeutet **L**inks nach Rechts, **L**inksableitung mit einem **(1)** im voraus gelesenem Zeichen.

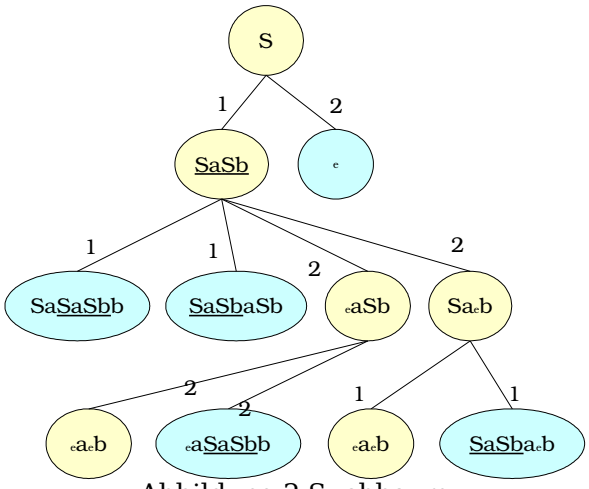


Abbildung 2 Suchbaum

LL(1) Parser werden direkt durch Methoden implementiert. Beispiel anhand der ersten Produktionsregel: S->SaSb.

S->SaSb

```
01 procedure S {
02   call S;
03   read a;
04   call S;
05   read b;
06 }
```

Beispiel 1

Ein Hauptnachteil von rekursiven Treeabstiegs-Parsern ist ihre Speichergrösse zur Laufzeit.

Bottom-Up Parser fangen im Suchbaum unten an und versuchen die Produktionsregeln in umgekehrter Richtung solange anzuwenden, bis das Startsymbol abgeleitet werden kann. LR(n) Parser arbeiten im Gegensatz zu den meisten LL(n) Parsern mit Tabellen, die von einem Parsergenerator erzeugt werden. Die Erzeugung dieser Tabellen würde den Rahmen dieser Ausarbeitung sprengen, so dass für nähere Informationen dazu auf einschlägige Fachliteratur [10][11] verwiesen werden muss.

STATE	ACTION			GOTO
	e	a	b	S
0	r2	r2	error – b not expected	1
1	accept	s2	error – b not expected	error
2	error – expecting a or b	r2	r2	3
3	error – expecting a or b	s4	s5	error
4	error – expecting a or b	r2	r2	6
5	r1	r1	error – b not expected	error
6	error – premature end of input	s4	s7	error
7	error	r1	r1	error

Tabelle 1 Action GOTO Tabelle

Im folgenden wird eine Handsimulation eines LR-Parsers ausgeführt. Dazu verwenden wir die vom Parsergenerator erzeugte Action-Goto Tabelle (siehe Tabelle 1). Diese Tabelle enthält alle im Parser möglichen Zustände (State) und deren Übergänge (GOTO).
Um das Wort **ab** abzuleiten, wird dieses auf den Parse-Stack gelegt. Begonnen wird mit Zustand **0**, weshalb dieser Wert - vom Wort durch ein **e** (Epsilon) getrennt - ebenfalls auf den Stack gelegt wird. Jetzt kommt die Action-Goto-Tabelle ins Spiel:

Zeile	Parse-Stack	Sequenz	Action
1	0e	abe	reduce 2 : S -> e
2	0Se	abe	goto 1
3	0S1e	abe	shift 2
4	0S1a2e	be	reduce 2 : S -> e
5	0S1a2Se	be	goto 3
6	0S1a2S3e	be	shift 5
7	0S1a2S3b5e	e	reduce 1 : S -> SaSb
8	0Se	e	goto 1
9	0S1e	e	accept

Tabelle 2 Ableitung des Wortes ab

Zeile 1: Wir sehen auf dem Parse-Stack eine **0** ein Trennzeichen **Epsilon** und lesen in der Sequenz ein **a**. Das heißt, wir befinden uns in Zustand **0** und lesen ein **a**. Die Tabelle gibt die Anweisung **r2** (Reduce 2) was soviel heißt wie, wende die Regel 2 auf das Epsilon an, aber in umgekehrter Richtung, und füge das Ergebnis vor dem Epsilon ein.

Zeile 2: Der neue Stack besagt jetzt, dass wir im Zustand **0** ein **S** lesen. Die Action-Goto-Tabelle liefert uns die Anweisung **GOTO** Zustand **1**. Dieser wird ebenfalls auf dem Stack vermerkt.

Zeile 3: In Zustand **1** wird ein **a** gelesen. Die Tabelle gibt die Anweisung **s2** (Shift 2) was soviel heißt wie, verschiebe das **a** und gehe in Zustand **2** über.

Zeilen 4-6 analog zu Zeile 1,2,3.

Zeile 7: Wir befinden uns im Zustand **5** und lesen ein **Epsilon**. Die Action-Goto-Tabelle gibt die Anweisung **r1**. Jetzt wird zum ersten Mal eine Produktionsregel angewendet, die wirklich den Stack abbaut. Wir erinnern uns, dass Regeln in umgekehrter Reihenfolge angewendet werden (Bottom-Up).

Zeile 8 analog zu Zeile 2.

Zeile 9: Im Zustand **1** liest der Parser ein **Epsilon** und die Tabelle liefert ihm ein **accept**. Dieses Wort konnte also vom Parser gelesen werden. Wörter die nicht mit der LR-Grammatik erzeugt werden können, enden in einer **error** Anweisung.

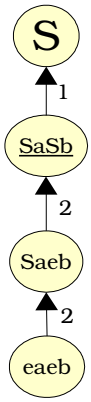


Abbildung 3 Reduce

Um jetzt den Syntaxbaum aufzubauen, müssen nur die **Reduce** Anweisungen beim Parsen ausgewertet werden. Der Vorteil gegenüber einem Top-Down-Parser ist, dass wir nicht ein leeres Gerüst nach und nach mit Inhalt füllen, sondern, dass wir beim kleinsten Element anfangen, und daraus immer größere Elemente des Syntaxbaumes aufbauen.

Der Nachteil von Tabellengesteuerten Parsern ist die Größe der Action-Goto Tabelle. Diesen Umstand versucht man durch das Zusammenfassen gleichartiger Zustände zu verbessern. Solche optimierten LR-Parser nennt man LALR-Parser. Der Nachteil den diese Optimierung mit sich bringt, ist, dass LALR-Parser bestimmten Einschränkungen unterliegen, und daher nur eine echte Teilmenge aller LR-Grammatiken verstehen.

Optimierte Action-Goto Tabelle

STATE	**ACTION**			**GOTO**
	e	**a**	**b**	**S**
0	r2	r2	error – b not expected	1
1	accept	s2	error – b not expected	error
24	error – expecting a or b	r2	r2	36
36	error – expecting a or b	s4	s5	error
57	error – expecting a or b	r1	r1	36

Tabelle 3 Optimierte Action Goto Tabelle

3.2 Aufbau einer Java Class-Datei

Dieser Abschnitt liefert das notwendige Wissen über den Aufbau einer kompilierten Java-Datei. Dabei wird keinesfalls der Anspruch auf eine vollständige Definition erhoben. Für ein tiefergehendes Verständnis der Materie verweise ich auf die JVM-Spezifikation [12]. Jede Java-Klasse oder Java-Interface wird in seiner kompilierten Form durch ein Binary (*Class-File*) repräsentiert. Die Struktur eines Binaries läßt sich grob in zwei Teile einteilen. Der erste Teil ist der Constant-Pool (siehe unten). Dieser Constant-Pool stellt die Menge aller innerhalb des Binaries referenzierten Typen (z.B. Klassen, Methoden) und Konstanten (z.B. Strings) dar. Der zweite Teil ist der Java-Bytecode.

3.2.1 Constant-Pool

Ein *Constant-Pool* [12] ist prinzipiell nichts anderes als eine Liste von Constant-Pool-Einträgen. Jeder Eintrag hat einen eindeutigen Typ, z.B. *MethodRef* oder *Class* und kann innerhalb seiner Strukturdaten auf weitere Constant-Pool-Einträge verweisen. Siehe Tabelle 4.

Strukturhierarchie der Constant-Pool-Einträge

Hierarchische Strukturelemente		**Elementare Strukturelemente**	
NameAndType	Utf8	Utf8	Bytefolge
Class	Utf8	Integer	Bytefolge
InterfaceMethodRef	Class NameAndType	Long	Bytefolge
MethodRef	Class NameAndType	Float	Bytefolge
FieldRef	Class NameAndType	Double	Bytefolge
String	Utf8		

Tabelle 4 Strukturhierarchie der Constant-Pool Einträge

3.2.2 Bytecode

Der *Bytecode* eines *Class-Files* ist nach Java-Elementen strukturiert (Klassen, Methoden, ...) und enthält die vom Compiler erzeugten JVM-Befehle. Jeder Befehl besteht aus einem *Opcode* und einer optionalen Liste von *Operanden*. Die genaue Definition der Opcodes kann in der JVM-Spezifikation [12] nachgeschlagen werden. Operanden können Referenzen auf den Constant-Pool enthalten.

3.3 ObjectTeams

Aspektbegriff

Ein Ansatz der aspektorientierten Softwareentwicklung ist, nach ähnlichen Aspekten kategorisierbare, verstreute Programmfunktionalität (*crosscutting Concerns*), in separate Module auszulagern. Die in diesen Modulen definierten *Aspekte* der Software, werden dann z.B. von einem Compiler nachträglich in den Kontrollfluß eines Programmes eingewebt.

Das ObjectTeams Programmiermodell

ObjectTeams ermöglicht zusätzlich zu dieser Modularisierung, die Kollaboration mehrerer Objekte innerhalb eines speziellen instanziierbaren Kontextes: dem Team. Die miteinander auf Team-Ebenen agierenden Objekte werden Rollen genannt. Rollenklassen stellen eine innere Klasse mit erweiterter Funktionalität dar. So kann jede Rolle eines Teams an eine gewöhnliche Anwendungsklasse (Basis) gebunden werden und mit dessen Kontrollfluss verwoben werden. Dieses nachträgliche Einweben wird innerhalb einer Rolle durch die Definition von Callin- und Callout-Bindings spezifiziert. Im Folgenden werden die hier kurz skizzierten ObjectTeams Sprachkonzepte erläutert. Für eine umfangreiche Definition der ObjectTeams Spracheigenschaften, sei auf die ObjectTeams Sprachdefinition [13] verwiesen.

Team

Das *Team* stellt das größte strukturelle Element innerhalb des ObjectTeams Konzeptes dar. In einem Team wird die Kollaboration mehrerer Klassen modularisiert. Die miteinander agierende Klassen innerhalb eines Teams bezeichnet man als Rollen. Methoden und Felder des Teams dienen der Interaktion mit dieser Rollen. Teamklassen können wie „normale“ Klassen benutzt werden. Sie sind instanziierbar und können vererbt werden. Innerhalb der Vererbungsbeziehung eines Teams gibt es einige Besonderheiten. Die Superklasse eines Teams muss immer ein Team sein. Falls nicht explizit ein Super-Team angegeben ist, erbt ein Team immer standardmäßig von `org.objectteams.Team`, und nicht von **`java.lang.Object`**. Alle Rollenklassen innerhalb eines Teams werden implizit an das Sub-Team weitervererbt. Deklariert wird ein Team in ObjectTeams/Java durch den `team`-Modifier.

Rolle

Innere Klassen eines Teams werden in ObjectTeams als *Rollen* bezeichnet und haben gegenüber „normalen“ inneren Java-Klassen, erweiterte Eigenschaften. Rollen können durch eine `playedBy`-Beziehung an eine *Basis* gebunden werden. Diese Beziehung wird für das Einweben von innerhalb der Rolle definierter Aspekt-Funktionalität verwendet. Innerhalb der Rolle kann durch sogenannte *Bindings* (*Callin-* und *Callout-Bindings*) genau definiert werden, an welchen Stellen im Kontrollfluss ein Verweben stattfinden soll. Rollen sind innerhalb eines Teams wie jede Java-Klasse instanziierbar und können zusätzlich zu ihren Bindings, Methoden und Felder enthalten.

Callin-Binding

Durch ein *Callin-Binding* kann innerhalb einer Rollenklasse deklariert werden, dass Methoden einer Basisklasse auf eine konkrete Rollenmethode abgebildet werden. Die dafür notwendige Syntax ist: `roleMethod <- baseMethod;`. Neben dieser signaturlosen Variante, kann auch die Signatur der Methoden mit angegeben werden. Dies ist besonders dann sinnvoll, wenn unterschiedliche Rückgabetypen oder Argumenttypen in der Rollenmethode und der Basismethode existieren. In diesem Fall muss durch ein anschließendes *Parameter-Mapping* innerhalb der `with`-Klausel eine Abbildung der Rückgabetypen und Argumenttypen definiert werden. Für eine Rollenmethode, die per *Callin-Binding* an eine Basismethode gebunden ist, kann mittels eines optionalen `callin`-Modifiers definiert werden, ob diese (Rollenmethode) anstelle (`replace`), vor (`before`) oder danach (`after`) aufgerufen wird.

Callout-Binding

Bei einem *Callout-Binding* ist die Aufrufrichtung umgekehrt zu der des Callin-Bindings. Durch ein Callout-Binding ist es möglich, eine Methode einer Rollenklasse auf eine Methode einer Basisklasse abzubilden. Dadurch kann erreicht werden, dass anstelle eines Aufrufes der Rollenmethode die gebundene Basisklasse aufgerufen wird. Analog zum Callin-Binding können hier ebenfalls vollständige Methodensignaturen und ein *Parameter-Mapping* angegeben werden.

3.3.1 Implicit-Inheritance

Dieses Kapitel soll einen Einblick in das durch S. Herrmann und M. Mezini geprägte *Implicit-Inheritance* Konzept und die damit verbundene Problematik vermitteln.

Implicit-Inheritance (implizite Vererbung) ist eine Vererbungsbeziehung, die nicht aufgrund einer expliziten `extends` Anweisung, sondern durch den Namen einer Klasse und ihres umschliessenden Kontextes hergestellt wird.

In ObjectTeams/Java werden zwischen zwei in einer direkten Vererbungsbeziehung stehenden Team-Klassen (`T2 extends T1`), alle im Super-Team (`T1`) definierten oder implizit geerbten Rollen-Klassen (`R1, R2`) an das Sub-Team (`T2`) vererbt.

Der Unterschied zur normalen Vererbung ist, dass die Vererbungsbeziehung zwischen Rollen aufgrund ihres Rollen-Namens manifestiert werden. Falls eine spezialisierte Sub-Rolle in einem Sub-Team notwendig ist, kann diese durch eine spezialisierende Neudefinition und eine gedachte `extends`-Beziehung zur namensgleichen Rolle im Super-Team erzeugt werden. Nur Rollenklassen innerhalb einer Teamklasse werden implizit vererbt.

In Java ist dieser Mechanismus bereits ansatzweise Verfügbar: innere Klassen und Interfaces werden implizit mitvererbt. Der Nachteil gegenüber ObjectTeams/Java ist

jedoch, dass die in Java realisierte Vererbung lediglich eine Art Sichtbarkeit darstellt, und daher keine implizite Spezialisierung zuläßt. Diese muss in Java explizit per `extends` (`Inner extends Superklasse.Inner`) hergestellt werden. Da in Java (konkret der JVM) keine Mehrfachvererbung erlaubt ist, ist es in Java folgedessen nicht möglich, eine weitere extends Beziehung zu einer spezialisierten Inneren-Klasse hinzuzufügen.

Die Abbildung 4 stellt den vollständigen Fall einer in ObjectTeams/Java gewünschten multiplen Vererbung dar. Die Rolle `R2` im Team `T2` (kurz `T2.R2`) erbt von `T2.R1` explizit und von `T1.R1` implizit. Ein standard Java-Compiler würde in diesem Fall eine Fehlermeldung über eine unzulässliche Vererbungshierarchie von `T2.R2` ausgeben.

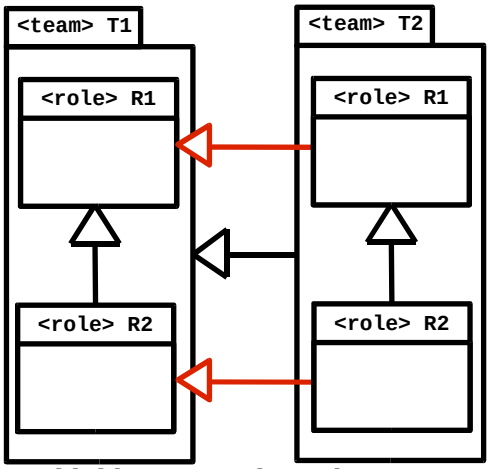


Abbildung 4 ImplicitInheritance

Die Realisierung von *Mehrfachvererbung* in Java kann auf unterschiedlichen Ebenen durchgeführt werden. So ist es denkbar, Java, bzw. die ausführende JVM, so zu erweitern, dass es die implizite Vererbung von Haus aus unterstützt. Der Nachteil wäre jedoch eine langfristige Festlegung auf eine konkrete (adaptierte) JVM. Dies würde letztendlich Performanceeinbußen oder technologiebedingte Anpassungsarbeiten an der JVM mit sich bringen. Anpassungsarbeiten am Compiler müssten zusätzlich durchgeführt werden. Die Folge wären mehrere voneinander indirekt abhängige Softwaresysteme, für die Entwicklungs- und Wartungsaufwand betrieben werden muss. Um diesen Mehraufwand möglichst gering zu halten, wird im Rahmen dieser Diplomarbeit nach einer Möglichkeit gesucht, Implicit-Inheritance ausschließlich auf der Ebene des Compilers zu realisieren.

3.3.2 Translation Polymorphism

Die Beziehung zwischen Rolle und Basis spielt in ObjectTeams eine zentrale Rolle. Basisobjekte können an eine Rolle gebunden werden. Innerhalb der Rolle wird dann über die Callin- oder CalloutBindings eine Beziehung zwischen Basis und Rolle festgelegt. Rollenobjekte erben auf diese Weise Eigenschaften vom Basisobjekt und dekorieren es mit erweiterter Funktionalität. Um diese Delegations- oder Vererbungsbeziehung zur Laufzeit durchführen zu können, bedarf es einer Relation, welche Rollen- und Basisobjekte aufeinander automatisch abbildet. Das Konzept *„Translation Polymorphism behandelt die Relation zwischen den Paaren von Rollen- und Basisobjekt und automatisiert die Navigation entlang dieser Relation“* [14]. Ähnlich eines Castings werden Rollenobjekte in Basisobjekte und umgekehrt transformiert. Die Transformation von einem Rollenobjekt zu einem Basisobjekt heißt **lowering**. Die Transformation des Basisobjekts zu einem Rollenobjekt heißt **lifting**. Lifting und Lowering müssen vom Programmierer generell nicht explizit durchgeführt werden, sondern die Umwandlung erfolgt implizit. Diese implizite Umwandlung wird immer genau dann durchgeführt, wenn ein Kontextwechsel zwischen dem die Rolle umschließenden Team und der die Basis enthaltenden Domaine stattfindet. Auf diese Weise soll sichergestellt werden, dass Rollenobjekte niemals den Kontext eines Teams verlassen.

3.4 Eclipse-Java Plugin

Eclipse

Eclipse ist eine Plattform zur Integration unterschiedlichster Anwendungen. Durch ihre gut strukturierte und offene Architektur ist es auch für Außenstehende möglich, bestehende Anwendungen zu erweitern oder ganz neu zu entwickeln. Der Erfolg der Plattform liegt nicht zuletzt an ihrem durchdachten Konzept und ihrer sehr guten und durchgängigen Dokumentation. Das die komplette IDE und auch sämtliche Sourcen inklusive einer vollständigen Java Entwicklungsumgebung kostenlos aus dem Internet heruntergeladen werden kann, hat zur Entwicklung zahlreicher Anwendungen beigetragen.

3.4.1 Plugin Development Eclipse

Der Integration von Anwendungen in die Eclipse-IDE liegt ein einheitliches Konzept zugrunde. Dieses Konzept basiert auf Komponenten (die eigentlichen Anwendungen) welche zur Laufzeit von der Plattform als Komponenten erkannt und in das System integriert werden. Die einheitliche Entwicklung dieser Komponenten, wird durch eine mitgelieferte Entwicklungsumgebung vereinfacht.
Um das Konzept für die Entwicklung von Anwendungen in den nachfolgenden Kapitel zu verstehen, werden im folgenden Abschnitt einige grundlegende Begriffe definiert, die eng mit dieser Eclipse-Plattform in Verbindung stehen.

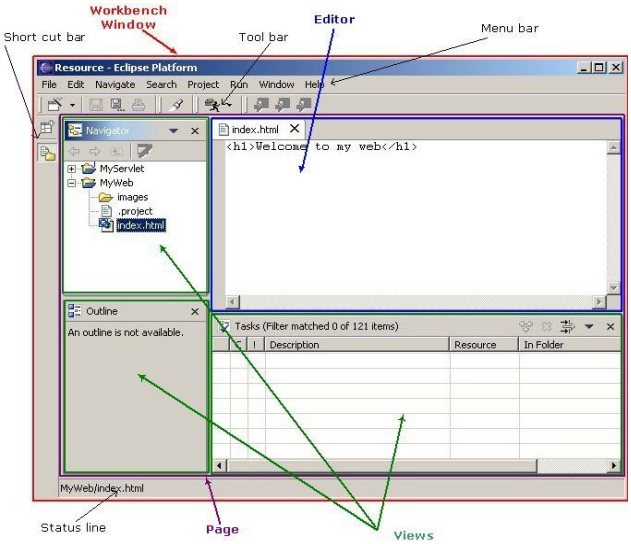


Abbildung 5 Eclipse Perspektive

Anwendungen werden in Form sogenannter *Plugins* von der Eclipse-Plattform automatisch erkannt und integriert. Dabei kann ein Plugin bestimmte Schnittstellen implementieren und sich an unterschiedlichen Stellen in der Eclipse-IDE verankern. So hat ein Benutzer durch ein Plugin die Möglichkeit *Ressourcen* in einem *Workspace***,** einem speziellen Verzeichnis innerhalb des Dateisystems, zu verwalten. Außerdem können Ressourcen innerhalb des grafisch sichtbaren Teils der Eclipse-Plattform, der sogenannten *Workbench***,** in vom Plugin definierten *Views* angezeigt, bzw. in einem *Editor* bearbeitet werden. Dabei ist es möglich, dass mehrere Plugins gleichzeitig auf dieselben Ressourcen zugreifen.

Damit das Arbeiten innerhalb der IDE auch mit vielen gleichzeitig verwendeten Plugins für den Benutzer überschaubar bleibt, kann dieser die durch das Plugin zur Verfügung gestellten Views, Editoren und Befehle für Menü- und Werkzeugleisten, in sogenannten *Perspektiven* zusammenstellen.

Die Anordnung der Views und Editoren in der Workbench ist äußerst flexibel, so dass der Benutzer die Möglichkeit hat, sich eine auf ihn zugeschnittene Entwicklungsumgebung zusammenzustellen.
Natürlich unterstützt Eclipse auch das komfortable Umschalten zwischen unterschiedlichen Perspektiven, z.B. über die Short-Cut-Bar (siehe Abbildung 5).

3.4.1.1 Architektur

Die Eclipse Plattform Architektur beruht im wesentlichen auf dem Prinzip, dass unterschiedliche Plugins in eine Laufzeitumgebung dynamisch eingebettet werden können. Diese Plugins unterliegen jedoch aufgrund ihrer Spezifikationen einigen Restriktionen, welche das Entwickeln von Plugins ohne spezielle Tools erschweren würde. Eclipse wird deshalb in seiner Standard-Version in Form einer Eclipse-SDK ausgeliefert, welche diese Tools bereits enthält. Dadurch stehen dem Entwickler neuer Eclipse-Plugins zwei mächtige Werkzeuge und eine umfangreiche API zur Verfügung, die ihn bei seiner Arbeit unterstützen.

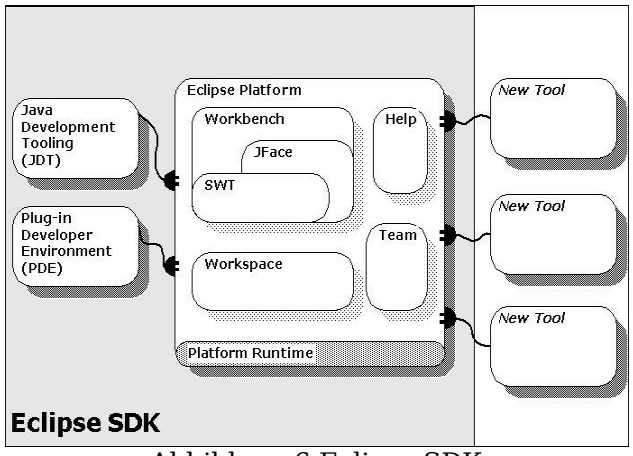


Abbildung 6 Eclipse SDK

JDT (Java development tooling)

Das JDT implementiert eine vollständige Java-Entwicklungsumgebung mit einem komfortablen Editor und Debugger. Dabei wird sowohl Syntaxhighlighting als auch das kontextbezogene Suchen unterstützt, um nur einige Vorzüge zu nennen.

PDE (Plug-in Developer Environment)

Das PDE ist eine spezielle Erweiterung, durch welche ein effizienteres Entwickeln von Plugins möglich ist. So können z.B. die Manifestdateien, welche für die Integration eines Plugins benötigt werden, über einen eigenen Editor erstellt werden.

Diese Tools dienen nicht nur ihrem offensichtlichen Zweck, sondern liefern auch ein großartiges Beispiel, wie durch den Plugin-Mechanismus neue Tools in die Plattform integriert werden können, um das System zu erweitern. So basieren die Tools auf einer Plattform API, welche dem Entwickler nützliche Funktionalitäten, für die schnelle Entwicklung eigener Plugins, zur Verfügung stellt.

Die wichtigsten Komponenten des Eclipse Software-Development-Kit werden im folgenden Abschnitt näher erläutert.

3.4.1.2 SDK

Runtime Core

Runtime Core implementiert die Laufzeitumgebung, welche die Basiskomponenten der Plattform startet und dynamisch alle Plugins hinzuladen kann. Ein Plugin ist eine strukturierte Komponente, die sich selbst dem System durch eine Manifest-Datei (plugin.xml) beschreibt. Die Plattform verwaltet eine Registry aller installierten Plugins und deren Funktionalität. Diese wird dem System durch ein allgemeines Erweiterungsmodell, welches sogenannte Extensionpoints verwendet, hinzugefügt. Extensionpoints sind klar definierte Stellen im System, die durch Plugins erweitert werden können. Außerdem können Plugins eigene Extensionpoints definieren, so dass andere Plugins diese verwenden können.

Dieser Extensionpoint Mechanismus stellt die einzige Möglichkeit dar, Funktionalität der Plattform bzw. anderen Plugins zur Verfügung zu stellen. Alle Plugins verwenden daher diesen Mechanismus. Extensions werden typischerweise in Java geschrieben unter Verwendung der Plattform API. Ein Hauptziel ist, dass der Enduser keine Speicher oder Performanceeinbußen wegen Plugins hat, die zwar installiert sind, aber nicht verwendet werden.

Resource Management

Das Resource Management Plugin definiert ein allgemeines Ressourcenmodell, um die Artefakte der Tool-Plugins zu verwalten. Plugins können Projekte, Verzeichnisse und Dateien genauso wie spezielle Ressourcen-Typen erzeugen und ändern.

Workbench UI

Die Workbench UI Plugins implementieren das Workbench Benutzerinterface und definieren eine Anzahl von Extensionpoints, welche anderen Plugins die Kontrolle über Menus und Toolbar, Drag&Drop Operationen, Dialoge, Wizards, und Benutzerdefinierten Views und Editoren ermöglichen. Die Workbench UI Plugins liefern außerdem ein Framework, welches nützlich ist, um Benutzerschnittstellen zu implementieren. Dieses Framework wird ebenfalls verwendet, um die Workbench selber weiterzuentwickeln. Dadurch wird das Entwickeln neuer Benutzerschnittstellen nicht nur vereinfacht, sondern automatisch ein einheitliches Aussehen und konsistente Handhabung erreicht.

Standard Widget Toolkit (SWT)

Das Standard Widget Toolkit ist ein Lowlevel betriebssystem-unabhängiges Widget Toolkit, welches eine portable API auf eine native OS GUI abbildet.

JFace UI framework

Das JFace UI Framework stellt komplexere Anwendungskonstrukte zur Verfügung, welche Dialoge, Wizards, Aktionen, Voreinstellungen und Widget-Management unterstützen.

Team support

Die Team Plugins erlauben anderen Plugins die Definition und Registrierung von Team-Programmierung, Repository-Zugriff, und Versionskontrolle. Die Eclipse SDK enthält ein CVS-Plugin, welches den Team support verwendet, um eine CVS-Client Unterstützung im SDK zu integrieren.

Debug support

Das Debug Plugin ermöglicht anderen Plugins sprachabhängige Programm-Starter und Debugger zu implementieren.

Help System

Das Help Plugin implementiert einen Plattform-optimierten Hilfe-Web-Server. Es definiert Extension-Points, welche von Plugins verwendet werden können, um Hilfe oder andere Browser-kompatiblen Dateiformate anzeigen zu lassen. Dabei unterstützt der Dokumenten-Web-Server spezielle Möglichkeiten, logische Plugin-basierte URLs, anstelle von statischen URLs des Dateisystems, zu verwenden.

Java Development Tooling (JDT)

Die Java Development Tooling Plugins erweitern die Plattform-Workbench durch spezielle Features wie Editierung, Darstellung, Kompilierung, Debugging, und Ausführen von Java-Code. Das JDT besteht aus einer Menge von Plugins, welche in die SDK integriert sind.

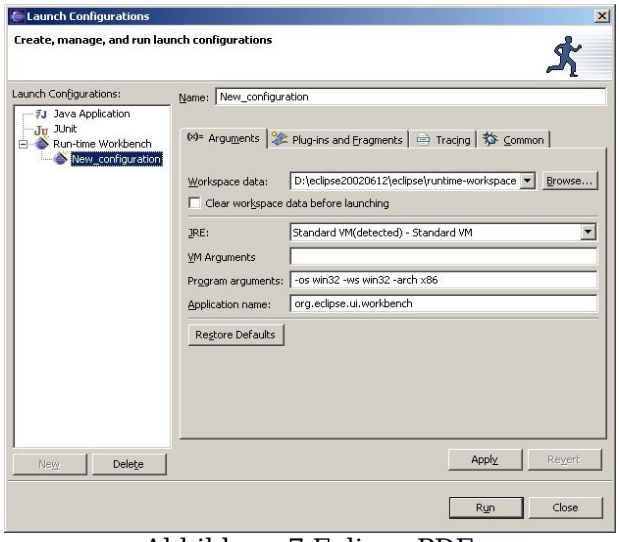


Abbildung 7 Eclipse PDE

Plugin Developer Environment (PDE)

Die Plugin Developer Environment stellt Tools zu Verfügung welche das Erzeugen, Manipulieren, Debuggen und Verwendung von Plugins automatisieren. Die PDE besteht aus einer Menge von Plugins, welche in die SDK integriert sind.

3.4.1.3 PDE

Die Plugin Development Environment wurde entwickelt, um das Entwickeln von Plugins zu erleichtern, während man selber mit der Plattform arbeitet. Es erweitert die Plattform um zahlreiche Tools wie spezielle Views, Editoren und Perspektiven. Durch diese Sammlung von Tools wird der Entwicklunsprozess von Plugins innerhalb der Workbench rationalisiert.

Abbildung 7 zeigt exemplarisch eine Erweiterung an den Programmstartoptionen. Die PDE ermöglicht dabei das Testen und Debuggen von Plugins in einer Runtime-Workbench. Um beim Ausführen eines Plugins in der externen Workbench, Konflikte zwischen gleichnamigen Plugins zu vermeiden, kann über einen Plugin-Selektor eine Menge an Plugins ausgewählt und für die Test-Workbench verwendet werden.

Neben Wizards zum Erstellen neuer Plugins, gibt es noch einen Editor zum Bearbeiten von Manifest-Dateien. In dieser Pluginbezogenen Manifest-Datei werden Schnittstellen eines Plugins über sogenannte Extensionpoints definiert.

3.4.1.4 Extensionpoints

Die Eclipse Plattform besteht aus einer Struktur, basierend auf dem Konzept der Extensionpoints. Extensionpoints sind genau definierte Stellen im System, an denen andere Tools (sogenannte Plugins) ihre Funktionalität zur Verfügung stellen können. Jedes größere Subsystem der Plattform ist in sich selber noch einmal in eine Anzahl von Plugins unterteilt, welche durch diese Extensionpoints miteinander in Verbindung stehen. Das Eclipse System wird mit dem selben Extensionpoint-Konzept entwickelt, wie sie auch für externe Plugins anderer Anbieter Verwendung findet.

Plugins können eigene Extensionpoints definieren oder Extensions für Extensionpoints anderer Plugins implementieren.

Die nachfolgende Tabelle stellt eine Übersicht über die quantitative Verwendung von Extensionpoints innerhalb einzelner Komponenten der Eclipse-SDK dar.

Komponente (Name)	Platform runtime	Workspace	Workbench	Team	Debug	Help	Other
Extensionpoints (Anzahl)	2	3	27	5	10	5	14

Tabelle 5

In der Eclipse-SDK gibt es zahlreiche Extensionpoints, die an den unterschiedlichsten Stellen innerhalb des Frameworks Verwendung finden. So gibt es einen Extensionpoint innerhalb der Workspace-Komponente, welcher die Basisfunktionalität zur Realisierung eines inkrementellen Compilers zur Verfügung stellt.

In der Abbildung 8 ist eine Übersicht über die unterschiedlichen Extensionpoints innerhalb der konkreten Workbench Komponente dargestellt. Durch diese Workbench Extensionpoints haben andere Plugins die

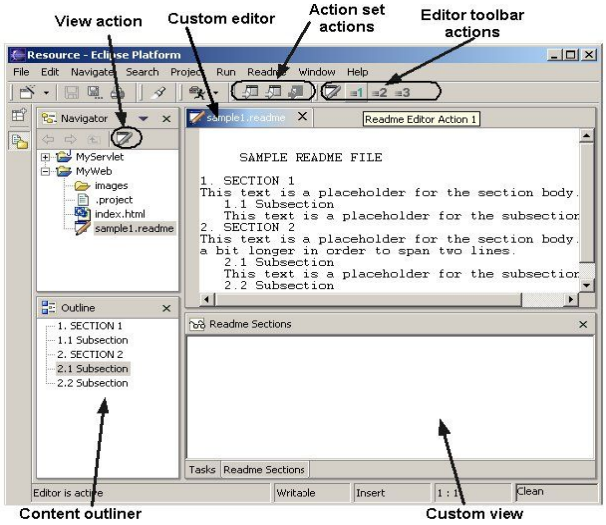


Abbildung 8 Extensionpoints

Möglichkeit, der Workbench z.B. eigene Toolbarelemente (Editor toolbar action) oder eigene Views (custom view) hinzuzufügen, die dann vom Anwender in einer gewünschten Perspektive aktiviert werden können.
Extensionpoints stehen immer in Verbindung mit einem konkreten Interface, welches dann von dem Plugin implementiert werden muss, das die Extension realisiert. Der Extensionpoint des Custom View stellt das Interface org.eclipse.ui.IViewPart zur Verfügung. Eine gängige Praxis ist es, Basisfunktionalität von einer bereits implementierten Klasse - in diesem Fall von org.eclipse.ui.part.ViewPart - zu erben.

3.4.1.5 Beispiel

Anhand eines Beispiels soll das Erstellen eines einfachen Plugins demonstriert werden. Der PDE Plugin-Wizard ermöglicht das Erstellen eines einfachen Hello-World-Plugins. Nachdem der Wizard abgeschlossen ist, finden sich im Workspace folgende Dateien: plugin.xml und HelloWorldView.java

Das Manifest (Plugin.xml) stellt eine statische Beschreibung des Plugins dar. Diese ist notwendig, damit z.B. genau definiert ist, welche Extensionpoints anderer Plugins verwendet werden, bzw., welche Extensionpoints es selber anderen Plugins zur Verfügung stellt. Außerdem enthält das Manifest Informationen über die verwendeten Klassen und den eindeutigen Identifier des Plugins.
Die eigentliche Funktionalität und Kommunikation mit anderen Plugins geschieht in der Datei HelloWorldView.java. Hier können abstrakte Methoden implementiert werden, um im View eigene Informationen anzuzeigen. Im folgenden Beispiel wird die createPartControl Methode implementiert, welche zum Erzeugen der visuellen Komponente vom Plugin-Framework aufgerufen wird.

Plugin.xml

```
01 <?xml version="1.0" ?>
02 <plugin
03   name="Hello World Example"
04   id="org.eclipse.examples.helloworld"
05   version="1.0">
06   <requires>
07     <import plugin="org.eclipse.ui" />
08   </requires>
09   <runtime>
10     <library name="helloworld.jar" />
11   </runtime>
12   <extension point="org.eclipse.ui.views">
13     <category
14       id="org.eclipse.examples.helloworld.hello"
15       name="Hello" />
16     <view
17       id="org.eclipse.examples.helloworld.helloworldview"
18       name="Hello Greetings"
19       category="org.eclipse.examples.helloworld.hello"
20       class="org.eclipse.examples.helloworld.HelloWorldView" />
21   </extension>
22 </plugin>
```

Beispiel 2

HelloWorldView.java

```
01 package org.eclipse.examples.helloworld;
02
03 import org.eclipse.swt.widgets.Composite;
04 import org.eclipse.swt.widgets.Label;
05 import org.eclipse.swt.SWT;
06 import org.eclipse.ui.part.ViewPart;
07
08 public class HelloWorldView extends ViewPart {
09    Label label;
10    public HelloWorldView() {
11    }
12    public void createPartControl(Composite parent) {
13       label = new Label(parent, SWT.WRAP);
14       label.setText("Hello World");
15    }
16    public void setFocus() {
17    }
18 }
```

Beispiel 3

In der Abbildung 9 sehen wir das Ergebnis unseres aktivierten Plugins. Es handelt sich um einen Benutzerdefinierten View, mit der statischen Bezeichnung „**Hello Greetings**“ und dem zur Laufzeit erzeugten Text „**Hello World**“.

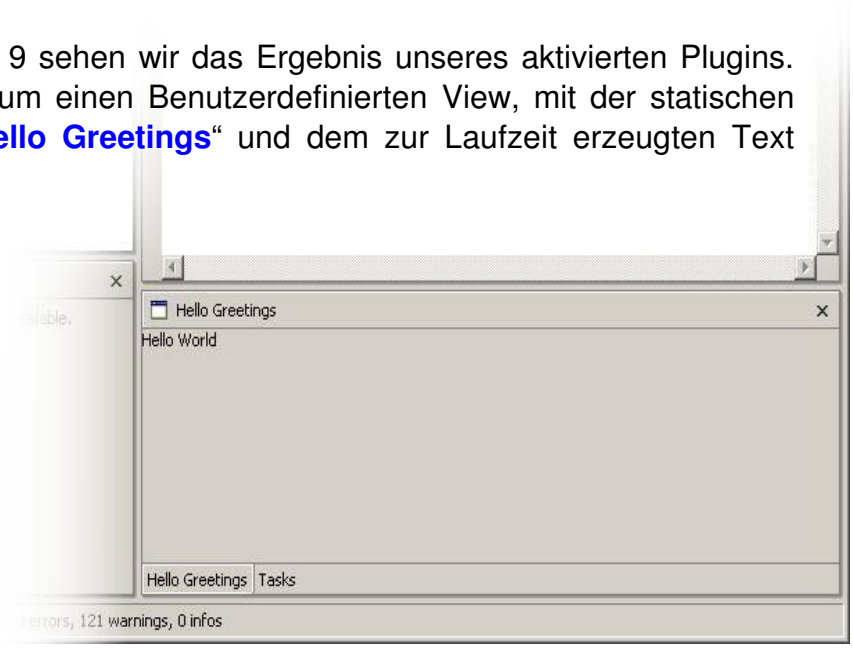


Abbildung 9 Hello World View

3.4.2 Eclipse Java Plugin

Das Eclipse-Java-Plugin stellt die Grundlage für die Entwicklung des ObjectTeams-Eclipse-Plugins dar. Basierend auf diesem Plugin, soll eine um das ObjectTeams Modell erweiterte Entwicklungsumgebung implementiert werden. Um diese Anpassung durchführen zu können, ist es neben dem im vorherigen Kapitel vermittelten allgemeinen Plugin-Konzept wichtig, die internen Strukturen und Abläufe des Eclipse-Java-Plugins zu verstehen.

3.4.2.1 JDT

Die in Eclipse integrierte Java Entwicklungsumgebung (Java Development Tools) stellt eine komfortable Java-IDE dar, mit der es möglich ist, jegliche Java-Applikation (auch Eclipse-Plugins) zu entwickeln. Sie besteht im wesentlichen aus drei Komponenten:

Benutzerschnittstelle	(User Interface)
Debugger	(Debug)
Infrastruktur	(Core)

JDT UI

Das JDT UI implementiert die Benutzerschnittstelle der Java IDE. Sie enthält neben dem Java Editor und den Wizards einen Package Viewer, eine Typhierarchieansicht und eine Strukturansicht für Java Klassen. Der *Java-Editor* unterstützt unter anderem mehrfarbiges Syntax-Highlighting, Kontextabhängige Code-Assistenten. Es gibt *Wizards* die das Erzeugen von Java-spezifischen Elementen, wie z.B. Projekte, Packages, Klassen und Interfaces, erleichtern. Der *Package Viewer* zeigt Java-Elemente aller Packages des Projektes an und Vererbungsbeziehungen zwischen Super- und Subklassen können hierarchisch in der *Typ-Hierarchie* dargestellt werden.

JDT Debug

Der integrierte Java-Debugger unterstützt das kontrollierte Ausführen von Programmen in einer Java VM. Das Setzen von Breakpoints wird ebenso unterstützt wie das schrittweise Ausführen und Überspringen von Codestellen. Das Auswerten von beliebigen Ausdrücken zur Laufzeit und die hierarchische Ausgabe von darstellbaren Werten aller Objektinstanzen ist ebenfalls standardmäßig im Debugger möglich.

3.4.2.2 JDT-Core

JDT-Core liefert die Infrastruktur für das Java-Plugin. Es stellt eine API für die Navigation im Java-Element-Tree zur Verfügung. Dieser Java-Element-Tree enthält alle zur Laufzeit notwendigen Elemente, wie Package-Informationen, zu kompilierende und bereits kompilierte Klassen, Typen, Methoden und Felder. Aufbauend auf diesem Element-Tree und der zugehörigen API, wurde die Infrastruktur um Funktionalitäten zum Suchen, z.B. für Code-Assistenten, Typhierarchie Auswertung und Refactoring, erweitert.

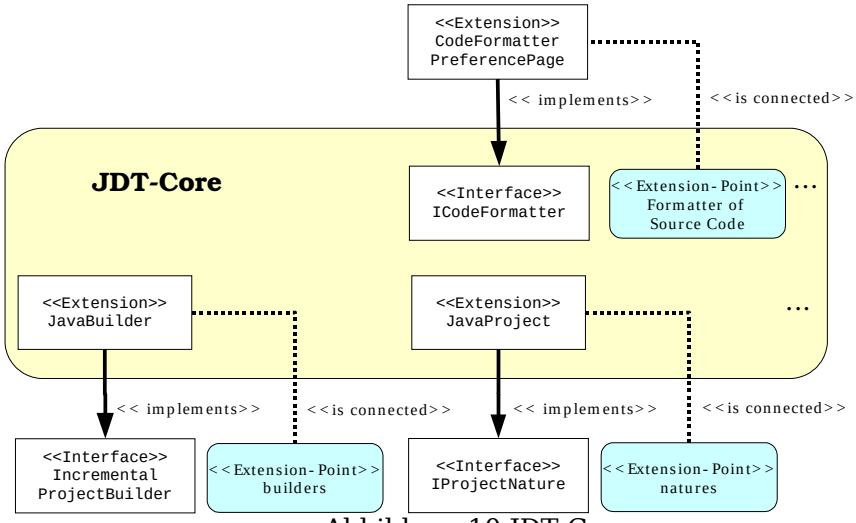


Abbildung 10 JDT-Core

Abbildung 10 stellt eine Auswahl an JDT-Core-Schnittstellen zu anderen Plugins dar. Der `Codeformatter` verwendet und implementiert einen JDT-Core-Extension-Point und bekommt somit die Möglichkeit von außen den Java-Code zu Formatieren.

Ein weiterer wichtiger Teil des JDT-Core ist der `JavaBuilder`, welcher das Interface `IncrementalProjectBuilder` implementiert.

3.4.2.3 JavaBuilder

Das Sequenzdiagramm aus Abbildung 11 zeigt die einzelnen Schritte des Build-Vorganges im Eclipse Java-Builder. Nachdem das Framework eine Änderung an den Ressourcen erkannt hat, ruft dieses die Methode `build()` des `JavaBuilder` auf. Diese instanziiert als erstes einen `BuildNotifier`, welcher im Verlauf eines Build-Vorganges Informationen über den aktuellen Zustand des Builds enthält. Nachdem sich der JavaBuilder selber initialisiert, erzeugt er eine Tabelle aller *Source-Deltas* (alle geänderten Ressourcen) und löscht den Status des letzten Build-Durchlaufes. Anhand dieses Status kann der Builder entscheiden, ob ein inkrementelles, oder ein vollständiges Build durchgeführt werden muss.

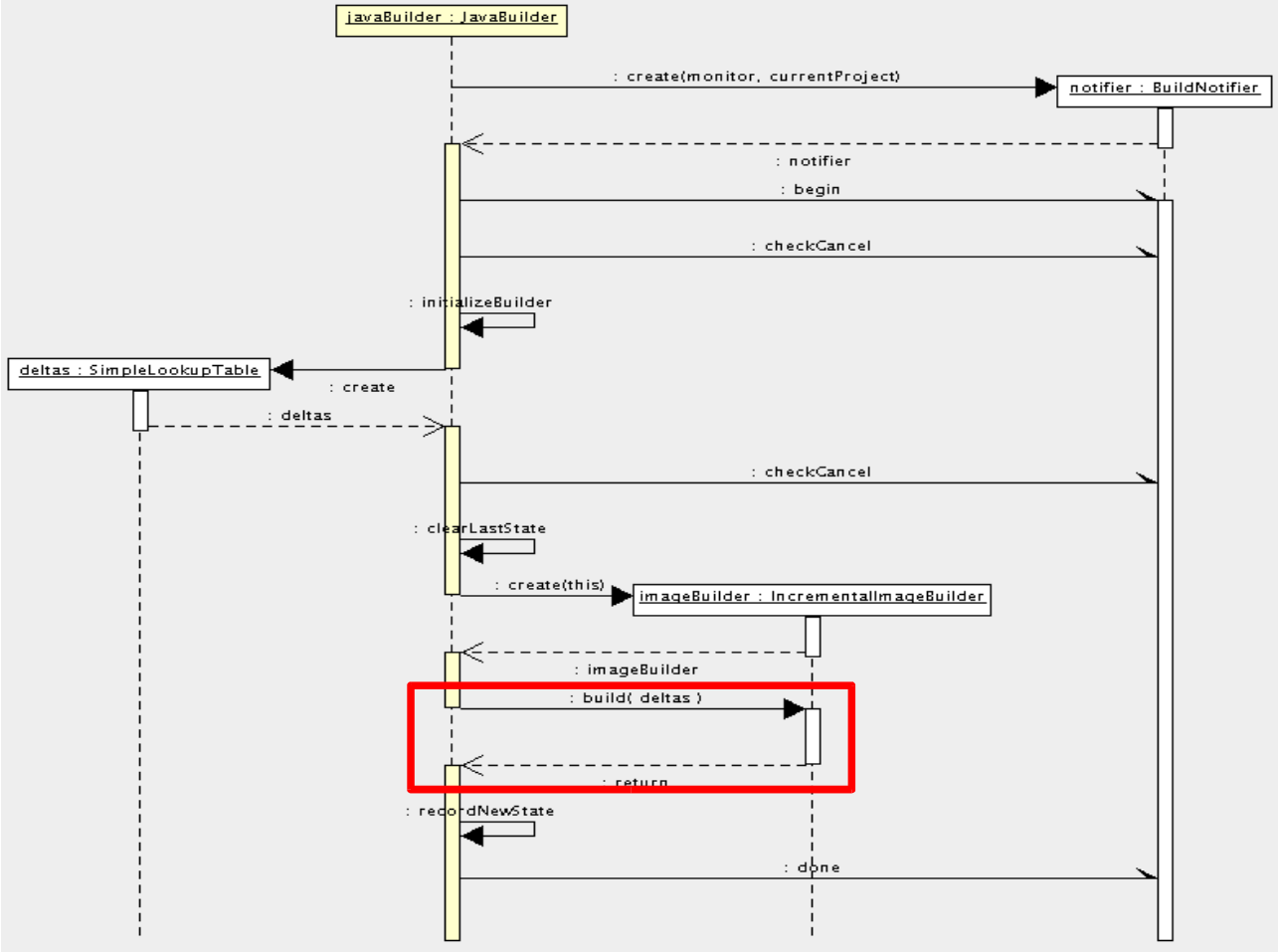


Abbildung 11 JavaBuilder

Nach einem Build-Fehler oder wenn kein Source-Delta zur Verfügung steht wird ein Komplettbuild ausgeführt.

Im Falle eines inkrementellen Builds übernimmt der `IncrementalImageBuilder` die Kontrolle über den weiteren Build Verlauf, andernfalls führt ein `BatchImageBuilder` ein vollständiges Build durch. Am Schluss wird noch der erreichte Build-Status vermerkt.

3.4.2.4 IncrementalImageBuilder

Inkrementelles Building hat Vorteile bezüglich der Performance von Projekten mit hunderten oder tausenden von Ressourcen. Compileraktivitäten z.B. nach jedem Speichern einer geänderten Datei, sind kaum merkbar. Dieser Vorteil beruht auf der Tatsache, dass auch bei großen Projekten in der Regel immer nur wenige Ressourcen gleichzeitig im Workspace geändert werden. Die technische Herausforderung für inkrementelles Building ist, exakt herauszufinden, was neu erstellt werden muss. Der `JavaBuilder` zum Beispiel, beinhaltet einen Abhängigkeitsgraphen und eine Liste von Problemen die während der Kompilierung aufgetaucht sind. Diese Information wird während des Inkrementellen Builds verwendet, um herauszufinden, welche Klassen in Abhängigkeit von der gerade ausgeführten Änderung im Sourcecode, neu kompiliert werden müssen. Obwohl die generelle Struktur für inkrementelle Builds in der Plattform verankert sind, liegt die wirkliche Arbeit beim aktiven Plugin, welches das Build ausführen soll. Ein allgemeines Konzept oder allgemeine Patterns gibt es jedoch nicht, welche diese Aufgabe übernehmen könnten.

Beispiel zum Veranschaulichen des Source-Delta Konzeptes:

A	**B**	**C**
`import B`	`import C`	
`class A{`	`class B{`	`class C{`
`   ...`	`   ...`	`   ...`
`}`	`}`	`}`

Beispiel 4 Source Delta Konzept

Der Abhängigkeitsgraph stellt die Beziehung zwischen mehreren Klassen dar. Im Beispiel oben ist A abhängig von B und B abhängig von C. Wenn vom Framework Änderungen an den Ressourcen festgestellt werden, wird ein SourceDelta gebildet, welches genaue Informationen (ADDED, CHANGED, REMOVED) über die geänderten Ressourcen enthält.

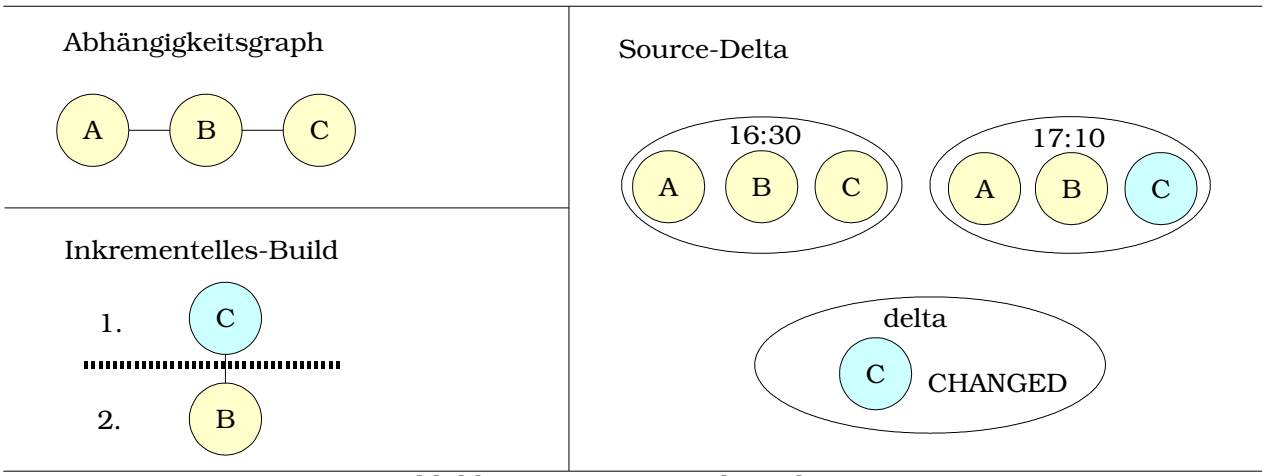


Abbildung 12 Source Delta Schema

In diesem Beispiel ist vom Framework eine Änderung an der Klasse C festgestellt worden. Diese Klasse muss also neu kompiliert werden. Die Klasse B ist abhängig von C, deshalb muss auch B neu kompiliert werden. Wenn jetzt das neu erstellte Binary der Klasse B identisch mit dem vorhanden ist, muss C nicht neu erstellt werden.

Im Folgenden soll die Funktionsweise des `IncrementalImageBuilders` nochmal anhand eines Sequenzdiagramms veranschaulicht werden.

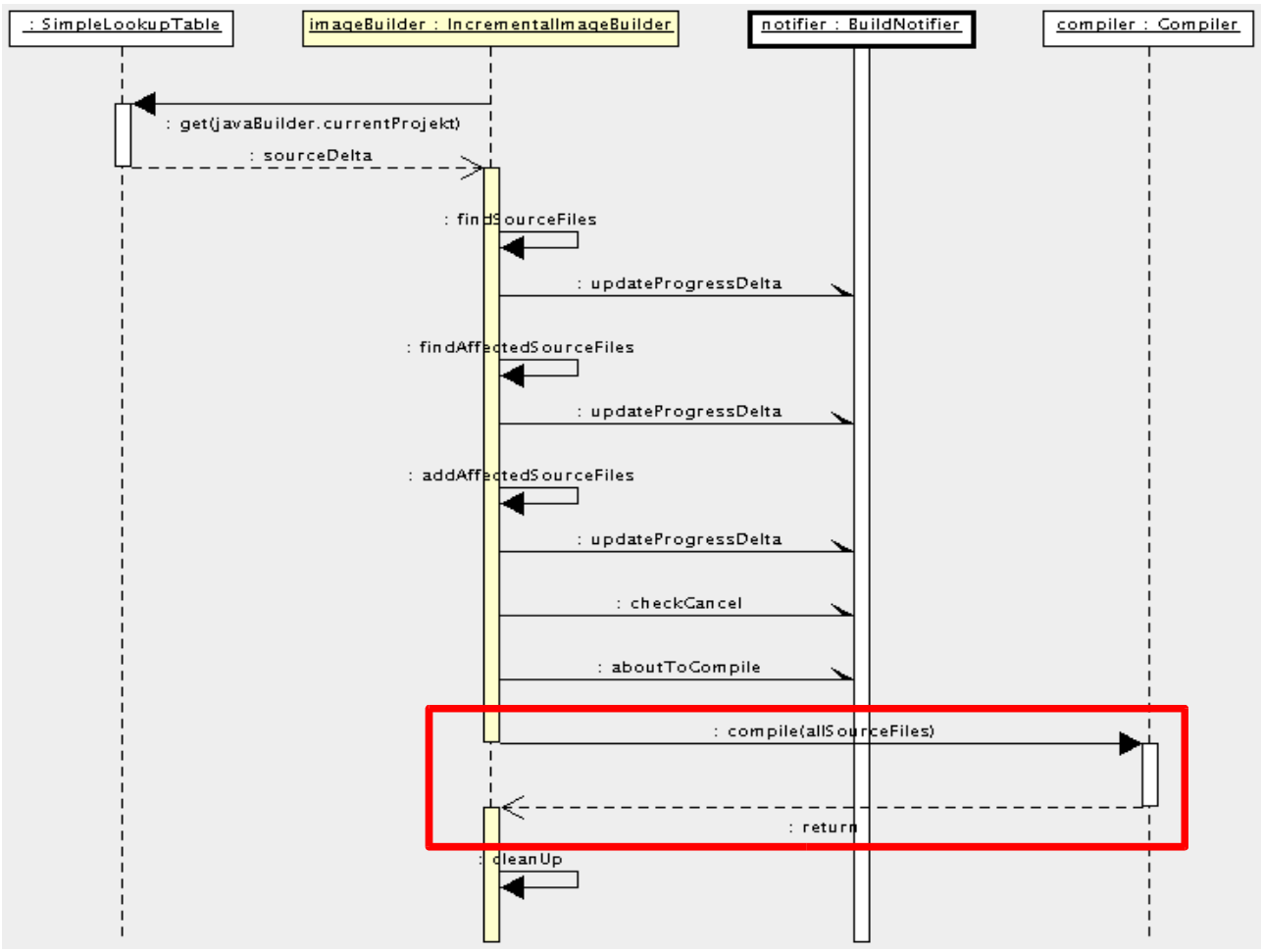


Abbildung 13 IncrementalImageBuilder

Als erstes werden die Source-Deltas aus der im `build()`-Methodenaufruf übergebenen `SimpleLookupTable` geholt und auf ihre Existenz hin überprüft (`findSourceFiles`). Dann werden alle Abhängigkeiten des Source-Deltas gesucht (`findAffectedSourceFiles`) und an die Liste der zu kompilierenden Dateien angehängt (`addAffectedSourceFiles`). Die Liste aller Dateien wird jetzt in einer Schleife solange kompiliert, bis alle Listenelemente abgearbeitet sind.

Der inkrementelle Vorteil kommt dadurch zustande, dass alle in diesem Compiler-Durchlauf erzeugten ClassFiles mit den vorhandenen verglichen werden. Nur falls ein Unterschied zwischen dem Erzeugten und dem vorhandenen ClassFile festgestellt wird, muss die zum ClassFile zugehörige SourceDatei unter Berücksichtigung der Abhängigkeiten neu kompiliert werden.

Wenn diese Liste vollständig abgearbeitet wurde, ist der inkrementelle Builder fertig.

3.4.2.5 Compiler

Die Methode `Compiler.compile()` stellt die Hauptschleife des Eclipse-Java-Compilers dar. Als Parameter bekommt diese Methode die zu kompilierenden SourceFiles übergeben. Für jede dieser SourceFiles wird der Kompilierungsvorgang angestoßen. Eine Besonderheit des Eclipse-Java-Compilers ist, dass ein SourceFile in zwei Stufen kompiliert wird. In der ersten Stufe innerhalb der Methode `beginToCompile`, wird aus der SourceUnit die Signatur einer Java-Datei erzeugt. In der zweiten Stufe innerhalb der Methode `process`, wird der eigentliche Bytecode generiert. In `acceptResult()` werden die Dateien dann physikalisch im Dateisystem erzeugt. Diese Aufgabe übernimmt der `requestor`. Da die Klasse `Compiler` eine zentraler Rolle für die Implementierung des ObjectTeams-Compilers spielt, wird die Funktionsweise und das Zusammenspiel mit anderen Klassen in den folgenden Abschnitten genauer Untersucht.

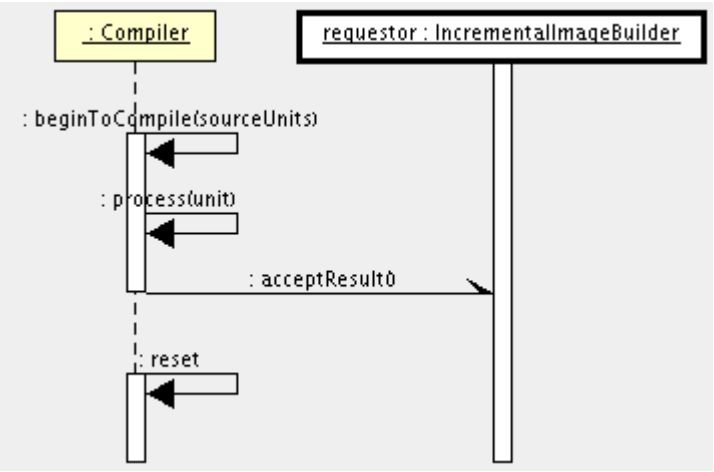


Abbildung 14 Sequenzdiagramm Compiler

3.4.3 Eclipse Java Compiler

Die auf unterster Ebene benötigten Klassen des Eclipse-Java-Compilers, sind der Scanner und der Parser. Diese Klassen stellen elementare Funktionalität zur Verfügung, um die für den Kompilierungsvorgang benötigten Datenstrukturen zu erzeugen. Die dem Parser zugrundeliegende Grammatik ist ebenfalls wichtig, da Teile des Compilers und des Parsers automatisch aus dieser Grammatik mit Hilfe des Parsergenerators jikespg generiert wurden.

Jikespg

Jikespg [9] ist als C-Quellcode erhältlich, und kann daher auf unterschiedlichen Betriebssystemen verwendet werden. Standardmäßig erzeugt Jikespg eine Menge von Java-Klassen, mit deren Hilfe ein LALR-Parser implementiert werden kann. Diese Dateien enthalten die vollständige Action-Goto Tabelle und zusätzliche Informationen, die vom Parser zur Laufzeit benötigt werden. Ein großer Nachteil ist die fehlende Dokumentation von Jikespg, so dass auf den schwer verständlichen dokumentierten Quellcode, bzw. auf Beschreibungen andere Parsergeneratoren zurückgegriffen werden muss. Glücklicherweise gibt es einige Beispiele für die Verwendung von Jikespg.

3.4.3.1 Eclipse Grammatik

Grundlage des Eclipse-Java-Compilers bildet die Grammatik. Diese wurde in einer BNF (siehe [15]) ähnlichen Notation definiert und enthält über 400 Regeln. Um Inkonsistenzen bzw. Fehler bei der Umsetzung der Grammatik in einen entsprechenden Parser zu vermeiden, verwendet man dafür in der Regel einen Parsergenerator. Zur Generierung des Eclipse-Parsers wurde der von IBM entwickelte Parsergenerator jikespg verwendet. Jikespg liest eine Grammatik ein und erzeugt daraus eine Anzahl von Tabellen in Form von Dateien. Eine Performance-optimierte Form dieser Tabellen wird vom Parser während des parsierens einer Datei ausgewertet und definiert auf diese Weise das Verhalten des Parsers. Grammatikerweiterungen müssen also über die Grammatik in den Parser hineingeneriert werden.

Neben der BNF-Notation enthält die Grammatikdatei einige Parsergeneratorspezifische Schlüsselwörter ($...), welche bestimmte Sektionen für Regeln, Terminale-Symbole o.ä definieren. Kommentare können innerhalb der Grammatik mit „--“ definiert werden.

Innerhalb der Grammatikdatei gibt es folgende, für eine Grammatik notwendigen Teile:

$Start	Startsymbol
$Terminals	Terminale-Symbole
$Rules	Produktionsregeln

Darüber hinaus gibt es noch weitere nützliche Schlüsselwörter, welche innerhalb der Grammatikdatei Verwendung finden.

$Define	Makrodefinition von Codefragmenten
$putCase	Makro für case-Teil innerhalb einer switch-Anweisung
$break	Makro für break-Anweisung
$rule_number	Nummer der aktuellen Regel
$rule_text	Die aktuelle Regel als String
$Alias In der	Grammatik verwendbare Alias Definitionen
$names	Definition von Sonderzeichen z.B. Operatoren und Klammern
$end	Markierung für das Ende der Grammatikdefinition
$empty	Epsilon (Siehe 3.1.3)

Goal stellt das Ziel für den Compiler dar. Wir erinnern uns, dass ein LALR(1) Parser die Regeln rückwärts anwendet. Wenn dieses Token vom Parser erreicht werden kann, wurde ein Wort korrekt gelesen.

```
...
165 $Start
166    Goal
...
```

Beispiel 5

Die Liste der Terminalen Symbole definiert alle Schlüsselwörter und Sonderzeichen einer Grammatik. Jedes dieser Terminalen Symbole darf nur an den durch die Regeln definierten Stellen innerhalb des Quellcodes auftreten. Nonterminale Symbole werden indirekt über die Grammatik definiert, so ist jedes Symbol, welches nicht als terminales Symbol definiert ist, automatisch ein nonterminales Symbol.

```
...
30 $Terminals
31     Identifier
32     abstract assert boolean break byte case catch char class
...
```

Beispiel 6

Der Abschnitt $Rules stellt die eigentlichen Regeln in BNF-Notation dar. Die Zuweisungen -> und ::= sind zwar semantisch identisch, haben aber innerhalb dieser Grammatik eine unterschiedliche Intention.

-> Redirektion
::= Produktionsregel

Jede Produktionsregel (siehe Beispiel 7) besteht aus der Regeldefinition (Zeile 1082) und einem optionalen Codeblock. „/. ... /.“ (Zeile 1083). Obwohl die Existenz eines Codeblockes nicht zwingend an eine Produktionsregel gebunden ist (Codeblöcke können an beliebiger Stelle innerhalb der Sektion $Rule auftreten), sollte man die Definition eines Codeblockes direkt unterhalb einer Regeldefinition durchführen, denn nur dort enthalten $rule_number und $rule_text den korrekten Wert. Statisch ergibt sich aus der Konkatenation aller Codeblöcke die vollständige `consumeRule(...)`-Methode, welche im Parser die Schnittstelle zwischen generiertem und handimplementiertem Parser-Code darstellt.

```
...
168  $Rules
...
1082 MethodInvocation ::= 'super' '.' 'Identifier' '(' ArgumentListopt ')'
1083 /.$putCase consumeMethodInvocationSuper(); $break ./
...
```

Beispiel 7

Die innerhalb der Codeblöcke verwendeten Makros sind im oberen Teil der Grammatikdatei definiert. (Siehe unten)

```
...
15 $Define
16 $putCase
17 /.    case $rule_number :  //System.out.println("$rule_text");
18       ./
19
20 $break
21 /.
22
23        break ;
24 ./
...
```

Beispiel 8

Dieser Teil (Beispiel 8) ist während der Implementierungsphase des Parsers relevant. Durch das Einkommentieren von `System.out.println(...)` in Zeile 17 ist es möglich, die Reihenfolge der vom Parser zu konsumierenden Regeln, zur Laufzeit sichtbar zu machen.

Die Sektion $Alias dient dazu, die Regeln der Grammatik lesbarer zu gestalten. Anstelle der in Sektion $Terminals definierten Bezeichner für Sonderzeichen (z.B. EQUAL_EQUAL), können mit Hilfe dieses Alias, die besser lesbaren Sonderzeichen (z.B. „==“) in die Regel geschrieben werden.

3.4.3.2 Eclipse Scanner

Der Scanner erzeugt aus einer Zeichenkette (Quellcode) eine Tokensequenz. Diese Aufgabe wird von der Methode `compiler.parser.Scanner.getNextToken()` erledigt. Die Methode wertet ausgehend von der aktuellen Scannerposition nachfolgende Zeichen aus und liefert als Ergebnis eine dem Schlüsselwort entsprechende Konstante zurück. Im Grunde ist die Implementierung dieser Methode eine auf Effizienz getrimmte große verschachtelte `switch/case`-Anweisung. Teile der Funktionalität sind in externe Methoden ausgelagert, z.B. das Lesen von Zahlen oder das Lesen von Schlüsselwörtern. Da alle Methoden ähnliche Funktionalität implementieren, soll deren Funktionsweise beispielhaft an der Methode `scanIdentifierOrKeyword()` demonstriert werden.

Scanner.java

```
2323 public int scanIdentifierOrKeyword() throws InvalidInputException {
...
2354  firstLetter = data[index];
2355  switch (firstLetter) {
...
2777  case 's' : //short static super switch synchronized strictfp
2778      switch (length) {
2779        case 5 :
...
2786          if ((data[index] == 'u')
2787            && (data[++index] == 'p')
2788            && (data[++index] == 'e')
2789            && (data[++index] == 'r')
2790            return TokenNamesuper;
2791          else
2792            return TokenNameIdentifier;
...
```

Beispiel 9

In Beispiel 9 ist die Implementierung für das Schlüsselwort `super` dargestellt. Zuerst wird der Anfangsbuchstabe (Zeile 2355) und dann die Länge der Zeichenkette (Zeile 2778) überprüft. Wenn dies noch nicht ausreicht, muss das Ergebnis noch in einer `if/else`-Anweisung verfeinert werden (Zeile 2786). Sobald das ganze Schlüsselwort erkannt wurde, kann das Token `TokenNamesuper` (Zeile 2790) zurückgeliefert werden. Sollte die Zeichenkette keinem bekannten Schlüsselwort entsprechen, wurde ein Identifier gefunden und es muss **`TokenNameIdentifier`** zurückgegeben werden (Zeile 2792).
Das Ergebnis von `getNextToken()` wird vom Parser ausgewertet um zu entscheiden, welche Aktion (Shift/Reduce/Goto) in Abhängigkeit des aktuellen Zustandes durchgeführt werden muss.

3.4.3.3 Eclipse Parser

Der in Eclipse verwendete Parser ist ein LALR(1) Parser. Grundsätzlich ist ein LALR-Parser eine komplexe *Stackmaschine*, welche Zustandsübergänge anhand von *Stacktransformationen* durchführt.

Im konkreten Eclipse-Java-Compiler wird mit Hilfe dieses Mechanismus ein AST aufgebaut. Dabei wird der AST nicht - angefangen mit dem größten Element - immer weiter verfeinert, sondern der Automat erzeugt aus elementaren AST-Elementen immer komplexere. Als Zwischenspeicher für diese kleineren AST-Elemente dient der `astStack`. Sobald der Automat eine komplexere Struktur erkannt hat, z.B. eine Methode, werden alle auf dem `astStack` liegenden AST-Objekte (Statements, Variablen, ...), welche den eigentlichen Inhalt der Methode ausmachen vom `astStack` geholt (gelesen und entfernt), um daraus ein neues AST-Objekt vom Typ MethodDeclaration zu erzeugen. Dieses neue Methoden-Objekt wird wieder auf dem `astStack` gespeichert, um daraus später eine Klasse (`TypeDeclaration`) erzeugen zu können.

Da beim Anlegen des Stacks sichergestellt werden muss, dass dieser in jedem Fall konsistent bleibt, wird ein eigener Längen Stack verwaltet. Auf diesem Stack wird die Anzahl der auf den Stack gelegten Einträge verwaltet. Dadurch ist es möglich mehrere oder auch kein Element auf den Stack zu legen und diese dann bewusst wieder auszulesen, bzw. als ausgelassen zu erkennen.

Für jeden Stack existiert ein Pointer, welcher auf das aktuelle Top-Element zeigt: `astPtr`, `intStackPtr`, ... Eine Ausnahme bildet hier der `identifierPositionStack`, welcher ebenfalls über den `identifierStackPtr` angesprochen wird.

Nachfolgende Tabelle zeigt die im Parser verwendeten Stacks und die zugehörigen Stack-Pointer

Definition/Name/Pointer	Beschreibung
AstNode[] astStack [astPtr]	Stack für AstNodes. Dieser Stack enthält alle bereits erkannten AstNodes. Während des parsierens wird dieser Stack von den consumeMethoden auf- und auch wieder abgebaut.
int[] astLengthStack [astLengthPtr]	Enthält die Anzahl von AstNodes auf dem Stack, z.B. die Länge einer Liste von einzelnen Argumenten die auf dem Stack liegen.
int[] intStack [intPtr]	Auf den intStack werden die den Token äquivalenten int-Werte abgelegt. Außerdem wird auf diesem Stack alles gespeichert was sonst noch int ist.
char[][] identifierStack [identifierPtr]	Beim Parsen werden in der consumeToken-Methode automatisch alle gelesenen Identifier auf dem identifierStack abgelegt. Diese werden dann von der entsprechenden consume-Methode ausgewertet und wieder vom Stack entfernt.
int[] identifierLengthStack [identifierLengthPtr]	Da Identifier im Parser optional sein können, muss es die Möglichkeit geben, zu erkennen, welches Token ausgelassen wurde. Wenn der Parser an einer Stelle ein leeres Token liest, dann wird auf dem identifierLengthStack eine 0 abgelegt. Falls das optionale Token gelesen werden konnte, wird eine 1 abgelegt. Dadurch ist gewährleistet, dass der identifierStack konsistent behandelt werden kann. Wenn in einer konkreten consume-Methode ein Identifier auf dem identifierStack erwartet wird, dann muss zuerst auf dem identifierLengthStack nachgesehen werden, wieviele Token aktuell gelesen werden müssen.

Definition/Name/Pointer	Beschreibung
long[] identifierPositionStack [identifierPtr]	Auf dem identifierPositionStack wird die Sourcecode-Position des Identifiers (Anfangs- und Endposition) in einem Long-Wert gespeichert. Die oberen 32-Bit stellen die Startposition und die unteren 32-Bit die Endposition dar. Als Pointer dient, wie für den identifierStack, ebenfalls der identifierPtr.
Expression[] expressionStack [expressionPtr]	Expressions werden in einem speziellen expressionStack verwaltet.
int[] expressionLengthStack [expressionLengthPtr]	Enthält die Anzahl von Expressions auf dem Stack, z.B. die Länge einer Liste von einzelnen Parametern die auf dem Stack liegen.

Tabelle 6

Für die Verwendung eines Stacks sind folgende Prozeduren üblich:

```
Push(value)    stack[++ptr]=value
Pop()          return stack[ptr--]
Top()          return stack[ptr]
```

Im Eclipse-Parser werden `Pop` und `Top` aus Performancegründen nicht über Methoden angesprochen, sondern sind inline kodiert. `Push` ist im Parser für sämtliche Stacks implementiert, um die Größe des Stacks dynamisch anpassen zu können.

Consume-Methode

Am Beispiel einer Consume-Methode für eine `MethodInvocationSuper` soll die Verwendung des Stack gezeigt werden. Consume-Methoden werden direkt vom automatisch generierten Teil des Parsers aufgerufen und müssen per Hand implementiert werden. Deshalb ist es wichtig, das generelle Konzept das dahinter steckt zu verstehen.

Zuerst muss herausgefunden werden, welche Informationen auf dem Stack (sofern er richtig aufgebaut wurde) zur Verfügung stehen. Da die Grammatik darüber Aufschluss gibt, wird in der ersten Zeile der vollständige Text der Regel, als Kommentar vermerkt (Zeile 3358). Zu lesen ist dieser Kommentar dann folgendermaßen: Es muss eine `MethodInvocation` erzeugt werden und es stehen dafür auf dem Stack die Informationen super, identifier und ArgumentListopt in umgekehrter Reihenfolge zur Verfügung. Welcher Stack allerdings diese Informationen enthält, kann aus dem Text der Regel nicht erkannt werden. Deshalb ist es wichtig, die unterschiedlichen Stacks und deren Verwendung (siehe Tabelle 6) zu kennen.

Als zweites muss der Stack abgebaut werden. Da die Erzeugung einer Argumentliste innerhalb mehrerer Consume-Methoden Verwendung findet, wird diese Aufgabe an eine Methode (`newMessageSend()`) delegiert. Diese holt sämtliche Argumente vom

Expression-Stack und fügt diese in das Attribut `MessageSend.arguments` (Zeile 7418) einer neuen Instanz eines `MessageSend`-Objektes ein und liefert es zurück. Dann werden innerhalb der consumeMethode die restlichen auf dem Stack verfügbaren Informationen gelesen. Der Identifier wird im Attribut `MessageSend.selector` (Zeile 3364) gespeichert. Die `SuperReference` wird im Attribut `MessageSend.receiver` gespeichert. Wichtig an dieser Stelle ist, dass nur Informationen auf dem Stack liegen, die auch gelesen werden müssen. Das Token super liegt z.B. genauso wie das Token „.“ auf keinem Stack. Denn innerhalb der Consume-Methode ist klar, dass der Parser dieses super bereits gelesen haben muss, denn sonst wäre die Consume-Methode nicht aufgerufen worden.

Als drittes muss das neu erzeugte `MessageSend`-Objekt wieder auf einen Stack geschoben werden, damit weitere Consume-Methoden damit arbeiten können. Hier wird diese Aufgabe an die Methode `pushOnExpressionStack()` (Zeile 3367) delegiert.

Parser.java

```
2801 protected void consumeMethodInvocationSuper() {
2802 // MethodInvocation ::= 'super' '.' 'Identifier' '(' ArgumentListopt ')'
2803
2804    MessageSend m = newMessageSend();
2805    m.sourceStart = intStack[intPtr--];
2806    m.sourceEnd = rParenPos;
2807    m.nameSourcePosition = identifierPositionStack[identifierPtr];
2808    m.selector = identifierStack[identifierPtr--];
2809   identifierLengthPtr--;
2810    m.receiver = new SuperReference(m.sourceStart, endPosition);
2811   pushOnExpressionStack(m);
2812 }
...
3385   // This method is part of an automatic generation : do NOT edit-modify
3386 protected void consumeRule(int act) {
3387    switch ( act ) {
...
4096    case 364 :  System.out.println("MethodInvocation ::= super DOT ...
4097      consumeMethodInvocationSuper();
4098      break ;
...
7408 protected MessageSend newMessageSend() {
7409 // '(' ArgumentListopt ')'
7410
7411    MessageSend m = new MessageSend();
7412    int length;
7413    if ((length = expressionLengthStack[expressionLengthPtr--]) != 0) {
7414      expressionPtr -= length;
7415      System.arraycopy(
7416      expressionStack,
7417      expressionPtr + 1,
7418      m.arguments = new Expression[length],
7419      0,
7420      length);
7421    };
7422    return m;
7423  }
...
```

Beispiel 10

Da der Parser eine eins-zu-eins Umsetzung der Java-Grammatik darstellt, ist es sinnvoll, sich diesen etwas näher im Vergleich zur Grammatik anzuschauen.

Ein kleiner Ausschnitt aus der LALR-Java-Grammatik:	Generierte Debug Ausgabe des Parsers:

...
PackageDeclaration ::= PackageDeclarationName ';'
/.$putCase consumePackageDeclaration(); $break ./
 	
PackageDeclarationName ::= 'package' Name
/.$putCase consumePackageDeclarationName();
$break ./
...
	...
	consumeToken('package')
	  shift
	consumeToken(Identifier(MyPackage1))
	  reduce
	consumePackageDeclarationName()
	  shift/reduce
	consumeToken(';')
	  reduce
	consumePackageDeclaration()
	...

Beispiel 11

Wie arbeitet der Parser, wenn er die Tokensequenz **[package] [MyPackage] [;]** in der Methode `parse()` auswerten soll? Der Parser liest das Token **package** und erhält aus der Action-Goto-Tabelle (`tAction`) eine Shift Anweisung. Danach liest der Parser das Token **MyPackage1** und erkennt, dass es sich um einen Identifier handelt. Diesen legt er für eine spätere Auswertung auf einem IdentifierStack ab. Gleich darauf wird in der Methode `consumePackageDeclaration()` ein `ImportReference` Objekt erzeugt. Jetzt wird endlich das letzte Token „**;**“ für eine `PackageDeclaration` gelesen. Der Parser erhält aus der Tabelle `tAction` die Anweisung `consumeRule` (`consumePackageDeclaration()`) und sollte jetzt eigentlich die neue Importreferenz (`ImportReference`) erzeugen. Da er dies aber schon vorzeitig gemacht hat, wird diese lediglich validiert.
Normalerweise sollte diese `ImportReference` erst erstellt werden, wenn die Methode `consumePackageDeclaration()` aufgerufen wird, aber um überflüssige Objekte (z.B. `ImportReferenceName`), die den AST nur unnötig aufblähen würden, zu vermeiden, wird diese `ImportReference` bereits erzeugt, wenn erste Informationen zur Verfügung stehen.

3.4.3.4 Eclipse Datenstrukturen

AST

Der Eclipse/Java Compiler benötigt während des Kompilierungsvorganges einen AST. Der Wurzelknoten dieses AST, ist in Eclipse die `CompilationUnitDeclaration` und repräsentiert eine Java-Datei in Form einer Baumstruktur. Die Erzeugung dieses AST wird von einem Parser in Zusammenarbeit mit einem Scanner durchgeführt. Während der Scanner aus einer Java-Datei kleine Häppchen - die sogenannten Token - erzeugt und diese dem Parser zur weiteren Verarbeitung zur Verfügung stellt, erzeugt dieser daraus, nach den Regeln einer Java-Grammatik, den AST.

Der sich beim Parsieren eines einfachen Programms ergebende AST (*Abstract-Syntax-Tree*), stellt ein Composite-Objekt [16] dar. Jedes AST-Objekt besteht aus einer Anzahl

weiterer AST-Objekte. Dadurch ist es möglich, eine `CompilationUnitDeclaration` so zu verwenden, als ob es sich um ein einzelnes Objekt, und nicht um eine hierarchische Struktur, handeln würde. Alle Methodenaufrufe an ein AST-Objekt, z.B. an `CompilationUnitDeclaration`, werden gegebenenfalls an seine Unterobjekte weitergereicht. Der AST besteht aus insgesamt 138 Klassen und bildet so die Java-Sprachelemente vollständig ab.
Im folgenden ist die Instanz einer `CompilationUnitDeclaration` eines typischen HelloWorld.java Programms nach dem ersten Compiler-Durchlauf abgebildet. Zu beachten ist, dass die Methodenrümpfe leer sind. Diese werden erst in einer zweiten Compilerphase gefüllt.

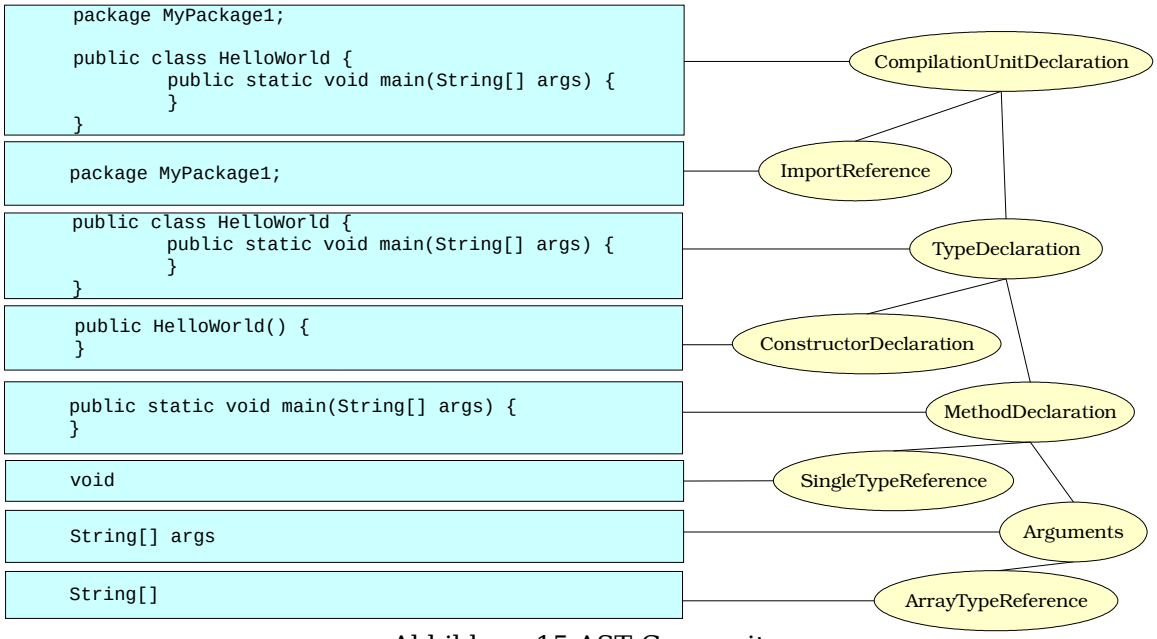


Abbildung 15 AST Composite

Binding

Neben dem AST stellen die Bindings innerhalb des Compilers eine Datenstruktur von zentraler Bedeutung dar. Der Zweck dieser Bindings soll im folgenden Abschnitt erläutert werden.
Eine eindeutige Identität eines Typs wird im Eclipse-Java-Compiler durch sogenannte Bindings repräsentiert. Bindings werden sowohl für Quellcodedateien als auch für Binary-Dateien erzeugt und können daher polymorph verwendet werden. Bindings sind während des Kompilierungsvorganges in einen Bindinggraph eingebettet. Welcher zu bestimmten Phasen des Compilers erzeugt bzw. erweitert wird. Siehe Abbildungen 17 und 18. Bindings enthalten die vollständige Signatur von Klasse (`TypeBinding`), Methode (`MethodBinding`) oder Attribut (`FieldBinding`) jedoch keine Semantik, also weder Statements noch Bytecode.
Während des Kompilierens einer Java-Datei, erzeugt der Parser aus einer `CompilationUnit` eine `CompilationUnitDeclaration` in Form eines Baumes, bestehend aus AstNodes. Innerhalb dieser `CompilationUnitDeclaration` kann es Referenzen auf Typen, z.B. auf Klassen, Methoden oder Felder geben. Da Java streng typisiert ist, sollten Typfehler bereits vom Compiler als Fehler erkannt werden. Um diese Überprüfung durchführen zu können, muss der Compiler den referenzierten Typ mit dem

realen Typ vergleichen. Dazu ist gegebenenfalls ein Laden des angeforderten Typs notwendig. Leider liegen nicht alle Typen als Java-Quellcode vor, sondern viele der referenzierten Typen existieren nur in Form von Binaries. Diese Binaries werden vom Compiler zwar auch geladen, es wird daraus jedoch kein AST erzeugt, sondern ein `BinaryTypeBinding`. Dieses `BinaryTypeBinding` enthält ebenfalls Methoden und Felder in Form von `MethodBindings` und `FieldBindings`. Auf den ersten Blick könnte man meinen, es handle sich bei diesem `BinaryTypeBinding` ebenfalls um eine AST-Struktur. Dies ist jedoch nicht der Fall, da die Bindings in einem nicht zyklenfreien gerichteten Graphen zusammenhängen. Um die Typen einer zu kompilierenden Java-Datei ebenfalls in diesen Graph einhängen zu können, wird aus dem AST, für alle darin deklarierten Typen, ein Binding erzeugt. Aus einer `TypeDeclaration` wird ein `SourceTypeBinding` erzeugt, aus einer `MethodDeclaration` ein `MethodBinding`, usw.
Anhand des Beispiels Nr. 12 soll die Verbindung zwischen AST und Binding dargestellt werden.

Quellcode

```
01  public class B{
02    String _str;
03    B(String str){
04      _str=str;
05    }
06  }
```

Beispiel 12

Abbildung 16 stellt den AST dar, nachdem die Methode `Compiler.beginToCompile()` für das obige Beispiel abgeschlossen ist. Die Bedeutungen der Abkürzungen sind aus Tabelle 7 zu entnehmen. Eine vollständige Liste aller im Compiler verwendeten Bindings ist im Package `org.eclipse.jdt.internal.compiler.lookup` zu finden.

Abkürzung	Vollständiger Name
CUD	CompilationUnitDeclaration
TD	TypeDeclaration
CD	ConstructorDeclaration
FD	FieldDeclaration
ARG	Argument
ASS	Assignment
NR	NameReference
TR	TypeReference

Abkürzung	Vollständiger Name
STB	SourceTypeBinding
BTB	BinaryTypeBinding
MB	MethodBinding
FB	FieldBinding
VB	VariableBinding

Tabelle 7

Der AST ist bis auf Signaturebene vollständig aufgebaut. `Statements` sind noch nicht im AST enthalten. In dieser Phase werden die eigenen Bindings erzeugt und bilden sozusagen einen Schatten des AST. Die `TypeDeclaration` erhält ein `SourceTypeBinding`, die `ConstructorDeclaration` erhält ein `MethodBinding` und die `FieldDeclaration` erhält ein `FieldBinding`. Durch den blauen Pfeil zwischen

AST und Binding soll visualisiert werden, dass im `AstNode` ein Attribut existiert, welches eine Referenz auf das Binding hält.

Compiler.beginToCompile	AST	Binding

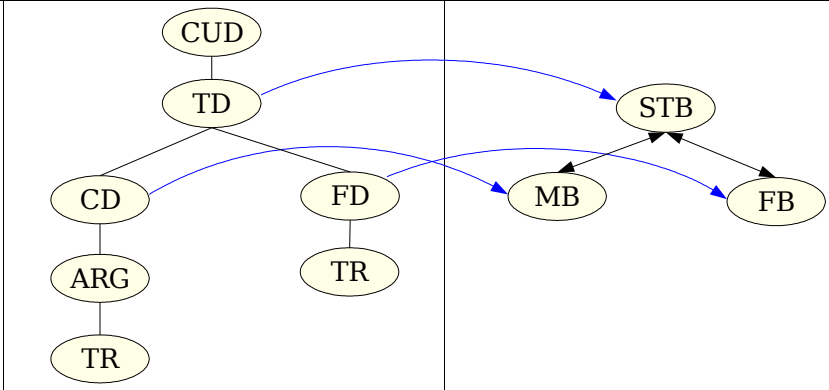
`public class B {`
`  String str;`
`  B(String string) {`
`  }`
`}`
		

Abbildung 16 Eigene Bindings

Aber die Erzeugung eines Graphen des Selbstzweckes wegen, macht keinen Sinn. Der Vorteil liegt darin, dass `TypeReferences` oder `NameReferences` im AST mit Hilfe von Bindings eindeutig aufgelöst werden können. Bindings enthalten den Fully-Qualified-Name des Typs. Im Gegensatz dazu enthalten `TypeReferences` oder `NameReferences` oft nur einen einfachen oftmals mehrdeutigen Namen (z.B. „arg“, oder „Data“);

In Abbildung 17 werden alle `TypeReferences` und `NameReferences` aufgelöst.

Compiler.process	AST	Binding

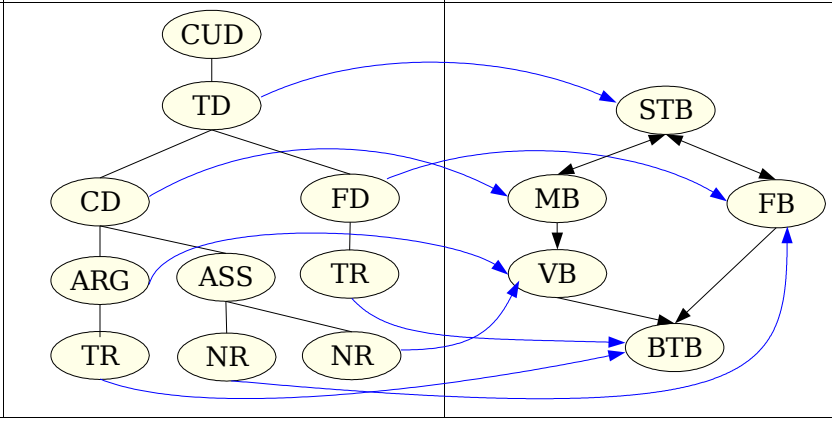
`public class B {`
`  String str;`
`  B(String string) {`
`  super();`
`  str = string;`
`  }`
`}`
		

Abbildung 17 Bindinggraph

Um das Diagramm in Abbildung 17 übersichtlicher zu halten, werden nicht alle Referenzen auf Bindings dargestellt. Ein Assignment hält noch eine weitere Referenz auf das `BinaryTypeBinding`, nämlich den Typ, welchen die Zuweisung selber zurückliefern würde. Das `BinaryTypeBinding` selber hält natürlich ebenfalls noch Referenzen auf die darin enhaltenen Methoden und Felder in Form von Bindings.

Scope

Ein Scope stellt Suchfunktionaliät innerhalb eines definierten Sichtbarkeitsbereiches zur Verfügung. Eine Scope-Instanz stellt eine verschachtelte Struktur dar. Ein `MethodScope` definiert die Sichtbarkeit z.B. von lokalen Variablen innerhalb einer Methode. Dieser `MethodScope` ist in ein `ClassScope`-Object eingebettet. Ein `ClassScope` kennt sämtliche Methoden und Attribute einer Klasse. Da jeder Scope seinen umschließenden Scope kennt, kann eine Suche leicht durch alle Scopes bis hin zum `CompilationUnitScope` ausgeführt werden. Außerdem stellt ein Scope Funktionalität zur Verfügung, um Bindings (z.B. für Methoden) zu erzeugen. Die Scopes selber werden in keiner speziellen Phase erzeugt, sondern unmittelbar bevor ein Binding erzeugt wird.

LookupEnvironment

Das `LookupEnvironment` stellt Suchfunktionalität für Typen (Klassen) zur Verfügung. Angeforderte Typen können entweder aus einem Cache geholt, oder falls diese noch nicht gecached sind, direkt aus dem Dateisystem erzeugt werden.

3.4.4 Phasenmodell des Eclipse-Java-Compilers

Um ObjectTeams spezifische Erweiterungen in den vorhandenen Eclipse-Java-Compiler zu integrieren, ist es notwendig, dessen Funktionsweise genaustens zu analysieren. Von besonderer Bedeutung sind für diese Analyse der AST (siehe 3.4.3.4), welcher vom Parser (siehe 3.4.3.3) erzeugt wird und die unterschiedlichen Operationen, die auf diesem AST, im Laufe des Kompilierungsvorganges, ausgeführt werden.
Ein Java-Compiler erzeugt aus Java-Quellcode, JVM-Bytecode. Übertragen auf ein Phasenmodell würde das bedeuten: Phase 1 beinhaltet das Einlesen des Java-Quellcodes und Phase 2 beinhaltet das Erzeugen des JVM-Bytecodes. Die Reihenfolge der unterschiedlichen Phasen ergibt sich dabei aus den Abhängigkeiten der Phasen untereinander. Neben diesen beiden offensichtlichen Phasen gibt es jedoch noch einige nicht intuitiv erschließbare Phasen. Um das vollständige Phasenmodell zu erstellen und um die Vorgänge innerhalb dieser Phasen zu verstehen, war es deshalb notwendig, neben einer statischen Quellcodeanalyse des Compilers, auch eine Analyse der instanziierten Objekte zur Laufzeit des Compilers durchzuführen. Aus der statischen Analyse ergab sich, an welchen Stellen innerhalb des Kontrollflusses interessante Informationen zu erwarten sind und aus der Analyse zur Laufzeit ergab sich ein konkretes Bild der jeweils erreichten Zustände. Das erste Resultat der Analyse war, dass der Zustand des AST und der Zustand der Bindings dabei eine zentrale Rolle spielen. Diese wurden deshalb während eines kompletten Kompilierungsvorganges beobachtet und sämtliche Änderungen wurden auf Relevanz, bezüglich des zu erzeugenden Phasenmodells, hin untersucht und notiert. In einem weiteren analytischen Schritt wurden dann die notierten Zustände konkreten Phasen zugeordnet. Um ein Phasenmodell zu erstellen, wäre diese Analyse zur Laufzeit zwar nicht unbedingt notwendig gewesen; sinnvoll aber schon deswegen, um die Zusammenhänge und Abhängigkeiten der Zustände zwischen den unterschiedlichen Phasen zu verstehen.

Das sich aus dieser Analyse des Eclipse-Java-Compilers ergebende Phasenmodell ist in Abbildung 18 zu sehen.

3.4.4.1 Phasenmodell

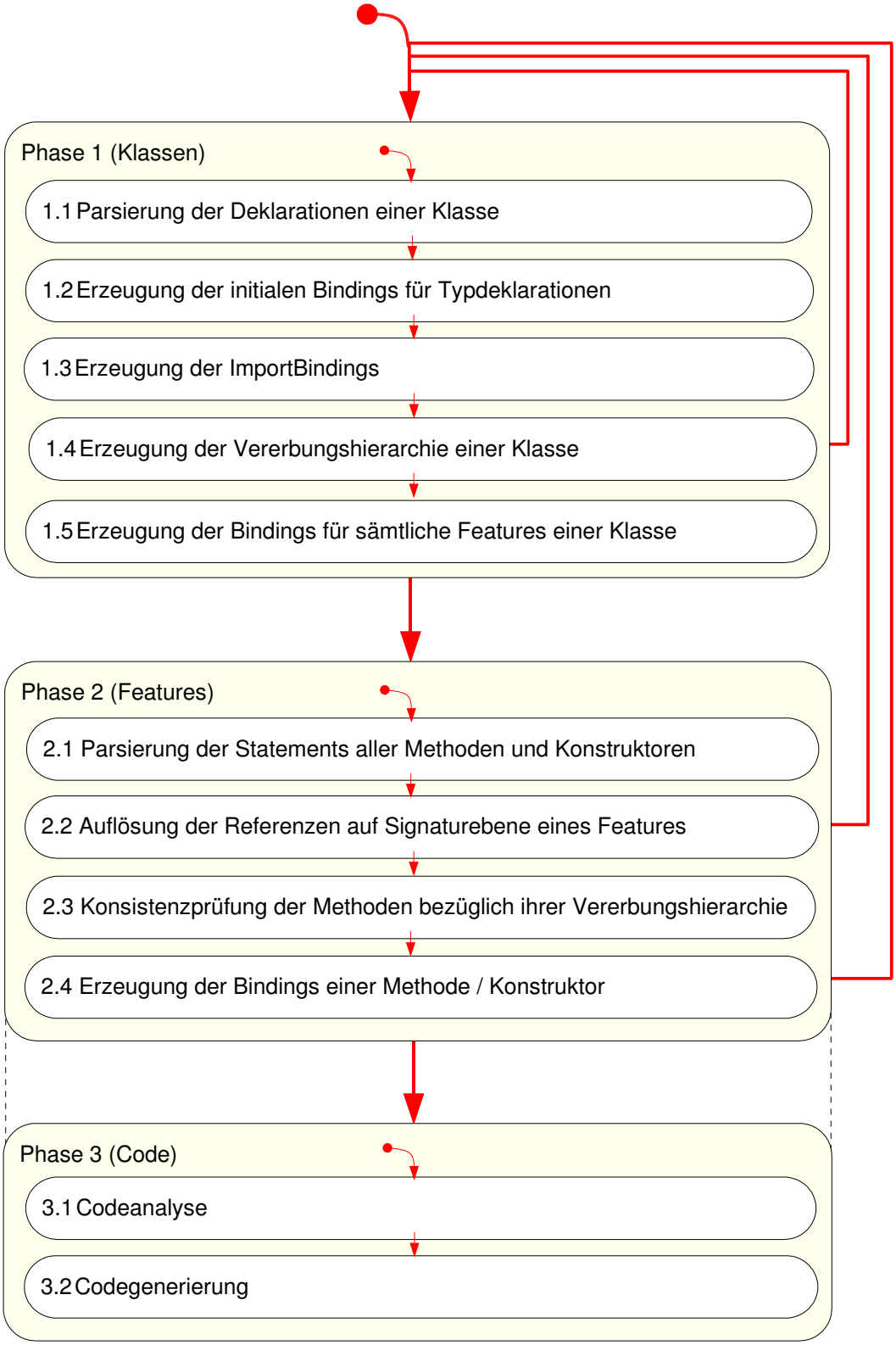


Abbildung 18 Compiler Phasenmodell

Das Phasenmodell besteht aus 3 übergeordneten und 11 untergeordneten Phasen, die in der abgebildeten Reihenfolge durchlaufen werden müssen. Die Methode `compiler.Compiler.beginToCompile()` wird vom Compiler für alle dem Compiler initial bekannten `CompilationUnits` aufgerufen und stößt für jede dieser `CompilationUnits` Phase 1 an. Wenn im Laufe des Kompilierungsprozesses weitere Java-Dateien kompiliert werden müssen, z.B. beim Auflösen der Referenzen auf die Superklasse, wird die Methode `compiler.Compiler.accept()` aufgerufen. Diese Methode accept() stößt dann für genau eine `CompilationUnit` (z.B. für die Superklasse) Phase 1 an.
Die in der Abbildung 18 einzeln dargestellten Phasen 2 und 3 sind im Gegensatz zu Phase 1 keine wirklich trennbaren Phasen. Beide werden im selben untrennbaren Kontrollfluss innerhalb der Methode `compiler.Compiler.process()` durchlaufen. Eine logische Trennung ist trotzdem insofern sinnvoll, da zum einen mit Phase 2 die Attributierung des AST abgeschlossen ist und zum anderen innerhalb der Phase 3 die Generierung des Bytecodes behandelt wird.

3.4.4.2 Phasenablauf

In der folgenden Abbildung 19, soll der Ablauf zwischen den Phasen, innerhalb des Eclipse-Java-Compilers, für das Kompilieren mehrerer `CompilationUnits`, gezeigt werden.

A, B, C, D seien `CompilationUnits` (Java-Dateien)
A und B seien dem Compiler direkt übergeben worden
C sei eine Superklasse von B
D sei eine innerhalb eines Methodenbodies von C per qualified-name referenzierte Klasse

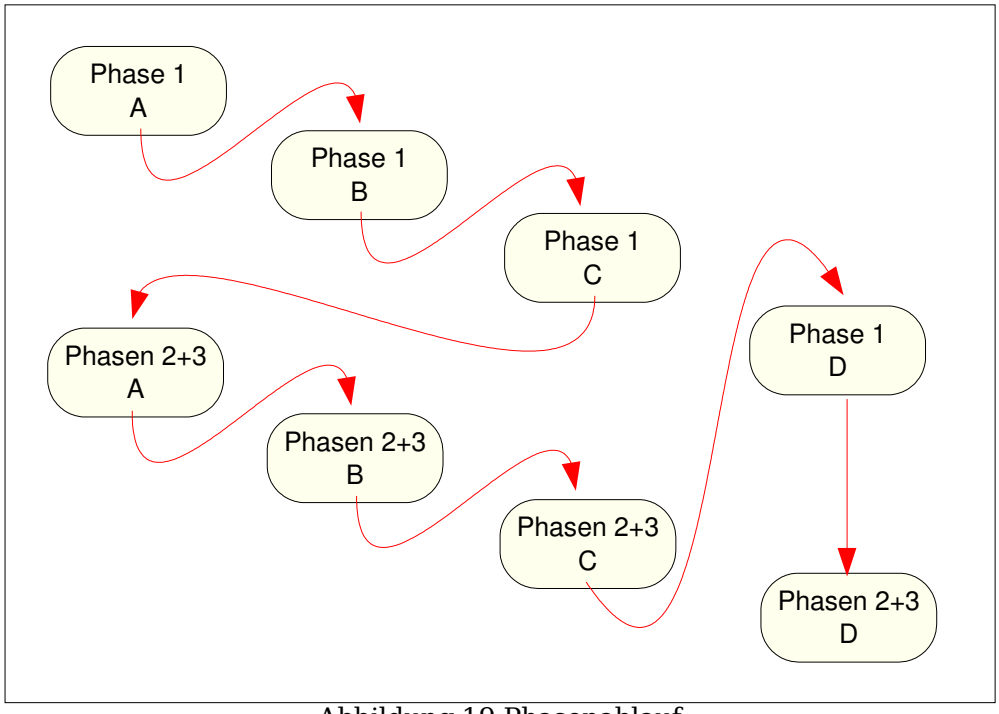


Abbildung 19 Phasenablauf

Die CompilationUnits A und B werden dem Compiler direkt übergeben und müssen deshalb in jedem Fall kompiliert werden. Für A und B wird deshalb Phase 1 sukzessiv angestoßen (beginToCompile(A,B)). Innerhalb der Hierarchieerzeugung von B (Phase 1.4) wird für die Superklasse von B ebenfalls Phase 1 angestoßen (accept(C)). Für A, B und C wird jetzt im weiteren Verlauf des Kompilierungsprozesses jeweils Phase 2+3 (process(A), process(B), process(C)) angestoßen. Während des gesamten Kompilierungsprozesses können zu jedem Zeitpunkt und aus jeder Phase heraus weitere CompilationUnits mit dem Anstoßen von Phase 1 nachgeladen werden. Java-Source-Dateien werden anschließend in eine Queue eingehängt und durchlaufen zu einem späteren Zeitpunkt Phase 2+3. So wie z.B. die in Phase 2+3 von C referenzierte CompilationUnit D. Phase 1 kann dabei sowohl von Java-Source-Dateien als auch von Java-Binary-Dateien durchlaufen werden.

3.4.4.3 Beschreibung der Phasen

Phase 1 (Klassen)

Der Fokus dieser Phase 1 ist die Klasse und deren Signatur. Ziel dieser Phase ist das möglichst effiziente Erzeugen von Bindings (TypeBinding, MethodBinding, FieldBinding). Für die Erzeugung dieser Bindings ist lediglich die Signatur einer Klasse bzw. des Features relevant. Der AST wird deshalb auch nur bis auf Signaturebene erzeugt.

beteiligte Methoden:
- compiler.Compiler.beginToCompile() || compiler.Compiler.accept()

Phase 1.1 Parsierung der Deklarationen einer Klasse

Der Parser erzeugt aus einer CompilationUnit eine CompilationUnitDeclaration. Dieser initial erzeugte AST enthält alle Deklarationen, die auf Signaturebene der Klasse sichtbar sind: TypeDeclaration, MemberTypeDeclaration, MethodDeclaration, FieldDeclaration, ConstructorDeclaration und Argument. Sämtliche innerhalb dieser Deklarationen notwendigen Typ-Referenzen, sind ebenfalls im AST verfügbar. So enthält eine CompilationUnitDeclaration Import-Referenzen (ImportReference) für das eigene Package und für importierte Packages. Klassen (TypeDeclaration, MemberTypeDeclaration) enthalten Referenzen (TypeReference) auf Superklasse und Superinterfaces. Konstruktoren (ConstructorDeclaration) enthalten Referenzen auf Typen innerhalb ihrer Argumente und ihrer Exceptions. Methoden haben zusätzlich noch einen Rückgabetyp für den eine Referenz existieren muss. Und natürlich wird auch der Typ eines Feldes über eine Typ-Referenz im AST definiert. Nicht im AST enthalten, sind die zu einer Methode oder Konstruktor gehörenden Statements, also der eigentliche Programmcode.

beteiligte Methoden:
- compiler.Compiler.beginToCompile() || compiler.Compiler.accept()
- compiler.parser.Parser.parse()

Phase 1.2 Erzeugung der initialen Bindings für Typdeklarationen

Ein Binding (siehe 3.4.3.4) stellt im Compiler die eindeutige Repräsentation einer Deklaration dar. Die ersten Bindings die im Kompilierungsprozess erzeugt werden, sind die sogenannten `TypeBindings`. `TypeBindings` enthalten z.B. den vollständigen qualifizierenden Namen (Fully-Qualified-Name) einer Klasse. Features (Methoden, Konstruktoren, Felder) sind noch nicht mit Bindings versehen und demzufolge auch noch nicht in das `TypeBinding` eingebettet.

beteiligte Methoden:
- `compiler.Compiler.beginToCompile() || compiler.Compiler.accept()`
- `compiler.lookup.LookupEnvironment.buildTypeBindings()`

Phase 1.3 Erzeugung der ImportBindings

Um Referenzen auf Typen auflösen zu können, muss eine Referenz entweder den vollständigen qualifizierenden Namen enthalten, oder es muss durch eine aufgelöste Importanweisung (`ImportBinding`) genau definiert werden, in welchen Klassen oder Packages diese Referenz gesucht werden soll.

beteiligte Methoden:
- `compiler.Compiler.beginToCompile() || compiler.Compiler.accept()`
- `compiler.lookup.LookupEnvironment.completeTypeBindings()`
- `compiler.lookup.CompilationUnitScope.checkAndSetImports()`

Phase 1.4 Erzeugung der Vererbungshierarchie einer Klasse

In dieser Phase wird die Vererbungshierarchie einer Klasse aufgebaut. Referenzen auf Superklasse und Superinterfaces werden aufgelöst und mit den Bindings der gefundenen Typen angereichert. Sollte ein Binding für eine Superklasse oder ein Superinterface noch nicht im Scope (einer Art Cache) existieren, muss dieses Binding erzeugt werden. Dafür wird Phase 1 für die noch nicht aufgelöste Referenz - also den gesuchten Typen - rekursiv angestoßen. Fehlerhafte Referenzen oder nicht auflösbare Typen werden mit einem `ProblemBinding` attributiert und später in Form einer Warnung/Fehlermeldung an den Programmierer ausgegeben.

beteiligte Methoden:
- `compiler.Compiler.beginToCompile() || compiler.Compiler.accept()`
- `compiler.lookup.LookupEnvironment.completeTypeBindings()`
- `compiler.lookup.CompilationUnitScope.connectTypeHierarchie()`

Phase 1.5 Erzeugung der Bindings für sämtliche Features einer Klasse

Als letzten Schritt innerhalb der ersten Phase werden sämtliche im AST enthaltenden Inneren-Klassen, Methoden, Konstruktoren und Felder mit ihren eigenen Bindings attributiert. Für eine Innere-Klasse wird ein `TypeBinding` erstellt, für eine Methode oder Konstruktor ein `MethodBinding` und für ein Feld ein `FieldBinding`. `MethodBindings` enthalten eine Referenz auf ihr umschliessendes `TypeBinding`.

Interessant ist, dass die Typen innerhalb der Signatur einer Methode (Argumenttypen, Rückgabetyp, Exceptiontypen) in dieser Phase noch nicht mit Bindings attributiert werden und demzufolge auch nicht in den Bindinggraphen eingefügt werden können. Dies bedeutet, dass mit Bindings attributierte Referenzen auf Methoden, später nicht ausschließlich anhand der Bindings eindeutig aufgelöst werden können, da die Signatur dem Binding unbekannt ist. Welches Binding soll beim Auflösen einer Referenz gewählt werden, wenn es mehrere überladene Methoden gibt? Hier kommt ein anderer Mechanismus zum Tragen: Bindings für Signaturen werden „on-demand“ angefordert und erst dann in den Binding-Graphen eingefügt.

beteiligte Methoden:
- `compiler.Compiler.beginToCompile() || compiler.Compiler.accept()`
- `compiler.lookup.LookupEnvironment.completeTypeBindings()`
- `compiler.lookup.CompilationUnitScope.buildFieldsAndMethods()`

Phase 2 (Features)

Der Fokus dieser Phase 2 ist nicht mehr wie in Phase 1 auf die Klasse, sondern auf die in dieser Klasse enthaltenen Features (Methoden, Konstruktoren, Felder) gerichtet. Jedes Feature wird bezüglich seines Inhaltes untersucht und mit entsprechenden Bindings für z.B. lokale Variablen versehen. Ziel dieser Phase 2 ist ein vollständig mit Bindings attributierter AST.

beteiligte Methoden:
- `compiler.Compiler.process()`

Phase 2.1 Parsierung der Statements aller Methoden und Konstruktoren

Nachdem vom Parser in Phase 1.1 lediglich die Signatur einer Klasse in Form von Deklarationen erzeugt wurde, wird in dieser Phase der AST um den noch fehlenden Programmcode (Statements) vervollständigt. Dafür wird die `CompilationUnit` erneut mit dem Fokus auf Methoden- und Konstruktor-Rümpfe parsiert. Die vom Parser erzeugten Statements werden in die vorhandenen Methoden eingefügt. Der AST ist nach dieser Phase 2.1 bezüglich seiner `AstNodes` vollständig aufgebaut.

beteiligte Methoden:
- `compiler.Compiler.process()`
- `compiler.Compiler.getMethodBodies()`

Phase 2.2 Auflösung der Referenzen auf Signaturebene eines Features

Für alle Typreferenzen innerhalb der Signatur von Elementen einer Klasse, werden die aufgelösten Bindings in das Binding des Features (`MethodBinding`, `FieldBinding`) eingehängt. Die Signatur beinhaltet eine Referenz auf den Rückgabetyp, eine Liste von Referenzen auf Argumenttypen und eine Liste von Referenzen auf Exceptiontypen. Anders als erwartet erfolgt keine Attributierung der aufgelösten Referenz. Das heißt, das

Binding wird nicht in die Typreferenz oder in das Argument eingetragen, sondern vorerst lediglich in den Bindinggraph eingehängt.

beteiligte Methoden:
- `compiler.Compiler.process()`
- `compiler.lookup.CompilationUnitScope.faultInTypes()`

Phase 2.3 Konsistenzprüfung der Methoden bezüglich ihrer Vererbungshierarchie

Für alle im AST enthaltenen Methoden wird eine Konsistenzüberprüfung entlang der gesamten Vererbungshierarchie (bis Supertyp `Object`) druchgeführt. Es wird zum einen geprüft, ob alle geerbten abstrakten Methoden implementiert wurden und zum anderen, ob die Signaturen aller überschriebenen und implementierten Methoden übereinstimmen.

beteiligte Methoden:
- `compiler.Compiler.process()`
- `compiler.lookup.CompilationUnitScope.verifyMethods()`

Phase 2.4 Erzeugung der Bindings einer Methode / Konstruktor

Alle Typreferenzen auf Signaturebene im AST werden mit in Phase 2.2 gefundenen Bindings attributiert. Außerdem werden für lokale Deklarationen (`LocalDeclarations`) innerhalb der Methoden Bindings erzeugt.

beteiligte Methoden:
- `compiler.Compiler.process()`
- `compiler.lookup.CompilationUnitScope.resolve()`

Phase 3 (Code)

Der Fokus der Phase 3 liegt auf Ebene des Bytecodes. Phase 3 ist zwar im Kontrollfluss des Java-Compilers nicht von Phase 2 zu trennen, beinhaltet jedoch Funktionalität (z.B. Kontrollflussanalyse), die sich von den ersten beiden Phasen unterscheidet und daher auch logisch zu trennen ist. Ziel dieser Phase ist die Generierung von Bytecode.

beteiligte Methoden:
- `compiler.Compiler.process()`

Phase 3.1 Codeanalyse

Bevor der Bytecode generiert werden kann, muss der in Form von `AstNodes` vorliegende Code noch auf eventuelle Fehler analysiert werden. Jedes Statement überprüft in seiner Ebene den Programmcode auf Fehler und ruft für die in seiner Composite-Struktur [16] enthaltenden Statements, ebenfalls `analyseCode()` rekursiv auf. Fehler sind unter anderem nicht erreichbarer Code, unbehandelte Exceptions und nicht initialisierte Variablen. Zum Beispiel werden Informationen über Zuweisungen oder Mehrfachinitialisierungen von final-Variablen in einem `FlowInfo`

(`UnconditionalFlowInfo`) gespeichert. Dieses `FlowInfo` speichert innerhalb eines Bitfeldes, ob eine Variable bereits initialisiert wurde oder nicht. Innerhalb der Flussanalyse wird dieses `FlowInfo` in Verbindung mit dem Binding ausgewertet.

beteiligte Methoden:
- `compiler.Compiler.process()`
- `compiler.ast.CompilationUnitDeclaration.analyseCode()`

Phase 3.2 Codegenerierung

In dieser letzten Phase wird der Bytecode der zu kompilierenden Klasse erzeugt. Jedes Statement trägt seine Informationen selber in das als Parameter übergebene `ClassFile` ein und ruft für die in seiner Composite-Struktur enthaltenden Statements, ebenfalls `generateCode()` rekursiv auf. Für die Codegenerierung sind sowohl AST als auch Binding erforderlich. Aus Bindings wird als Ergebnis dieser Phase der Constant-Pool erzeugt und aus den Statements wird der eigentliche Bytecode inklusive der Referenzen auf den Constant-Pool generiert. Beim späteren (nicht in dieser Phase stattfindenden) Speichern des ClassFiles in seine zugehörige .class-Datei, wird die komplette auszugebende Bytefolge aus der Konkatenation von Constant-Pool und Bytecode gebildet.

beteiligte Methoden:
- `compiler.Compiler.process()`
- `compiler.ast.CompilationUnitDeclaration.generateCode()`

4 Entwurf

Ausgehend von den in der Analyse erarbeiteten Anforderungen an den zu realisierenden ObjectTeams Compiler, wurde ein informeller Entwurf erstellt. In diesem wurden Strategien erarbeitet, wie sich die notwendigen ObjectTeams Sprachfeatures unter Berücksichtigung der gegebenen Voraussetzungen umsetzen lassen. Ein wichtiges Mittel zur Erarbeitung dieses Entwurfes war die Simulation der zu realisierenden Konzepte. Ein reines ObjectTeams-Programm wurde solange verändert und erweitert, bis die gewünschte ObjectTeams-Funktionalität mittels reinem Java simuliert werden konnte. Der Vergleich zwischen dem originalen und dem auf diese Weise veränderten Quellcode, enthielt alle notwendigen vom Compiler durchzuführenden Transformationen.

4.1 Implicit-Inheritance

Dieser Abschnitt stellt eine mögliche Realisierung für das in Abschnitt 3.3.1 vorgestellte Konzept von Implicit-Inheritance vor.

Um Implicit-Inheritance auf Ebene des Compilers zu realisieren, muss eine Möglichkeit gefunden werden diese zu simulieren. Das bedeutet, dass der ObjectTeams/Java Programmierer keinen Unterschied zwischen einer simulierten und einer echten Vererbungsbeziehung wahrnehmen soll und dass der vom Compiler erzeugte Bytecode von einer normalen JVM ausgeführt werden können muss.

In ObjectTeams/Java wird dies durch einen speziellen Mechanismus realisiert, welcher im Folgenden als Copy-Inheritance bezeichnet wird. Die bei Rollen-Klassen durch Copy-Inheritance realisierte Duale-Vererbung kann als Sonderform der Mehrfachvererbung angesehen werden, bei der eine explizite und eine implizite Vererbungsbeziehung erlaubt ist.

4.1.1 Copy-Inheritance in ObjectTeams/Java

Der Trick in ObjectTeams/Java ist, dass die implizite Vererbungsbeziehung (in der Abbildung 4 (Seite 17) rot gekennzeichnet) durch Copy-Inheritance simuliert wird. Für die Durchführung dieses Tricks werden vom Prinzip her während des Kompilierungsprozesses sämtliche Attribute und Methoden einer Superklasse z.B. von **`T1.R1`** in dessen Subklasse **`T2.R2`** kopiert.

Beispiele 13 und 14 stellen eine mögliche Implementierung des Beispiels aus Abbildung 4 dar. Ausgehend von diesem Beispiel sollen die Besonderheiten und das Konzept von Implicit-Inheritance verdeutlicht werden.

T1.java

```
01  public team class T1 {
02    public class R1 {
03      public static final String name = "T";
04      public String getName() {
05        return name;
06      }
07      public String getVersion(){
08      return "1";
09      }
10    }
11    public class R2 extends R1{
12      public void out(){
13        System.out.println(getName()+getVersion());
14      }
15    }
16    public R2 r2;
17    public T1() {
18      r2 = new R2();
19    }
20  }
```

Beispiel 13

T2.java

```
01  public team class T2 extends T1{
02    public class R1 {
03      public String getVersion(){
04        return "2";
05      }
06    }
07    public class R2 extends R1{
08      public void out(){
09        System.out.println(getName()+getVersion());
10      }
11    }
12    public T2() {
13      super();
14      r2.out();
15    }
16  }
```

Beispiel 14

Die Rolle `T1.R2` inklusive der Methode `T1.R2.out()` ist nur aus pragmatischen Gründen mit gleichem Inhalt in `T2` überschrieben. Dadurch soll gezeigt werden, dass die in T1.R1 definierte Methode `getName()` in `T2.R2.out()` aufgrund expliziter und impliziter Vererbung verfügbar ist. Wäre die Rolle `R2` in `T2` nicht definiert, würde sie aufgrund Implicit-Inheritance so verfügbar sein, als wäre sie, wie im Beispiel 14, manuell in `T2` überschrieben.

4.1.2 Dynamisches Binden von Rollenklassen

Dass ein einfaches Kopieren zur Realisierung von Copy-Inheritance allein jedoch noch nicht ausreicht, soll am folgenden Beispiel 15 demonstriert werden. In diesem Beispiel sei

das Attribut `name` und die Methode `getName()` per Copy-Inheritance automatisch durch den Compiler von `T1.R1` nach `T2.R1` kopiert worden. Der sich aus der einfachen Transformation ergebende Code würde dann wie folgt aussehen:

T2 transformiert

```
01  public team class T2 extends T1{
02    public class R1 {
03      public  static final String name = "T";
04      public String getName() {
05        return name;
06      }
07      public String getVersion(){
08        return "2";
09      }
10    }
11    public class R2 extends R1{
12      public void out(){
13        System.out.println(getName()+getVersion());
14      }
15    }
16    public T2() {
17      super();
18      r2.out();
19    }
20  }
```

Beispiel 15

Nach einem Aufruf von **`new T2()`** gibt das Programm folgendes aus:

```
T1
```

Das für ObjectTeams/Java gewünschte Ergebnis wäre jedoch:

```
T2
```

Das Problem beruht auf der Tatsache, dass in Java das dynamische Binden[1] der Inneren Klassen nicht funktioniert. Gewünscht wäre bei einer Instanziierung von `R2` innerhalb einer Objektinstanz vom Typ `T2`, eine Instanziierung der zugehörigen inneren Klasse `T2.R2`. Erreicht werden kann diese dynamische Instanziierung durch das Verwenden von Instanziierungsmethoden. Denn Methoden werden in Java dynamisch gebunden.

Um das Konzept der Instanziierungsmethoden anzuwenden, müssen im Programmcode sämtliche Konstruktoraufrufe `new MyClass(args)` durch die Instanziierungsmethode z.B. `createMyClass(args)` ersetzt werden. Die Implementierung der Methode macht nichts anderes, als ein Objekt der lokal definierten Klasse zu erzeugen und als Rückgabewert zurückzuliefern. Problematisch ist dabei, dass der Rückgabewert einer überschriebenen Methode vom gleichen Typ sein muss. Dafür reicht es nicht aus, dass die Typen den selben Namen haben, sondern der Fully-Qualified-Name muss identisch

1 *Dynamisches Binden ist ein Konzept, bei dem zur Laufzeit in Abhängigkeit vom Typ einer Objektinstanz, eine konkrete Methode, aus der zum Objekt zugehörigen Vererbungshierarchie, ausgewählt wird. Generell ist dies die spezialisierteste Überschreibung einer Methode innerhalb der Super-Vererbungshierarchie bezüglich des Types der Objektinstanz. In ObjectTeams/Java ist dieses dynamische Binden auf spezielle innere Klassen (Rollen) adaptiert worden.*

sein. Das Problem der notwendigen Typgleichheit der überschriebenen Instanziierungsmethoden ist von großer Bedeutung für die Realisierung von Implicit-Inheritance. Der folgende Abschnitt geht deswegen ausgiebig auf das für die Realisierung von Implicit-Inheritance notwendige Konzept ein.

4.1.3 Typkonformität von Rollenklassen

ObjectTeams/Java erzeugt diese Typkonformität durch eine Abbildung der impliziten Rollenklassen-Vererbungshierarchie auf eine übergeordnete Interface-Vererbungshierarchie[2]. Die Erzeugung dieser Hierarchie findet während des Kompilierungsvorganges statt und ist für den ObjectTeams/Java-Programmierer von außen nicht sichtbar.

In Abbildung 20 ist diese vom Compiler durchzuführende Transformation des ObjectTeams/Java-Programms aus Abbildung 4, als Klassendiagramm zu sehen.

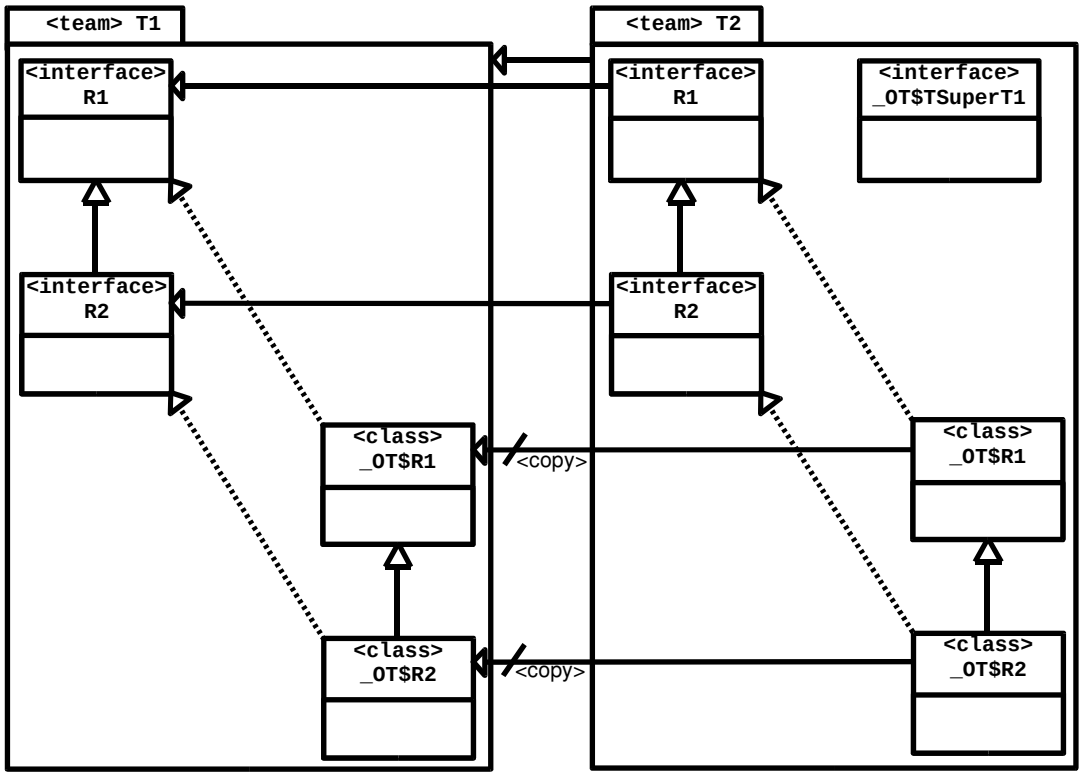


Abbildung 20 Copy-Inheritance

Jede Rolle wird vom Compiler in ein Interface und eine dieses Interface implementierende Klasse aufgesplittet. Die implizite Vererbungsbeziehung zwischen Rollen unterschiedlicher Teams wird zum einen über Interface-Vererbung und zum anderen durch das Kopieren von Methoden und Attributen von z.B. `T1._OT$R1` nach `T2._OT$R1` hergestellt (hier als durchgestrichene Vererbungsbeziehung mit der Bezeichnung <copy> dargestellt). Auf Team-Ebene wird für jeden Rollenkonstruktor eine spezielle Instanziierungsmethode eingefügt. Außerdem werden alle Instanziierungsaufrufe (`new Role()`) im Programmcode, durch einen Aufruf der entsprechenden Instanziierungsmethode ersetzt.

2 Interfaces unterliegen in Java nicht der Einschränkung bezüglich Mehrfachvererbung.

Das Interface erhält Deklarationen aller Methoden aus der Rolle die nicht den Modifier `private` haben. Außerdem werden in das Interface alle in der Rolle als `static final` deklarierten Attribute eingetragen. Der Klassen-Teil erhält dagegen alle Konstruktoren und implementierten Methoden und alle Attribute die nicht bereits in das Interface verschoben wurden. Sobald eine Rolle aufgesplittet ist, werden sämtliche Methoden und Attribute aus der Super-Rolle kopiert. Dabei müssen bestimmte Erweiterungen an der Signatur von Methoden und Konstruktoren vorgenommen werden, um die Vererbungbeziehungen zwischen den generalisierten und spezialisierten Methoden zu erhalten. Wie diese Transformation im einzelnen konkret aussieht, soll am folgenden Beispiel verdeutlicht werden.

Die Beispiele 16 und 17 zeigen die in ObjectTeams/Java durchgeführte Transformation von Beispiel 13 und 14.

Transformation von T1

```
01  public team class T1 {
02    public interface R1 {
03      public static final String name = "T";
04      public String getName();
05      public String getVersion();
06    }
07    public class _OT$R1 implements T1.R1{
08      public String getName() {
09        return name;
10      }
11      public String getVersion(){
12        return "1";
13      }
14    }
15    public interface R2 extends T1.R1{
16      public void out();
17    }
18    public class _OT$R2 extends T1._OT$R1 implements T1.R2{
19      public void out(){
20        System.out.println(getName()+getVersion());
21      }
22    }
23    public R2 r2;
24    public T1() {
25      r2 = _OT$createR2();
26    }
27    public T1.R1 _OT$createR1(){
28      return (T1.R1) new _OT$R1();
29    }
30    public T1.R2 _OT$createR2(){
31      return (T1.R2) new _OT$R2();
32    }
33  }
```

Beispiel 16

Transformation von T2

```
01    public team class T2 extends T1{
02    public interface TSuper_OT$T1 {
03    }
04    public interface R1 extends T1.R1{
05      public String getVersion();
06    }
07    public class _OT$R1 implements T2.R1 {
08      public String getName() {
09        return name;
10      }
11      public String getVersion(TSuper_OT$T1 _OT$marker){
12        return "1";
13      }
14      public String getVersion(){
15        return "2";
16      }
17    }
18    public interface R2 extends T1.R2, T2.R1{
19      public void out();
20    }
21    public class _OT$R2 extends T2._OT$R1 implements T2.R2{
22      public void out(TSuper_OT$T1 _OT$marker){
23        System.out.println(getName()+getVersion());
24      }
25      public void out(){
26        System.out.println(getName()+getVersion());
27      }
28    }
29    public T2() {
30      super();
31      r2.out();
32    }
33    public T1.R1 _OT$createR1(){
34      return (T1.R1) new _OT$R1();
35    }
36    public T1.R2 _OT$createR2(){
37      return (T1.R2) new _OT$R2();
38    }
39  }
```

Beispiel 17

Das für ObjectTeams/Java gewünschte Ergebnis[3] ist jetzt korrekt:

T2

Im Beispiel 17 wurde ein wichtiger Vererbungsmechanismus außen vorgelassen. Ähnlich einer `super`-Referenz soll es in der impliziten Vererbung möglich sein, auf eine Methode/Konstruktor der Super-Rolle zu referenzieren. In ObjectTeams/Java wurde für diesen Zweck ein neues Schlüsselwort (`tsuper`) eingeführt. Im Falle einer Methodenüberschreibung kann durch `tsuper` weiterhin auf die Methode der Super-Rolle

3 Um die Beispiele 16 und 17 kompilieren zu können wird empfohlen, sämtliche '$' durch ein Java-konformes Zeichen zu ersetzen. Einige Compiler (z.B. der Eclipse-2.1.1-Java-Compiler) „stolpern“ sonst über dieses Sonderzeichen. Der ObjectTeams/Java Compiler erzeugt intern jedoch genau die obenstehende Signatur, inklusive '$'. Außerdem muss der 'team' Modifier auskommentiert werden.

zugegriffen werden. Um dieses zu erreichen, wird für diesen konkreten Fall in die Signatur der kopierten Methoden/Konstuktoren, ein zusätzlicher Parameter eingefügt. Dieser Parameter ist vom Typ eines, mit dem Super-Team assoziierten, Interfaces (im folgenden Markerinterface genannt). Beim Referenzieren einer Methode der Super-Rolle mittels `tsuper`, wird dann vom Compiler diese `tsuper`-Referenz (`tsuper.method(Argumente)`) in eine `this`-Referenz (`this.method(Argumente, Markerargument)`) transformiert, wobei Markerargument ein nach Markerinterface gecastetes `null` Argument ist. Bei einer Referenz auf einen `tsuper`-Konstruktor verhält es sich analog; `tsuper(Argumente)` wird während des kompilierens in `this(Argumente, Markerargument)` transformiert.

Für eine nähere Erläuterung wird `T2.R1.getVersion()` aus Beispiel 13 wie folgt geändert:

```
...
03      public String getVersion(){
04        return "2" + tsuper.getVersion();
05      }
...
```

Beispiel 18

Der vom Compiler transformierte Code aus Beispiel 17 enthält anstelle der `tsuper`-Referenz eine um das Markerargument erweiterte `this`-Referenz:

```
...
14      public String getVersion(){
15        return "2" + this.getVersion((TSuper_OT$T1)null);
16      }
...
```

Beispiel 19

Die Ausgabe des Programms ist wie erwartet:

```
T21
```

Für eine Implementierung von Copy-Inheritance in ObjectTeams/Java reicht es nicht aus, einfach eine Sourcentransformation ähnlich eines Präprozessors durchzuführen. ObjectTeams soll ebenfalls mit vorkompilierten Super-Team Klassen arbeiten können für die kein Quellcode zur Verfügung steht. Um dies zu realisieren muss Copy-Inheritance nach dem selben Prinzip auch auf Bytecode-Ebene durchgeführt werden. Im Compiler wird genaugenommen eine Mischung aus beidem verwendet. Die auf Bytecode-Ebene zu kopierenden Methoden, werden in einer frühen Compilerphase vom Zustandsautomaten als leere Gerüste im AST erzeugt. Die Signatur dieser Methoden wird bereits vollständig aus den zur Verfügung stehenden Bindings erzeugt. Während der Codegenerierungsphase (siehe Abbildung 18) wird dann der Bytecode kopiert und in das ClassFile geschrieben.

Durch die obigen Beispiele sollte das Grundprinzip von Copy-Inheritance verständlich geworden sein. In den folgenden Abschnitten wird dieses Wissen vorausgesetzt.

4.1.4 Bytecodecopy

Das im vorherigen Abschnitt beschriebene Konzept zur Realisierung von Copy-Inheritance, scheint eine ideale Aufgabe für einen Compiler-Präprozessor zu sein. Voraussetzung dafür wäre allerdings, dass dem ObjectTeams/Java Compiler zu jeder Team-Klasse, der zugehörige Quellcode zur Verfügung stände. Diese Einschränkung ist in ObjectTeams/Java allein schon aus dem Grund nicht gewollt, da elementare Team-Klassen und andere für das ObjectTeams/Java-Laufzeitsystem notwendige Klassen, dem Programmierer nur in einem separaten Jar-Package zur Verfügung stehen sollen.

Diese Einschränkung hat zur Folge, dass Copy-Inheritance in jedem Fall zu einem Teil auf Bytecodeebene durchgeführt werden muss. Um die Herkunft von Fehlern im Compiler (z.B. Fehler die den Compiler zum Absturz bringen) eindeutig bestimmen zu können, ist es sinnvoll, möglichst wenig (redundante) Programmteile analysieren zu müssen. Für dieses konkrete Problem bedeutet dies; warum sollte ein Konzept in zwei unterschiedlichen Versionen (Sourcodecopy und Bytecodecopy) implementiert werden, wenn im Endeffekt eines davon (Bytecodecopy) ausreichen würde?
Die einzige Abhängigkeit die es durch ein ausschließliches Kopieren von Bytecode gibt, ist der als Quelle dienende Bytecode selber. Deshalb muss sichergestellt werden, dass dieser auch zum benötigten Zeitpunkt zur Verfügung steht. Im Compiler kann dies leicht durch eine Analyse vor dem eigentlichen Kompilierungsvorgang sichergestellt werden, bei der geprüft wird, ob der Bytecode der Super-Team-Klasse vorhanden ist, oder nicht. Wenn der Bytecode vorhanden ist, kann dieser als Quelle für Copy-Inheritance auf Bytecode-Ebene dienen. Wenn das Super-Team noch nicht in kompilierter Form vorliegt, muss der Kompilierungsvorgang rekursiv für das Super-Team ausgeführt werden. Dadurch wird sichergestellt, dass der notwendige Bytecode vor der Ausführung von Copy-Inheritance zur Verfügung steht.

Generell ist es denkbar, die im Rahmen von Copy-Inheritance zu kopierenden Methoden, vollständig in der Codegenerierungsphase (Phase 3.2 siehe Abbildung 18) aus dem Bytecode zu generieren. Während des Kompilierungsvorganges benötigt der Compiler jedoch schon zu einer früheren Phase 2.2 (siehe Beschreibung der Phasen in Abschnitt 3.4.4.3) die korrekten Bindings der zu kopierenden Methoden. Würden hier die zu kopierenden Methoden fehlen, wären etliche Sonderbehandlungen, z.B. für die Validierung von Referenzen auf mögliche fehlende, in einer späteren Phase zu erzeugenden Klassen/Methoden/Felder notwendig. Um diesen Mehraufwand durch ein sinnvolles Design zu vermeiden, werden vor einer Phase möglichst viele der darin benötigten Informationen gesammelt. Dadurch kann viel Arbeit von den im Compiler integrierten Automatismen erledigt werden.

Das Überprüfen von Typreferenzen kann von einem Compiler nur ausgeführt werden, wenn der referenzierte Typ in irgendeiner Form für diese Überprüfung vorliegt. Im Eclipse-

Java-Compiler wird aus einer Binary-Datei ein sogenanntes `BinaryTypeBinding` erzeugt. Dieses enthält für sämtliche in der Binary-Datei vorkommenden Typen (Klassen, Methoden und Felder), gültige Bindings, also deren vollständige Signatur. Genau aus diesen bereits vom Compiler zur Verfügung gestellten Bindings, werden auch die AST-Deklarationen, der zu kopierenden Typen, generiert. Außer für Methoden und Konstruktoren ist für keine der anderen erzeugten AST-Deklarationen das Kopieren von Bytecode notwendig. Aus diesem Grund wird auch nur für diese beiden Deklarationen eine Referenz auf den zu kopierenden und transformierenden Bytecode gespeichert.
Bindings enthalten in Eclipse-Java zwar von Haus aus keinen Bytecode, da diese speziellen Bindings jedoch direkt aus dem Bytecode erzeugt werden, ist es ein leichtes, sich eine Referenz auf das Erzeuger-Objekt zu merken, welches den Bytecode kennt.

4.1.5 Bytecodetransformation

Das Kopieren von Bytecode geschieht während der Codegenerierung in Phase 3.2 aus Abbildung 18. Kopieren bedeutet soviel wie, hole eine Kopie des Bytecode aus der Quelle und transformiere alle im Bytecode auftretenden Referenzen auf den Constant-Pool der Quelldatei, in Referenzen auf den aktuellen Constant-Pool der Zieldatei. Gegebenenfalls muss die neue Referenz erst noch im Constant-Pool erzeugt werden. Danach kann der transformierte Bytecode in die Zieldatei geschrieben werden.

In Beispiel 20 wird diese Transformation auf Bytecodeebene anhand der zu kopierenden `getVersion()` Methode dargestellt. Sämtliche in den Beispielen 20 und 21 angezeigten Bytecodeinformation wurden mit listclass [17], ein auf der BCEL-Library [18] basierendes Tool aus der Binary-Datei extrahiert. Die ersten vier Zeilen stellen einen Ausschnitt aus dem Constant-Pool dar. Die danach folgenden Zeilen repräsentieren eine für Menschen lesbare Version des eigentlichen Bytecode der Methode `getVersion()`.

T1$_OT$R1.class

```
...
41    27)CONSTANT_Utf8[1]("1")
42    28)CONSTANT_String[8](string_index = 27)
...
67    public String getVersion()
68    Code(max_stack = 1, max_locals = 1, code_length = 3)
69    0:    ldc               "1" (28)
70    2:    areturn
...
```

Beispiel 20

Der zu kopierende Bytecode enthält in diesem Fall lediglich eine einzige Referenz (28) auf den Constant-Pool (Zeile 69). Diese Referenz muss jedoch beim Kopieren berücksichtigt werden. In diesem Fall ist der referenzierte Eintrag an Position 28 (Zeile 42) im Constant-Pool, ein String mit dem Wert `"1"` (Zeile 41). Deshalb wird dieser aus dem Quell-Constant-Pool an der Stelle 27 (Zeile 41) ausgelesen. Jetzt muss im Ziel-Constant-Pool ein neuer Eintrag vom Typ String mit identischem Wert erzeugt werden. Sobald die

Position des Eintrages im Ziel-Constant-Pool feststeht, kann auch die Referenz korrigiert werden. Hier in diesem Fall wurde die Referenz (Zeile 80) von 28 auf 29 korrigiert.

T2$_OT$R1.class

```
...
45    28)CONSTANT_Utf8[1]("1")
46    29)CONSTANT_String[8](string_index = 28)
...
78    public String getVersion(T2$TSuper_OT$T1 _OT$marker)
79    Code(max_stack = 1, max_locals = 2, code_length = 3)
80    0:    ldc               "1" (29)
81    2:    areturn
...
```

Beispiel 21

Anhand dieser einfachen Transformation sollte das Konzept der Bytecodetransformation deutlich geworden sein. Wie diese Transformation im einzelnen aussieht und welche Vorarbeit dafür geleistet werden muss, wird im nächsten Abschnitt ausführlich erläutert.

4.1.6 Transformation der Constant-Pool Referenzen

Zuerst müssen im Bytecode die zu mappenden Referenzen auf den Constant-Pool ausfindig gemacht werden. Dafür muss im Bytecode zwischen Opcode und Operanden unterschieden werden können. Vorbedingung für einen korrekt funktionierenden Algorithmus ist natürlich ein korrekter Bytecode. Der erste Opcode kann aufgrund der Struktur des Code-Attributes, in dem zu einer Methode gehörenden Bytecode, gefunden werden. Da Opcodes eine unterschiedliche Anzahl von Operanden haben können, wird aus einer Tabelle die Anzahl der nachfolgenden Operanden ausgelesen. Der Opcode selber wird nicht weiter ausgewertet. Die nachfolgenden Operanden werden nach Referenzen auf den Constant-Pool hin untersucht. Die Position von Referenz innerhalb der Operandenliste eines Befehls ist für jeden Opcode eindeutig definiert. Wenn eine Referenz auf den Quell-Constant-Pool gefunden wurde, muss diese Referenz auf einen Eintrag(=Typ) im Ziel-Constant-Pool umgeleitet werden, sofern dieser bereits existiert. Sollte der Typ im Ziel-Constant-Pool noch nicht existieren, muss der Ziel-Constant-Pool um den gesuchten Typ (und alle durch diesen Typ referenzierten Typen) erweitert werden. Erst nach einer erfolgreichen Erweiterung des Ziel-Constant-Pools, kann die im Bytecode gefundene Referenz auf den Ziel-Constant-Pool umgemapped werden.

Folgende Tabelle (8) listet alle möglichen Einträge eines Constant-Pools auf und gibt Aufschluss über ein ggf. notwendiges Mapping der darauf verweisenden Referenzen:

Referenztyp	Mapping
Class	Wenn Class auf die Quell-Super-Rollenklasse zeigt, dann muss Class auf die aktuelle Ziel-Sub-Rollenklasse geändert werden. Es werden nur Referenzen auf den (nach dem Splitting erzeugten) Klassenteil einer Rolle gemapped. Für ein Mapping wird vorausgesetzt, dass der die Rolle umschliessende Typ (T1) ein Team ist. Bsp.: `class T1._OT$R1 => class T2._OT$R1` Bsp.: `interface T1.R1 => interface T1.R1`
MethodRef	**Class** siehe Class **NameAndType** kein Mapping erforderlich Bsp.: `String getName() =>  String getName()`
FieldRef	**Class** siehe Class **NameAndType** kein Mapping erforderlich Bsp.: `String name => String name`
NameAndType	keine direkte Referenz möglich, daher kein Mapping
InterfaceMethodRef	**Class** siehe Class **NameAndType** kein Mapping erforderlich Bsp.:`String getVersion() => String getVersion()`
String	Zeichenkette wird kopiert und Referenz wird auf Duplikat im Ziel Constant-Pool gesetzt
Utf8	keine direkte Referenz möglich, daher kein Mapping erforderlich
Integer	Integer Wert wird kopiert und Referenz wird auf Duplikat im Ziel Constant-Pool gesetzt
Long	Long Wert wird kopiert und Referenz wird auf Duplikat im Ziel Constant-Pool gesetzt
Float	Float Wert wird kopiert und Referenz wird auf Duplikat im Ziel Constant-Pool gesetzt
Double	Double Wert wird kopiert und Referenz wird auf Duplikat im Ziel Constant-Pool gesetzt

Tabelle 8

4.2 Translation Polymorphism

Um eine Navigation von Basis zur Rolle durchführen zu können, verwendet das in ObjectTeams/Java verwendete Translation Polymorphism Konzept, einen als Smart Lifting bezeichneten Mechanismus. Die generelle Funktionsweise dieses Mechanismus wird im nachfolgenden Abschnitt erklärt.

4.2.1 Smart Lifting

Der Smart Lifting Mechanismus bildet eine Basis auf eine Rolle ab und berücksichtigt dabei die Vererbungsbeziehungen der zur Laufzeit verfügbaren Objektinstanzen. Um zu einem gebundenen Basisobjekt das jeweils spezialisierteste Rollenobjekt zu finden, muss ein Mapping, zwischen den Vererbungshierarchien von Rolle und Basis, definiert werden. Statisch verbunden sind die beiden Vererbungshierarchien über alle deklarierten Bindungen zwischen Rollenklassen und Basisklassen (`Role playedBy Base`). Diese deklarierten Bindungen bilden nur im Idealfall jedes Basisobjekt auf genau ein Rollenobjekt ab. In der Praxis wird es bei komplexen Strukturen die Regel sein, dass innerhalb der Vererbungshierarchie nicht sämtliche Basisklassen und Rollenklassen explizit als gebunden deklariert sind. Diese Basisklassen sollen aber dennoch auf die am besten passendste Rollenklasse abgebildet werden. Wie dies erreicht wird, ist nachfolgend beschrieben. Zu der ungebundenen Basisklasse wird eine gebundene Super-Basisklasse gesucht. Für diese ist ein Mapping auf eine Rollenklasse definiert. Die auf diese Weise gefundene Rollenklasse ist jedoch für das gewünschte Ergebnis von Smart Lifting nicht spezialisiert genug. Die Besonderheit von Smart Lifting ist, dass Subtyp-Beziehungen von Basisklassen, automatisch auf Subtyp-Beziehungen von Rollenklassen abgebildet werden. Deshalb wird ausgehend von der gefundenen Rollenklasse ebenfalls die spezialisierteste ungebundene Sub-Rollenklasse gesucht und als Ergebnis von Smart Lifting zurückgeliefert.

Hierarchieanalyse

Problematisch sind Fälle, in denen kein eindeutiges Mapping definiert werden kann. Zum Beispiel wenn mehrere Rollenklassen an eine Basisklasse gebunden sind. Teilweise können diese Problemfälle bereits während des Kompilierungsvorganges erkannt werden, andere erst zur Laufzeit, wodurch eine `LiftingFailedException` ausgelöst werden muss. Für die Behandlung der Problemfälle ist während des Kompilierungsvorganges eine Hierarchieanalyse der Basis- und Rollen-Vererbungshierarchien notwendig.

Die möglichen Problemfälle und das vollständige Konzept für Smart Lifting ist unter [14] ausgiebig beschrieben.

5 Implementierung

5.1 ObjectTeams-Prototyp

Um nach der Entwurfsphase erste praktische Erfahrungen für die Erweiterung des Eclipse-Java-Compilers sammeln zu können, wurde in zwei Schritten ein einfacher Prototyp entwickelt. Zuerst wurde ein vertikaler Prototyp erstellt, um die notwendigen Änderungen für ein konkretes Element, durch alle Schichten des Systems kennenzulernen. Eine Schicht wurde dann in einem zweiten Verfeinerungsschritt bis in die Breite vollständig implementiert.

5.1.1 Vertikaler Prototyp

Innerhalb dieses vollständig durch alle Schichten des Systems implementierten vertikalen Prototypen, wurden zwei für ObjectTeams notwendige Spracheigenschaften in den bestehenden Compiler integriert: das `within`-Statement und der `team`-Modifier. Diese beiden Sprachelemente sollten vollständig implementiert werden. Für die Realisierung waren Anpassungen der Grammatik, des Scanners, des Parsers und des AST notwendig. Zusätzlich mussten noch einige kleinere Änderungen vorgenommen werden, wie z.B. die Erzeugung von Fehlermeldungen oder das Erstellen einer Methode zum Abfragen, ob der `team`-Modifiers innerhalb einer Klasse (`TypeDeclaration`) gesetzt ist.

1. Der erste Schritt war die Erweiterung der Grammatik. Diese musste um zwei Token und drei Regeln erweitert werden.
2. Die geänderte Grammatik musste in den Eclipse-Compiler integriert werden.
   Dieses wurde per Hand nach den unter [19] beschriebenen Schritten durchgeführt.
3. Der Scanner musste angepasst werden, so dass er die beiden neuen Token erkennt.
   Dafür war eine Anpassung der Methode `scanIdentifierOrKeyword()` notwendig.
4. Der AST musste um die Klasse `WithinStatement` erweitert werden.
5. Der Parser musste um die Methode `consumeWithinStatement()` erweitert werden.
   Diese Methode instanziert das `WithinStatement` und fügt es an die entsprechende Stelle im AST ein.
6. Innerhalb der Klasse `WithinStatement` musste überprüft werden, ob der übergebene Ausdruck eine Team-Instanz ist. Andernfalls sollte eine Fehlermeldung ausgegeben werden.

5.1.2 Horizontaler Prototyp

In einem zweiten Schritt wurde der Prototyp horizontal erweitert. Ziel dieses Vorgehens war, einen ersten ObjectTeams Compiler zu entwickeln, welcher sämtliche syntaktischen Sprachelemente scannen und parsen kann. Die Grammatik musste dafür vervollständig werden. Außerdem wurden die Klassen Scanner und Parser auf einen ersten finalen Stand gebracht.

5.1.2.1 Grammatikerweiterung

Mit der Vorlage der in Binder [20] definierten ObjectTeams Grammatik entstand eine erste erweiterte, im Eclipse-Parser verwendbare, ObjectTeams/Java-Grammatik (Siehe Beispiele 8 bis 23). Jedes dieser Beispiele stellt einen erweiterten oder geänderten Teil der insgesamt über 1300 Zeilen langen Grammatikdatei dar.

```
...
26 $Terminals
...
39  base tsuper callin within playedBy with team as
40 -- Special-Identifier:
41    result replace after before
...
102  BINDIN
103  BINDOUT
104  CALLOUT_OVERRIDE
...
```

Beispiel 22

Da ObjectTeams eine auf Java basierende Sprache ist, gibt es aufgrund der Java-API Einschränkungen bezüglich der Definition von neuen Schlüsselwörtern. Als abschreckendes Beispiel sei hier das Schlüsselwort „within“ genannt, welches in seiner urtümlichen Form „in“ lautete und dadurch das Problem mit sich brachte, `java.lang.System.in` nicht mehr innerhalb eines ObjectTeams/Java Programmes verwenden zu können. Durch eine spezielle Namenskonvention (z.B. dass alle neuen ObjectTeams Schlüsselwörter mit „_ot_“ anfangen) könnten zwar die meisten Probleme vermieden werden, dies würde aber dazu führen, dass ObjectTeams-Syntax innerhalb eines ObjectTeams-Programmes als etwas Fremdartiges empfunden werden könnte. Da diese psychologische Barriere genauso wenig wie das Problem mit vorhandener API gewollt ist, wird in ObjectTeams ein besonderer Mechanismus verwendet:

SpecialIdentifier

Durch die Einführung von *SpecialIdentifieren* ist es möglich, die als SpecialIdentifier definierten Schlüsselwörter an beliebiger Stelle im ObjectTeams-Quellcode als Identifier zu verwenden, ohne dass der Compiler eine Fehlermeldung ausgibt. Dies wird erreicht, indem SpecialIdentifier in den Regeln der Grammatik als normale Identifier auftreten und erst während des parsierens kontextbezogen ausgewertet werden.
Jedoch gibt es auch für diesen Mechanismus Einschränkungen bezüglich der Grammatik. In einer LALR(1) Grammatik (Siehe 3.1.3) ist es nicht möglich, mehrere aufeinander folgende optionale Identifier innerhalb einer Regel zu definieren, da der daraus resultierende Automat nicht entscheiden könnte, welcher Identifier gerade gelesen wurde. Jikespg würde dieses Problem mit einer Fehlermeldung „This grammar is not LALR(1)“ quittieren.
Wichtig ist noch, dass SpecialIdentifier innerhalb der Grammatik in keiner Regel auftreten dürfen, da sonst der syntaktische Kontext definiert wäre. Eine Definition der SpecialIdentifier innerhalb der Liste aller terminalen Symbole ist trotzdem sinnvoll, um die

im Parser benötigten Konstanten für diese SpecialIdentifier durch jikespg automatisch generieren zu lassen.
Die Zeile 102 bis 104 aus Beispiel 22 definieren die terminalen Symbole für Callin und Callout-Bindings in Form von Token. Die Zuweisung der entsprechenden terminalen Symbol findet in der Sektion $names statt (Siehe Beispiel 23 $names).

```
...
1355 $names
...
1410 BINDIN ::= '<-'
1411 BINDOUT ::= '->'
1412 CALLOUT_OVERRIDE ::= '=>'
...
1415 $end
```

Beispiel 23

```
...
109 $Alias
...
160  '<-'   ::= BINDIN
161  '->'   ::= BINDOUT
162  '=>'   ::= CALLOUT_OVERRIDE
...
```

Beispiel 24

```
...
168 $Rules
169 /. // This method is part of an automatic generation : do NOT edit-modify
170 protected void consumeRule(int act) {
171   switch ( act ) {
172 ./
...
240 Type -> LiftingType
241 LiftingType ::= ClassType 'as' ClassType
242 /.$putCase consumeLiftingType(); $break ./
...
353 Modifier -> 'team'
354 Modifier -> 'callin'
...
368 -- ClassHeader ::= ClassHeaderName ClassHeaderExtendsopt
ClassHeaderImplementsopt
369 ClassHeader ::= ClassHeaderName ClassHeaderExtendsopt
ClassHeaderImplementsopt ClassHeaderPlayedByopt
...
384 ClassHeaderPlayedByopt ::= $empty
385 ClassHeaderPlayedByopt -> ClassHeaderPlayedBy
386
387 ClassHeaderPlayedBy ::= 'playedBy' ClassType
388 /.$putCase consumeClassHeaderPlayedBy(); $break ./
...
410 ClassMemberDeclaration -> BindingDeclaration
411
412 ---- CALLOUT-BINDING
413 BindingDeclaration -> CalloutBinding
414
```

```
415 CalloutBinding ::= Modifiersopt MethodSpecShort CalloutKind MethodSpecShort
CalloutParameterMappingsopt
416 /.$putCase consumeCalloutBindingShort(); $break ./
417
418 CalloutBinding ::= MethodSpecLong CalloutKind MethodSpecLong
CalloutParameterMappingsopt
419 /.$putCase consumeCalloutBindingLong(); $break ./
420
421 CalloutKind -> '->'
422 CalloutKind -> '=>'
423
424 ---- CALLIN-BINDING
425 BindingDeclaration -> CallinBinding
426
427 CallinBinding ::= Modifiersopt MethodSpecShort '<-' CallinModifier
MethodSpecsShort CallinParameterMappingsopt
428 /.$putCase consumeCallinBindingShort(); $break ./
429
430 CallinBinding ::= MethodSpecLong '<-' CallinModifier MethodSpecsLong
CallinParameterMappingsopt
431 /.$putCase consumeCallinBindingLong(); $break ./
432
433 -- replace before after
434 CallinModifier ::= 'Identifier'
435 /.$putCase consumeCallinModifier(); $break ./
436
437 ---
438
439 MethodSpecShort ::= SimpleName
440 /.$putCase consumeMethodSpecShort(); $break ./
441
442 MethodSpecsShort -> MethodSpecListShort
443
444 MethodSpecListShort -> MethodSpecShort
445 MethodSpecListShort ::= MethodSpecListShort ',' MethodSpecShort
446 /.$putCase consumeMethodSpecList(); $break ./
447
448 ---
449
450 MethodSpecLong ::= MethodHeaderName MethodHeaderParameters
451 /.$putCase consumeMethodSpecLong(); $break ./
452
453 MethodSpecsLong -> MethodSpecListLong
454
455 MethodSpecListLong -> MethodSpecLong
456 MethodSpecListLong ::= MethodSpecListLong ',' MethodSpecLong
457 /.$putCase consumeMethodSpecList(); $break ./
458
459 ---- CALLOUT-BINDING-PARAMETERMAPPING
460 CalloutParameterMappingsopt ::= ';'
461
462 /.$putCase consumeCalloutParameterMappingsopt(); $break ./
463 CalloutParameterMappingsopt -> CalloutParameterMappings
464 diese
465 CalloutParameterMappings ::= 'with' '{' CalloutParameterMappingList '}'
466
467 CalloutParameterMappingList -> CalloutParameterMapping
```

```
468 CalloutParameterMappingList ::= CalloutParameterMappingList ','
CalloutParameterMapping
469 /.$putCase consumeCalloutParameterMappingList(); $break ./
470
471 CalloutParameterMapping ::= Expression '->' 'Identifier'
472 /.$putCase consumeCalloutParameterMappingOut(); $break ./
473
474 CalloutParameterMapping ::= 'Identifier' '<-' Expression
475 /.$putCase consumeCalloutParameterMappingIn(); $break ./
476
477 ---- CALLIN-BINDING-PARAMETERMAPPING
478 CallinParameterMappingsopt ::= ';'
479 /.$putCase consumeCallinParameterMappingsopt(); $break ./
480 CallinParameterMappingsopt -> CallinParameterMappings
481
482 CallinParameterMappings ::= 'with' '{' CallinParameterMappingList '}'
483
484 CallinParameterMappingList -> CallinParameterMapping
485 CallinParameterMappingList ::= CallinParameterMappingList ','
CallinParameterMapping
486 /.$putCase consumeCallinParameterMappingList(); $break ./
487
488 CallinParameterMapping ::= 'Identifier' '<-' Expression
489 /.$putCase consumeCallinParameterMappingIn(); $break ./
490
491 CallinParameterMapping ::= Expression '->' 'Identifier'
492 /.$putCase consumeCallinParameterMappingOut(); $break ./
...
681 ExplicitConstructorInvocation ::= 'tsuper' '(' ArgumentListopt ')' ';'
682 /.$putCase consumeExplicitConstructorInvocation
(0,ExplicitConstructorCall.Tsuper); $break ./
683
684 ExplicitConstructorInvocation ::= 'base' '(' ArgumentListopt ')' ';'
685 /.$putCase consumeExplicitConstructorInvocation
(0,ExplicitConstructorCall.Base); $break ./
...
812 Statement -> Within
813 Within ::= 'within' '(' Expression ')' Statement
814 /.$putCase consumeWithin(); $break ./
...
1085 MethodInvocation ::= 'tsuper' '.' 'Identifier' '(' ArgumentListopt ')'
1086 /.$putCase consumeMethodInvocationTSuper(); $break ./
1087
1088 MethodInvocation ::= 'base' '.' 'Identifier' '(' ArgumentListopt ')'
1089 /.$putCase consumeMethodInvocationBase(); $break ./
...
1348 /.  }
1349 } ./
```

Beispiel 25

ObjectTeams soll die Java-Syntax erweitern und nicht einschränken. Deshalb mussten auch fast keine Änderungen an bestehenden Regeln vorgenommen werden. Die Zeile 369 zeigt den einzigen invasiven Eingriff in die bestehende Java-Syntax. Ein Klassen-Header kann jetzt eine optionale `playedBy` Referenz haben. Alle anderen Erweiterungen an der Grammatik konnten durch das Einfügen zusätzlicher Regeln durchgeführt werden.

Die Entwicklung der Grammatik war ein mühseliger und keinesfalls einfacher Prozess. Immer wieder traten Probleme auf, deren Ursache nicht offensichtlich zu erkennen war. Die häufigsten Probleme waren shift/reduce Konflikte. Trat ein solcher Konflikt auf, dann half nur noch ein tiefer Blick in die von jikespg erzeugte Logdatei (<grammatikname>.l). Diese fast 50000 Zeilen lange Datei gibt letztendlich Aufschluss über sämtliche erzeugten Zustände, Tabellen und Konflikte die während der Parsergenerierung von Bedeutung sind. Immer wieder wurde dabei deutlich, dass die ObjectTeams-Syntax die Mächtigkeit einer LALR(1) Grammatik durchaus auslotet.
Besondere Schwierigkeiten bereitete die Definition von `MethodSpecs` (Beispiel 25 Zeilen 439 ff).

MethodSpec ist in der ObjectTeams-Sprachdefinition unter $8.3.4 [13] folgendermaßen definiert:

```
01 MethodSpec:
02        Identifier
03        MethodHeader
```

Die Definition von MethodHeader in der "Java language specification" [21] in der Sektion 8.4. Method Declarations lautet:

```
01 MethodHeader:
02        MethodModifiersopt ResultType MethodDeclarator Throwsopt
```

Die intuitive Umsetzung von $8.3.4 innerhalb der ObjectTeams-Grammatik sähe folgendermaßen aus:

```
01 MethodSpec ::= 'Identifier'
02 MethodSpec -> MethodHeader
```

Ein sich daraus ergebendes nicht offensichtliches Problem ist, dass, wenn man diese in $8.3.4 definierte ObjectTeams Spracheigenschaft eins-zu-eins umsetzt, dass die daraus resultierende Grammatik nicht mehr LALR(1) ist.

Nach dem Aufruf von `jikespg objectteams.g` ergibt sich folgende Fehlermeldung:

```
01 IBM Research Jikes Parser Generator            Fri Jun 20 15:34:57 2003
...
33 *** Shift/reduce conflict on "Identifier" with rule 460
34 *** Shift/reduce conflict on "Identifier" with rule 460
35 *** Shift/reduce conflict on "Identifier" with rule 460
36 *** Shift/reduce conflict on "Identifier" with rule 460
37 *** Shift/reduce conflict on "Identifier" with rule 460
38 *** Shift/reduce conflict on "Identifier" with rule 460
39 *** Shift/reduce conflict on "COMMA" with rule 180
40 *** Shift/reduce conflict on "Identifier" with rule 460
41
42 This grammar is not LALR(1).
43
```

```
44 Number of Terminals: 119
45 Number of Nonterminals: 222
46 Number of Productions: 487
47 Number of Single Productions: 181
48 Number of Items: 1444
49 Number of States: 664
50 Number of Shift actions: 4348
51 Number of Goto actions: 4846
52 Number of Shift/Reduce actions: 434
53 Number of Goto/Reduce actions: 756
54 Number of Reduce actions: 8360
55 Number of Shift-Reduce conflicts: 8
56 Number of Reduce-Reduce conflicts: 0
...
```

Beispiel 26

Tritt solch eine Fehlermeldung auf, kann nicht mehr sichergestellt werden, dass alle Spracheigenschaften, vom aus der Grammatik automatisch generierten Parser, korrekt geparsed werden. Die Ursachen für diese nicht LALR-Konformität liegt in den sich ergebenden shift/reduce Konflikten innerhalb der Action-Goto-Parsertabellen. Der Parser kann nicht entscheiden, ob er ein shift oder ein reduce anwenden soll. Im Idealfall kann ein Parser trotzdem fehlerfrei arbeiten; er entscheidet sich im Zweifelsfall deterministisch entweder für shift oder reduce. Die dadurch auftretenden Seiteneffekte sind allerdings nicht trivial.

Die Hoffnung, dass der Parser trotz dieser Konflikte korrekt arbeitet, erfüllte sich leider nicht. Er erkannte zwar einwandfrei alle `MethodSpec` innerhalb eines ObjectTeams-Programmes, aber an einer anderen Sourcecode-Stelle dafür keine Callin-Bindings mehr. Callout-Bindings wurden dagegen immer noch korrekt behandelt. Weitere Auswirkungen hätten sein können, dass innerhalb der bestehenden Java-Grammatik nicht mehr alle Spracheigenschaften zur Verfügung stehen.

Gelöst wurde das Problem in der endgültigen Grammatik durch die Aufteilung der MethodSpec in einen Long und einen Short Teil. Außerdem wird anstelle eines Identifiers ein `SimpleName` gelesen.

```
272 SimpleName -> 'Identifier'
```

5.1.2.2 Präprozessor für optionales replace

Ein durch die Grammatik nicht lösbares Problem, ist die Definition eines optionalen CallinModifiers (Beispiel 25 Zeile 430ff) der gleichzeitig ein SpecialIdentifier ist. Dies hängt damit zusammen, dass bei zwei aufeinander folgenden Identifiern, von denen der erste optional ist, der Automat nicht entscheiden kann, welchen er gerade liest. Um trotzdem einen optionalen Callin-Modifier zu ermöglichen, wird bei der Initialisierung des Scanners ein Trick angewendet: ein Defaultwert (`replace`) wird mittels eines Präprozessors anstelle eines nicht vorhandenen Callin-Modifiers in den Sourcecode eingefügt (`compiler.parser.Scanner.insertOptionalReplace()`). Dadurch muss der `CallinModifier` in der Grammatik nicht als optional definiert werden. Nachteil

dieses Tricks ist, dass der Scanner zweimal über den Quellcode laufen muss, was natürlich Performanceeinbußen mit sich bringt.

5.1.3 Parsergenerierung

Der Parsergenerator jikespg [9] erzeugt aus einer Grammatik eine Reihe von Tabellen in Form von Java-Dateien. Ein Teil dieser Tabellen muss per Copy&Paste direkt in die Klasse Parser.java kopiert werden, ein anderer Teil ist erst mit einem Java-Programm in Binaries zu konvertieren. Insgesamt sind es zu viele Schritte, um diese öfters als einmal per Hand durchführen zu wollen. Zur Vermeidung von Fehlern, bei der sowieso schon schwierigen Parsergenerierung, wurde ein einfach zu bedienendes bash-Skript (Beispiel 27) generateOTParser.sh erstellt, welches genau die unter [19] beschriebenen Arbeitsschritte durchführt.

Während der Parsergenerierung werden durch das Skript folgende Dateien des Eclipse-OTDTCore-Plugins verändert, bzw. ersetzt.

compiler/org/eclipse/jdt/internal/compiler/parser/Parser.java
compiler/org/eclipse/jdt/internal/compiler/parser/TerminalTokens.java
compiler/org/eclipse/jdt/internal/compiler/parser/ParserBasicInformation.java
compiler/org/eclipse/jdt/internal/compiler/parser/parser[1-5].rsc

Das Skript benötigt als Kommadozeilenparameter die Grammatik und das OTDTCore-Plugin. Bei einem Aufruf ohne Parameter wird eine kurze Hilfe ausgegeben.

Beispiel:
data@home>generateOTParser.sh /home/data/objectteams.g /home/data/OTDTCore

Der nachfolgende Ausschnitt aus dem Skript zeigt den Mechanismus, durch welchen in die existierende Datei Parser.java die neue `consumeRule()` Methode eingefügt wird. Mit Hilfe des Unix-Kommandozeilen-Tools awk wird in der Datei Parser.java nach dem Beginn der `consumeRule()`-Methode gesucht. Die gesamte Methode wird durch das Schlüsselwort TMP_CUT_OUT ersetzt. Dann wird die Datei an dieser Stelle zerschnitten. Zwischen beiden Hälften wird dann die neue von jikespg generierte `consumeRule()`-Methode eingefügt.

generateOTParser.sh

```
...
155 cd output
156 echo "Cut all unnecessary from: JavaAction.java"
157 cat JavaAction.java | awk '{\
158  if($0 !~ /^[ \t]*\/\/ This method is part of an automatic gene\
159           ration : do NOT edit-modify[ \t]*$/ ){\
160    print $0\
161  }\
162 }\
163 ' > JavaAction_tmp.java
```

generateOTParser.sh

```
164
165 echo "Replace method consumeRule with token TMP_CUT_OUT"
166 cat Parser.java  | awk '{
167 if($0 ~ /^[ \t]*protected void consumeRule\(int act\) \{[ \t]*$/ ){
168    while((getline nextline) == 1 && nextline !~ /^[ \t]*}[ \t]*$/){
169      nextline="" ;
170    }
171    while((getline nextline) == 1 && nextline !~ /^[ \t]*}[ \t]*$/){
172      nextline="" ;
173    }
174    print "TMP_CUT_OUT";
175    while((getline nextline) == 1 ){
176      print nextline ;
177    }
178  } else {
179    print $0
180  }
181 }
182 ' > Parser_tmp.java
183
184 cat Parser_tmp.java  | awk '{
185   if($0 ~ /^TMP_CUT_OUT$/ ){
186    while((getline nextline) == 1){
187      nextline="" ;
188    }
189  } else {
190    print $0
191  }
192 }
193 ' > Parser_1_tmp.java
194
195 cat Parser_tmp.java  | awk '{
196  if($0 ~ /^TMP_CUT_OUT$/ ){
197    while((getline nextline) == 1){
198      print nextline;
199    }
200  }
201 }
202 ' > Parser_2_tmp.java
203
204 echo "Insert content of JavaAction.java into Parser.java"
205 cat Parser_1_tmp.java JavaAction_tmp.java Parser_2_tmp.java > Parser.java
...
```

Beispiel 27

Durch dieses Beispiel soll deutlich werden, dass direkte Abhängigkeiten zwischen dem Skript und der Formatierung der Datei Parser.java existieren. Das Skript schlägt fehl, sobald die *Regulären Ausdrücke* nicht mehr die richtigen Stellen innerhalb des Quelltextes finden können. Deshalb sollte bei einer Neu- oder Umformatierung der Datei Parser.java immer überprüft werden, ob diese eventuell Auswirkungen auf die Funktionsfähigkeit des Skriptes haben könnte.

Wenn sich das Skript fehlerfrei beendet, wurden alle für die Anpassung des Parsers notwendigen Schritte innerhalb des OTDTCore-Plugins durchgeführt. Sollten während der

Generierung, aufgrund von Skriptfehlern, Probleme mit den erzeugten Dateien auftreten, können diese leicht durch die im Ordner backup automatisch gesicherten Originaldateien wieder ersetzt werden.

5.1.4 AST-Erweiterung

Der AST wurde um einige ObjectTeams-spezifische Klassen erweitert. Die Anzahl aller AST Klassen ist jedoch so groß, dass sich ein vollständiges Klassendiagramm nicht sinnvoll innerhalb dieser Arbeit als ein Gesamtbild abbilden läßt. Für eine vollständige Übersicht über alle Klassen, sei auf das Package `org.eclipse.jdt.internal.compiler.ast` verwiesen.

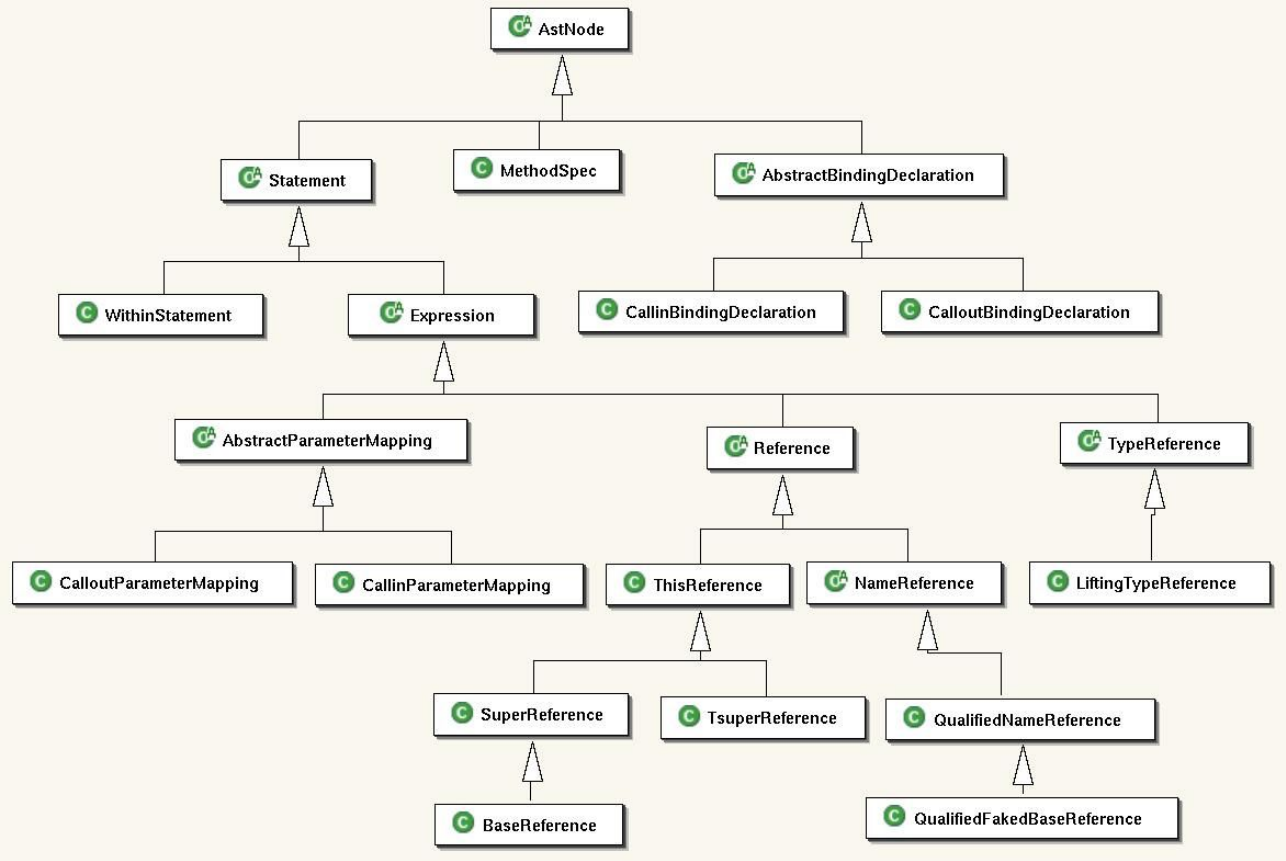


Abbildung 21 AST Erweiterung

Hinzugefügte AST-Klasse	Beschreibung
AbstractBindingDeclaration	**extends** AstNode
CallinBindingDeclaration	**extends** AbstractBindingDeclaration Ein Callin-Binding besteht aus MethodSpec(s) und CallinParameterMapping(s) Beispiel: A <- B;

Hinzugefügte AST-Klasse	Beschreibung
CalloutBindingDeclaration	**extends** AbstractBindingDeclaration Callout-Binding besteht aus MethodSpec(s) und CalloutParameterMapping(s) Beispiel: A -> B;
AbstractParameterMapping	**extends** Expression
CallinParameterMapping	**extends** ParameterMapping Parameter-Mapping Teil eines Callin-Bindings Beispiel: a <- b;
CalloutParameterMapping	**extends** ParameterMapping Parameter-Mapping Teil eines Callout-Bindings Beispiel: a -> b;
LiftingTypeReference	**extends** TypeReference Liffing-Type A as B
BaseReference	**extends** SuperReference Referenz auf die an die Rolle gebundene Basisklasse Beispiel: base();
TsuperReference	**extends** ThisReference Referenz auf die implizite Superklasse einer Rolle. Beispiel: tsuper()
QualifiedFakedBaseTagReference	**extends** QualifiedNameReference Referenz auf ein spezielles Attribut der Basisklasse, welches zur Compilerzeit noch nicht innerhalb der Basisklasse existiert. Sondern erst zur Laufzeit. Beispiel: _OT$base_arg._OT$Tag
MethodSpec	**extends** AstNode Spezifikation der Methodensignaturen innerhalb eines Callin- oder Callout-Bindings Beispiel: int A(int a)
WithinStatement	**extends** Statement Statement zur Aktivierung eines Teams Beispiel: within(MyTeam)

Tabelle 9

5.2 Portierung

Ausgehend von einem existierenden ObjectTeams/Java-Compiler, soll dessen ObjectTeams-Spezifische Funktionalität in einen neuen, auf einer anderen Architektur basierenden Java-Compiler portiert werden. Dafür galt es herauszufinden, was portiert

werden kann und auf welche Weise der Referenz-Compiler sonst noch für die Entwicklung des Systems von Nutzen ist.

Portiert werden können neben der offensichtlichen Programmfunktionalität in Form von (modularisiertem) Quellcode auch theoretische Konzepte, die für die Realisierung des Compilers verwendet wurden, sich aber nicht innerhalb des Systems an einer Stelle lokalisieren lassen. Diese hier gemeinten theoretischen Konzepte sind vielmehr über das System, entlang des Kontrollflusses, verteilt.
Aber es gibt noch weitere für den Entwicklungsprozess nützliche Eigenschaften eines Referenzsystems. Der existierende ObjectTeams-Compiler otc [13] diente der Analyse vorhandener Compiler-Funktionalitäten. ObjectTeams-Programme konnten kompiliert, und das Ergebnis mit Hilfe von jad [22] oder listclass [17] ausgewertet werden. Anhand dieser Ergebnisse wurde ermittelt, welche Transformationen der ObjectTeams-Compiler für ein konkretes ObjectTeams-Sprachfeature durchführt. Dieses Ergebnis konnte zum einen dazu verwendet werden, um die für dieses Sprachfeature notwendige Transformation zu implementieren, zum anderen aber auch dazu, einzelne Teile des implementierten Systems zu validieren. Eine nicht realisierte Idee ging sogar soweit, die Ergebnisse beider Compiler über eine Testsuite miteinander zu koppeln, und dann das Ergebnis auf Bytecodeebene automatisch vergleichen zu lassen.

Die Grundlage für die Implementierung der LiftTo-Methode (siehe 4.2) wurde mit Hilfe des Referenz-Compilers realisiert. Es wurde mit otc ein ObjectTeams-Programm kompiliert, welches als Ergebnis die erzeugten LiftTo-Methoden enthält. Der auf diese Weise erzeugte Bytecode wurde mit jad dekompiliert. Diese jad-Datei enthielt jetzt den vollständigen vom Compiler transformierten Code. Die daraus extrahierten LiftTo-Methoden wurden verallgemeinert und deren Erzeugung in den Compiler integriert.

5.2.1 Was wurde portiert?

Da die Architektur beider Compiler stark unterschiedlich ist, ließen sich nur wenige Teile portieren. Eine erste Idee, Adapter für bestehende Klassen zu implementieren, wurde schnell verworfen. Es existierten zu viele Abhängigkeiten zu anderen Klassen. Als einziges wurde die Klasse OTByteCodes.java portiert. Diese Klasse stellt auf Bytecodeebene Funktionalität zur Verfügung, um die Position von Constant-Pool-Referenzen innerhalb des Bytecodes ausfindig zu machen. Die Klasse enthält nur statische Methoden, welche bis auf eine Ausnahme als Argument- und Rückgabetypen ausschließlich primitive Datentypen enthalten. Die einzige Methode, welche keine primitiven Datentypen enthielt, konnte auskommentiert werden, da diese aufgrund eines anderen Compiler-Konzeptes nicht verwendet wird. Eine weitere Abhängigkeit zum Referenzsystem waren die verwendeten Konstanten (Opcodes). Diese waren in einem Interface definiert, welches in ähnlicher Form bereits im Eclipse-Compiler existiert. Aufgrund unterschiedlicher Namen mussten alle innerhalb der Klasse OTByteCodes auftretenden Konstanten umbenannt werden.

5.3 Implementierung

5.3.1 Implementierung des Zustandsautomaten

Der Eclipse-Java-Compiler durchläuft während seines Kompilierungsvorganges mehrere unterschiedliche Phasen (siehe Phasenmodell Abbildung 18). In jeder dieser Phasen werden sämtliche Aufgaben erledigt, die für die Durchführung der nachfolgenden Phasen notwendig sind.
Das in dieser Arbeit vorgestellte Konzept zur Realisierung von Implicit-Inheritance (siehe Abschnitt 3.3.1) soll nach dem Prinzip implementiert werden, möglichst viele der notwendigen Aufgaben, durch bereits vorhandene Mechanismen, in der jeweils dafür zuständigen Compilerphase, automatisch erledigen zu lassen. Während der für Implicit-Inheritance notwendigen AST-Transformation, werden AST-Objekte erzeugt bzw. verändert. Für diese AST-Objekte muss Funktionalität implementiert werden, die sicherstellt, dass das Objekt der aktuell erreichten Compilerphase entspricht. Außerdem soll keine inverse Funktionalität zu bereits vorhandener implementiert werden müssen (Invers wäre z.B. Funktionalität, welche ein bestehendes in seinen Scope eingetragenes Binding wieder aus demselbigen entfernt).
Natürlich wäre es denkbar, den Compiler seine Arbeit verrichten zu lassen, um dann erst in der letzten Phase vor der Codegenerierung alle Transformationen auf einen Schlag durchzuführen. Der Vorteil wäre ein leicht nachvollziehbarer Kontrollfluss. Der Nachteil wäre die Notwendigkeit, bestehende Zustände des Compilers inklusive des AST korrigieren zu müssen (z.B. Fehlermeldungen abfangen, bestehende Bindings ändern, Scope korrigieren).
Implicit-Inheritance benötigt für seine vollständige Implementierung Daten, welche erst zu unterschiedlichen Compilerphasen im AST zur Verfügung stehen. RoleSplitting kann bereits direkt nach dem Parsen der CompilationUnit ausgeführt werden, wogegen Copy-Inheritance erst nach dem Auflösen der Referenz auf die Superclasse des Teams durchgeführt werden kann. Transformationen an Statements, wie z.B. das Ersetzen aller Rollenkonstruktor-Aufrufe durch die zugehörige Konstruktor-Methode kann erst durchgeführt werden, wenn die Methodenrümpfe (die Statements) parsiert sind.
Stellt man für das gesamte Konzept einen Abhängigkeitsgraphen auf, wird ersichtlich, dass Daten in unterschiedlichen Phasen des Compilers benötigt, erzeugt, transformiert und evtl. noch adaptiert werden müssen. Ohne eine Art Zustandsmodell welches mit den Compilerphasen synchronisiert wird, ist es schwer möglich, sämtliche quer zum Kontrollfluss liegenden Transformationen konsistent durchzuführen. Die naheliegendste Lösung wäre ein zusätzliches Attribut innerhalb eines AstNodes, in welchem der aktuelle Zustand gespeichert wird. Dies wäre jedoch ein vermeidbarer invasiver Eingriff in den AST und wurde deshalb als Möglichkeit verworfen. Alternativ kommt für diesen Zweck eine quer zum Kontrollfluss existierende Menge von Objekten in Frage, in denen Zustände auf die zu transformierenden Daten (z.B. AstNodes) abgebildet werden. Die Menge aller statischen möglichen Transformationen zwischen verschiedenen Zuständen bezeichnet man in der Softwaretechnik als Zustandmodell (Statecharts) und das zugrundeliegende zustandstransformierende Modell einen Zustandsautomaten (Statemachine). Das folgende Statechart-Diagramm (Abbildung 22) stellt das für ObjectTeams/Java in Eclipse integrierte Zustandsmodell zur Realisierung der benötigten Transformationen dar. Die Funktionsweise des Zustandsautomaten wird in Abschnitt 5.3.1 beschrieben.

Übersicht über das Zustandsmodell

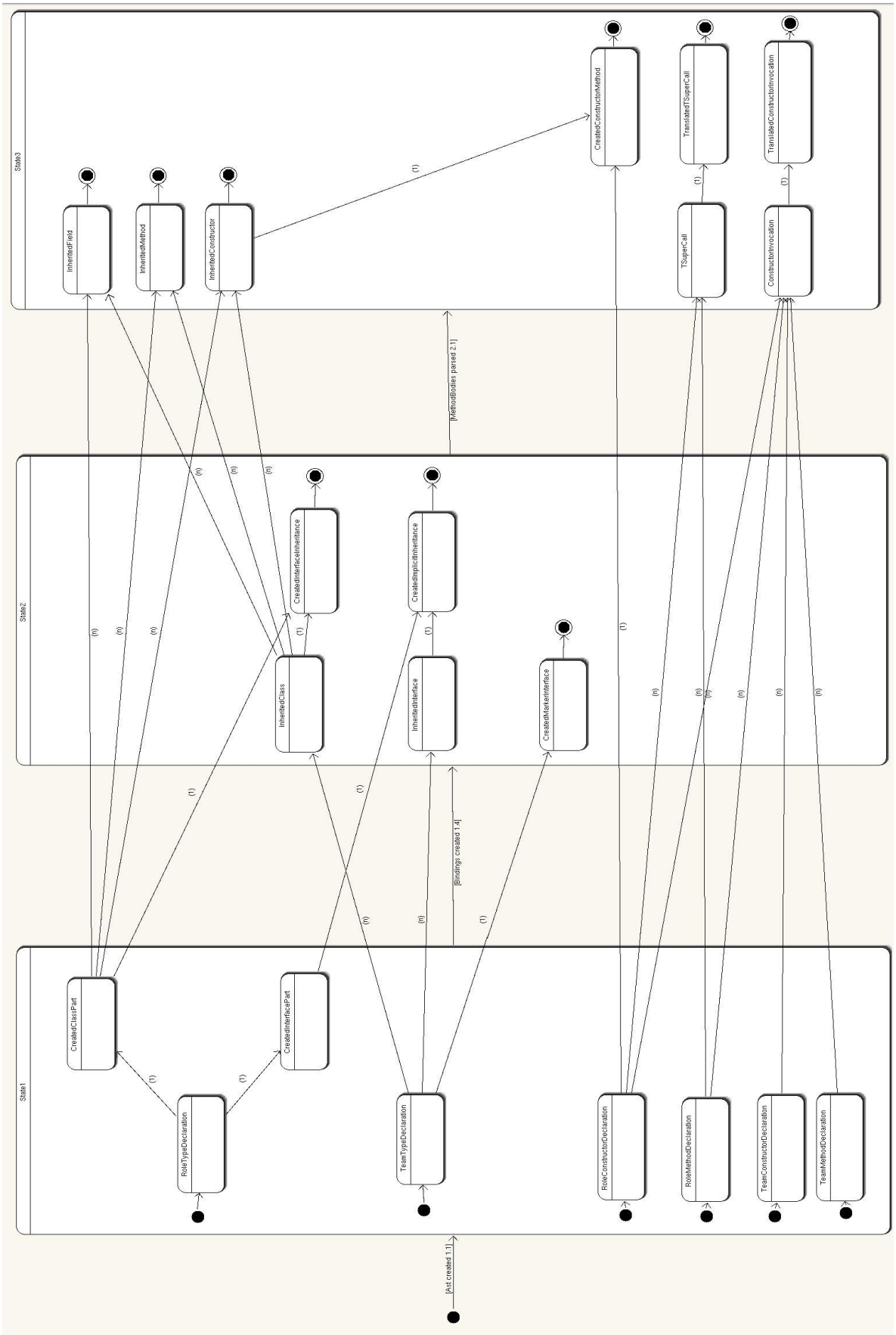


Abbildung 22 Zustandsmodell

Eine Besonderheit des in Abbildung 22 definierten Zustandsmodells ist dessen Fähigkeit, einen Zustand in mehrere, quasi parallel arbeitende, Zustände zu transformieren. Zum Beispiel wird im Kontext State1 der Zustand `RoleTypeDeclaration` in die beiden Zustände `CreatedClassPart` und `CreatedInterfacePart` transformiert. Der transformierte Zustand wird während der Transformation aus dem Modell entfernt und jeder der beiden Nachfolgezustände wird jetzt unabhängig voneinander im Zustandsautomaten verwaltet. Eine bei einer Erweiterung des Modells zu berücksichtigende Einschränkung ist, dass die Vorbedingung für alle Folgezustände erfüllt sein muss, damit eine Transformation stattfinden kann. Der Grund dafür ist eine Abhängigkeit beim Erzeugen der direkten Folgezustände. Als Beispiel sei dafür das Umbenennen der Rollenklasse (während des Erzeugens von `CreatedClassPart`) und das Erzeugen des Interfaces (während des Erzeugens von `CreatedInterfacePart`) im Rahmen von RoleSplitting erwähnt.

Wenn man davon ausgeht, dass innerhalb des Zustandes `RoleTypeDeclaration` eine Referenz auf einen `AstNode` gehalten wird und nicht ein Duplikat des `AstNode`-Objektes im Zustand existiert, macht es durchaus einen Unterschied, ob man die als Quelle dienende Rollenklasse zuerst umbenennt und dann ein Interface daraus erzeugt, oder ob man zuerst das Interface erzeugt und danach die Umbenennung der Rollenklasse vornimmt. Natürlich sind beide Varianten implementierungstechnisch möglich, es muss aber klar definiert sein, in welcher Reihenfolge Nachfolgezustände erzeugt werden.
Eine weitere Besonderheit stellen die Zustände State1 bis State3 dar. Diese Zustände stellen keine im Zustandsautomaten direkt abbildbaren Zustände dar, sondern dienen im Diagramm des Zustandmodells vielmehr der Übersichtlichkeit. Die Bedeutung der Bedingung für einen Zustandsübergang von StateN nach StateN+1 ist, dass sämtliche inneren Zustände diese Bedingung explizit prüfen und ein Zustandsübergang nur ausgeführt wird, wenn sie erfüllt ist.

5.3.1.1 Zustände des Zustandsautomaten

Die in Abbildung 22 abgebildeten Zustände lassen sich in 3 unterschiedliche Kategorien einteilen. Initiale Zustände haben keinen Vorgänger, und werden direkt aus einer `CompilationUnitDeclaration` erzeugt. Transiente Zustände haben sowohl einen Vorgänger als auch einen Nachfolger und werden immer aus den Daten erzeugt, welche im Vorgänger verfügbar sind. Der Endzustand hat weder Daten noch einen nachfolgenden Zustand und ist selbstterminierend. Bis auf den Endzustand enthalten alle Zustände eine oder mehrere Referenzen auf alle benötigten und mit dem Zustand assoziierten Objekte. Dies sind in der aktuellen Implementierung nur Referenzen auf `AstNodes`. Es ist aber durchaus denkbar, weitere Informationen, Objekte, oder Referenzen (z.B. Bindings) in einem Zustand zu speichern. In der Beschreibung von Tabelle 10 werden neben der `AstNode`-Referenz noch der Zustand selber und die (Vor-)Bedingung zum Erreichen dieses Zustandes spezifiziert.

Zustand	Beschreibung
1. Initiale Zuständen	
RoleTypeDeclaration	**Referenz:** Enthält eine Referenz auf eine Rollenklasse (=MemberTypeDeclaration), wie sie vom Parser inital im AST erzeugt wird. **Zustand:** Für die Rollenklasse gilt: isRole()==true. Klasse ist vollständig als Quellcode verfügbar. **Bedingung:** Compilerphase 1.1 für aktuelle CompilationUnit abgeschlossen
RoleMethodDeclaration	**Referenz:** Enthält eine Referenz auf eine Rollenmethode (=MethodDeclaration), wie sie vom Parser initial erzeugt wird, und auf die zugehörige Rollenklasse. **Zustand:** Methode ist vollständig als Quellcode verfügbar. **Bedingung:** Compilerphase 1.1 für aktuelle CompilationUnit abgeschlossen
RoleConstructorDeclaration	**Referenz:** Enthält eine Referenz auf einen Rollenkonstruktor (=ConstructorDeclaration), wie er vom Parser initial erzeugt wird, und auf die zugehörige Rollenklasse. **Zustand:** Konstruktor ist vollständig als AST verfügbar. **Bedingung:** Compilerphase 1.1 für aktuelle CompilationUnit abgeschlossen
TeamTypeDeclaration	Enthält eine Referenz auf eine Teamklasse (=TypeDeclaration), wie sie vom Parser inital im AST erzeugt wird. **Zustand:** Für eine Teamklasse gilt: isTeam()==true. Team ist vollständig als AST verfügbar. **Bedingung:** Compilerphase 1.1 für aktuelle CompilationUnit abgeschlossen
TeamMethodDeclaration	**Referenz:** Enthält eine Referenz auf eine Teammethode (=MethodDeclaration), wie sie vom Parser initial erzeugt wird, und auf die zugehörige Teamklasse. **Zustand:** Methode ist vollständig als AST verfügbar. **Bedingung:** Compilerphase 1.1 für aktuelle CompilationUnit abgeschlossen
TeamConstructorDeclaration	**Referenz:** Enthält eine Referenz auf einen Teamkonstruktor (=ConstructorDeclaration), wie er vom Parser initial erzeugt wird, und auf die zugehörige Teamklasse. **Zustand:** Konstruktor ist vollständig als AST verfügbar. **Bedingung:** Compilerphase 1.1 für aktuelle CompilationUnit abgeschlossen

Zustand	Beschreibung
2. Transiente Zustände	
CreatedMarkerInterface	**Referenz:** Enthält eine Referenz auf das erzeugte MarkerInterface (=MemberTypeDeclaration). **Zustand:** MarkerInterface ist vollständig als AST verfügbar. **Bedingung:** Compilerphase 1.4 für aktuelle CompilationUnit abgeschlossen
CreatedClassPart	**Referenz:** Enthält eine Referenz auf den Klassenteil (=MemberTypeDeclaration) einer im Rahmen von Implicit-Inheritance gesplitteten Rollenklasse. **Zustand:** Erzeugte Klasse ist vollständig als AST verfügbar. **Bedingung:** Compilerphase 1.1 für aktuelle CompilationUnit abgeschlossen
CreatedInterfacePart	**Referenz:** Enthält eine Referenz auf den Interfaceteil (=MemberTypeDeclaration) einer im Rahmen von Implicit-Inheritance gesplitteten Rollenklasse. **Zustand:** Erzeugtes Interface ist vollständig als AST verfügbar. **Bedingung:** Compilerphase 1.1 für aktuelle CompilationUnit abgeschlossen
CreatedInterfaceInheritance	**Referenz:** Enthält eine Referenz auf den Klassenteil (=MemberTypeDeclaration) einer im Rahmen von Implicit-Inheritance gesplitteten Rollenklasse. **Zustand:** Die Verbindung der Klasse zu ihrem zugehörigen Interface wurde hergestellt. **Bedingung:** Compilerphase 1.4 für aktuelle CompilationUnit abgeschlossen
CreatedImplicitInheritance	**Referenz:** Enthält eine Referenz auf den Interfaceteil (=MemberTypeDeclaration) einer im Rahmen von Implicit-Inheritance gesplitteten Rollenklasse. **Zustand:** Die Verbindung des Interfaces zu ihrem gleichnamigen im Superteam definierten Interface wurde hergestellt. **Bedingung:** Compilerphase 1.4 für aktuelle CompilationUnit abgeschlossen

Zustand	Beschreibung
InheritedClass	**Referenz:** Enthält eine Referenz auf einen im Rahmen von Copy-Inheritance geerbten Klassenteil (=MemberTypeDeclaration). **Zustand:** Die vom Superteam geerbte Klasse ist als AST verfügbar. Sie enthält noch keine Methoden oder Felder. **Bedingung:** Compilerphase 1.4 für aktuelle CompilationUnit abgeschlossen
InheritedInterface	**Referenz:** Enthält eine Referenz auf einen im Rahmen von Copy-Inheritance geerbten Interfaceteil (=MemberTypeDeclaration). **Zustand:** Das vom Superteam geerbte Interface ist als AST verfügbar. Es enthält noch keine Methoden oder Felder. **Bedingung:** Compilerphase 1.4 für aktuelle CompilationUnit abgeschlossen
InheritedField	**Referenz:** Enthält eine Referenz auf ein im Rahmen von Copy-Inheritance geerbtes Feld (FieldDeclaration). **Zustand:** Das von der Superrolle geerbte Feld ist als AST verfügbar. **Bedingung:** Compilerphase 2.1 für aktuelle CompilationUnit abgeschlossen
InheritedMethod	**Referenz:** Enthält eine Referenz auf eine im Rahmen von Copy-Inheritance geerbte Methode (MethodDeclaration). **Zustand:** Nur Signatur ist im AST verfügbar. Anstelle eines Bodies enthält diese Methode eine Referenz auf den später zu kopierenden Bytecode. **Bedingung:** Compilerphase 2.1 für aktuelle CompilationUnit abgeschlossen
InheritedConstuctor	**Referenz:** Enthält eine Referenz auf einen im Rahmen von Copy-Inheritance geerbten Konstruktor (ConstructorDeclaration). **Zustand:** Nur Signatur ist im AST verfügbar. Anstelle eines Bodies enthält dieser Konstruktor eine Referenz auf den später zu kopierenden Bytecode. **Bedingung:** Compilerphase 2.1 für aktuelle CompilationUnit abgeschlossen

Zustand	Beschreibung
CreateConstructorMethod	**Referenz:** Enthält eine Referenz auf eine im Rahmen von Implicit-Inheritance erzeugte KonstruktorMethode (=MethodDeclaration). **Zustand:** Diese Methode hat einen bereits vordefinierten Body und ist vollständig als AST verfügbar. **Bedingung:** Compilerphase 2.1 für aktuelle CompilationUnit abgeschlossen
TSuperCall	**Referenz:** Enthält eine Referenz auf einen tsuper Aufruf (=TsuperReference) innerhalb einer Rollenmethode oder eines Rollenkonstruktors. **Zustand:** Statement ist vollständig als AST verfügbar **Bedingung:** Compilerphase 2.1 für aktuelle CompilationUnit abgeschlossen
ConstructorInvocation	**Referenz:** Enthält eine Referenz auf einen Konstruktoraufruf (=AllocationExpression) für eine Rollenklasse, wie er innerhalb jeder Methode oder innerhalb jedes Konstruktors eines Team oder einer Rolle vorkommen kann. **Zustand:** Statement ist vollständig als AST verfügbar **Bedingung:** Compilerphase 2.1 für aktuelle CompilationUnit abgeschlossen
TranslatedTSuperCall	**Referenz:** Enthält eine Referenz auf den transformierten tsuper-Aufruf. **Zustand:** Liste der Parameter im tsuper-Aufruf wurde mit nach `null` gecastetem Markerinterface erweitert und ist vollständig als AST verfügbar. **Bedingung:** Compilerphase 2.1 für aktuelle CompilationUnit abgeschlossen
TranslatedConstructor-Invocation	**Referenz:** Enthält eine Referenz auf den erzeugten Konstruktormethodenaufruf (=CastExpression). **Zustand:** Der AST enthält jetzt die CastExpression mit dem Aufruf der Konstruktormethode. **Bedingung:** Compilerphase 2.1 für aktuelle CompilationUnit abgeschlossen
3. Finale Zustände	
EndState	**Referenz:** Enhält keine Referenz und entfernt sich selbst aus der Menge aller vom Zustandsautomaten zu verarbeitenden Zustände. **Zustand:** Endzustand **Bedingung:** keine

Tabelle 10

5.3.1.2 Implementierung des Zustandsautomaten

Abbildung 23 stellt das Klassendiagramm des in den Compiler integrierten Zustandsautomaten dar. Der als Singleton [16] implementierte Zustandsautomat (StateMachine) verwaltet für jede `CompilationUnitDeclaration` eine Menge von Zuständen (`States`), deren Schnittstellen durch das Interface `IState` festgelegt sind. Intern verwendet der Zustandsautomat eine Hash-Tabelle, um eine `CompilationUnitDeclaration` auf eine Liste von Zuständen (`States`) zu mappen. Auf diese Weise kann innerhalb des Zustandsautomaten für jede `CompilationUnitDeclaration` eine eigene Menge von Zuständen verwaltet werden. Erzeugt werden die initialen Zustände aus der `CompilationUnitDeclaration` mit Hilfe eines AST-Visitors (`StatesCreatorVisitor`) [16] in Kombination mit den in der Klasse `StateFactory` zur Verfügung gestellten Factory-Methoden [16].

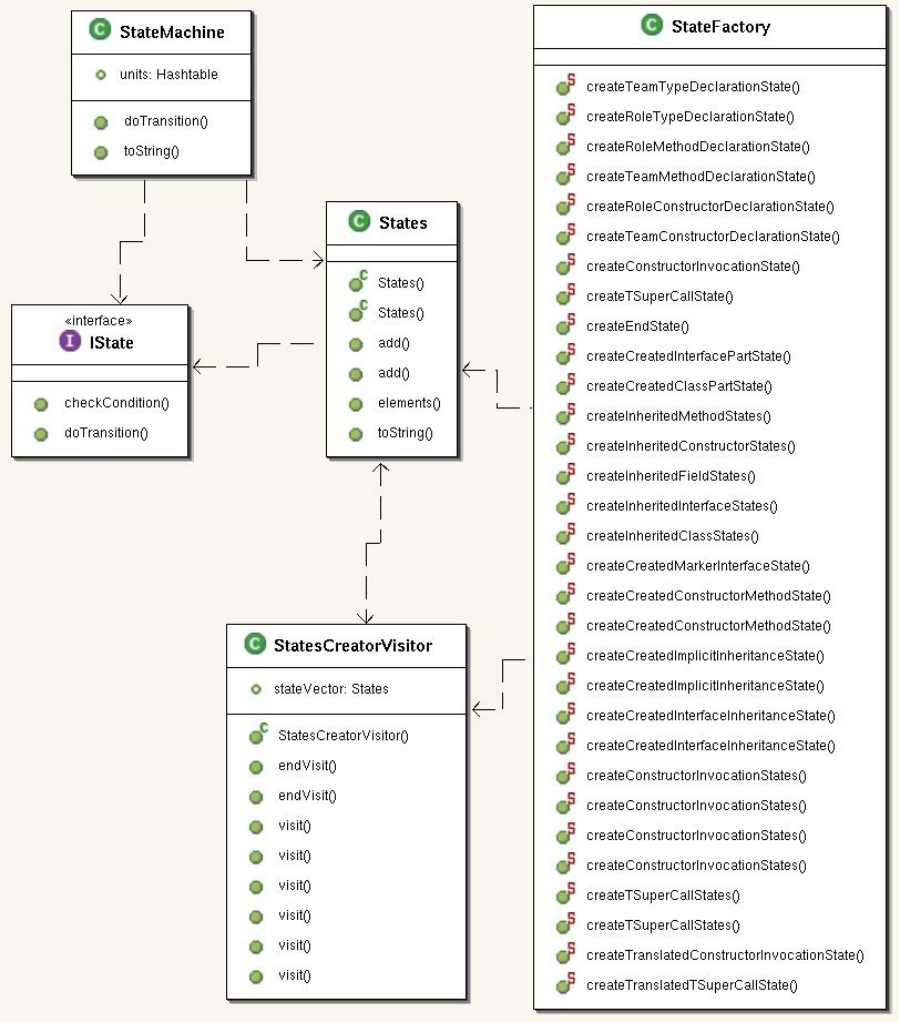


Abbildung 23 Zustandsautomat

Der Zustandsautomat muss während des Kompilierungsvorganges zwischen mehreren Phasen die Möglichkeit bekommen, seine Arbeit verrichten zu können. Deshalb wird nach Phase 1.1, nach Phase 1.4 und nach Phase 2.1 jeweils `StateMachine.doTransition()` aufgerufen. Der Zustandsautomat überprüft jetzt für die zur aktuellen `CompilationUnitDeclaration` assoziierten Zustände (`States`), ob deren Bedingung

für eine Transformation erfüllt ist (`IState.checkCondition()`). Wenn ja, wird für diesen Zustand `IState.doTransition()` ausgeführt. Die Implementierung von `IState.doTransition()` muss jetzt dafür sorgen, dass der Zustand in der Menge der zu transformierenden Zustände (`States`) durch sämtliche seiner erzeugten Nachfolgezustände ersetzt wird. Da diese neuen Zustände ebenfalls eine Bedingung haben, die bereits erfüllt sein könnte, wird in einem rekursiven Prozess `StateMachine.doTransition()` solange aufgerufen, bis für alle Zustände `IState.checkCondition()` `false` liefert, also keine Zustände mehr transformierbar sind. Eine Besonderheit an der implementierten Rekursion von `StateMachine.doTransition()` ist, dass in jedem Rekursionsschritt nur eine Teilmenge aller Zustände verarbeitet werden muss. Dies wird erreicht, indem an einen weiteren Rekursionsaufruf von `StateMachine.doTransition()` keine Zustände übergeben werden, für die `IState.checkCondition()` bereits einmal nicht erfüllt war. Dadurch wird erreicht, dass nur potentiell transformierbare Zustände überprüft werden und diese Überprüfung für jeden Zustand nur genau einmal pro Transformationsschritt (z.B. von State 1 nach State 2) stattfindet.

5.3.1.3 Implementierung der Zustände

Jeder Zustand wird durch eine `IState` implementierende Klasse repräsentiert. Als Beispiel für eine solche Implementierung und für das Zusammenspiel mit der `StateMachine` soll hier die Klasse `RoleTypeDeclarationState` erläutert werden. Die Klasse wird durch den obengenannten `StatesCreatorVisitor` instanziiert und als Objekt der Menge von Zuständen hinzugefügt.

```
01public class RoleTypeDeclarationState extends AbstractTypeDeclarationState
02{
03
04  /**
05   * Constructor for RoleTypeDeclarationState
06   * @param states set of states where this state is an element
07   * @param roleType the corresponding Role-MemberTypeDeclaration
08   */
09  public RoleTypeDeclarationState(States states, TypeDeclaration roleType){
10    super(states,roleType);
11  }
12
13  /* (non-Javadoc)
14   * @see IState#doTransition()
15   */
16  public void doTransition(){
17    states.remove(this);
18    states.add(createCreatedInterfacePartState());
19    states.add(createCreatedClassPartState());
20  }
21
22  /* (non-Javadoc)
23   * @see IState#checkCondition()
24   */
25  public boolean checkCondition(int statecontext){
26    return statecontext>=PHASE_1_1);
27  }
```

```
28
29  /* (non-Javadoc)
30  * @see Object#toString()
31  */
32 public String toString(){
33    String str;
34    str="RoleTypeDeclarationState\n";
35    str+=super.toString()+"\n";
36    return str;
37  }
38
39  /**
40   * this method is only to visualize the connection between the States
41   * there is no other meaning for the existence of this delegation methods
42   */
43  private CreatedClassPartState createCreatedClassPartState(){
44    return StateFactory.createCreatedClassPartState(this);
45  }
46
47  /**
48   * this method is only to visualize the connection between the States
49   * there is no other meaning for the existence of this delegation methods
50   */
51  private CreatedInterfacePartState createCreatedInterfacePartState(){
52    return StateFactory.createCreatedInterfacePartState(this);
53  }
54}
```

Beispiel 28

Die implementierte Methode `checkCondition()` überprüft, ob Phase 1.1 bereits abgeschlossen ist und liefert einen entsprechenden Booleschen Wert zurück. Die `StateMachine` wertet das Ergebnis von `checkCondition()` aus und ruft im Falle von `true` `doTransition()` auf und erzeugt sofort sämtliche Nachfolgezustände. Sofort bedeutet, dass die Vorbedingung für deren Erzeugung bereits erfüllt sein muss. Deshalb sei hier noch einmal erwähnt, dass die innerhalb von `checkCondition()` geprüfte Bedingung eine Konjunktion aller in obiger Tabelle definierten Bedingungen der Nachfolgezustände sein muss. In diesem konkreten Fall heißt das: Die aus der Tabelle ablesbare Bedingung von `CreatedClassPartState` ist, dass Phase 1.1 abgeschlossen sein muss, und die von `CreatedInterfacePartState` ebenfalls, dass Phase 1.1 abgeschlossen sein muss. Die Konjunktion beider Nachfolgezustände ist, dass Phase 1.1 abgeschlossen sein muss. Alternativ wäre folgende Implementierung anstelle der Zeilen 25 bis 27 möglich:

```
01  public static boolean checkPrecondition(int phase){
02    return statecontext>=PHASE_1_1;
03  }
04
05  public boolean checkCondition(int phase) {
06    return   CreatedClassPartState.checkPrecondition(statecontext) &&
07          CreatedInterfacePartState.checkPrecondition(statecontext));
08  }
```

Beispiel 29

Die beiden privaten Methoden haben mehr einen praktischen Grund. Es soll eine sichtbare Verbindung innerhalb einer Klasse zu den Nachfolgezuständen geben. Dies hängt damit zusammen, dass das Eclipse-UML-Plugins Omondo [23] Abhängigkeiten zwischen mehreren Klassen innerhalb eines Klassendiagramms nur anzeigt, wenn diese auf Signaturebene referenziert werden. Beim späteren Einfügen weiterer Zustände kann so visuell deren korrekte Verbindung zu anderen Zuständen überprüft werden.

5.3.2 Implementierung der Bytecodecopy/transformation

Durch die Copy-Inheritance AST-Transformation werden Methoden erzeugt, welche anstelle von Statements, eine Referenz auf die zu kopierende Methode enthalten. Diese als Quelle dienende Methode (`MethodBinding`) muss eine Referenz auf den zu kopierenden und zu transformierenden Bytecode haben. Die Transformation wird in der `generateCode()` einer `MethodDeclaration/ConstructorDeclaration` durchgeführt.

Vorbedingung für die Bytecodetransformation ist, dass für jede zu transformierende Constant-Pool-Referenz, der aus dieser Transformation resultierende Typ bereits im AST und als Binding existiert. Für elementare Datentypen wie int oder String oder für Referenzen die nicht transformiert werden, gilt diese Vorbedingung nicht.

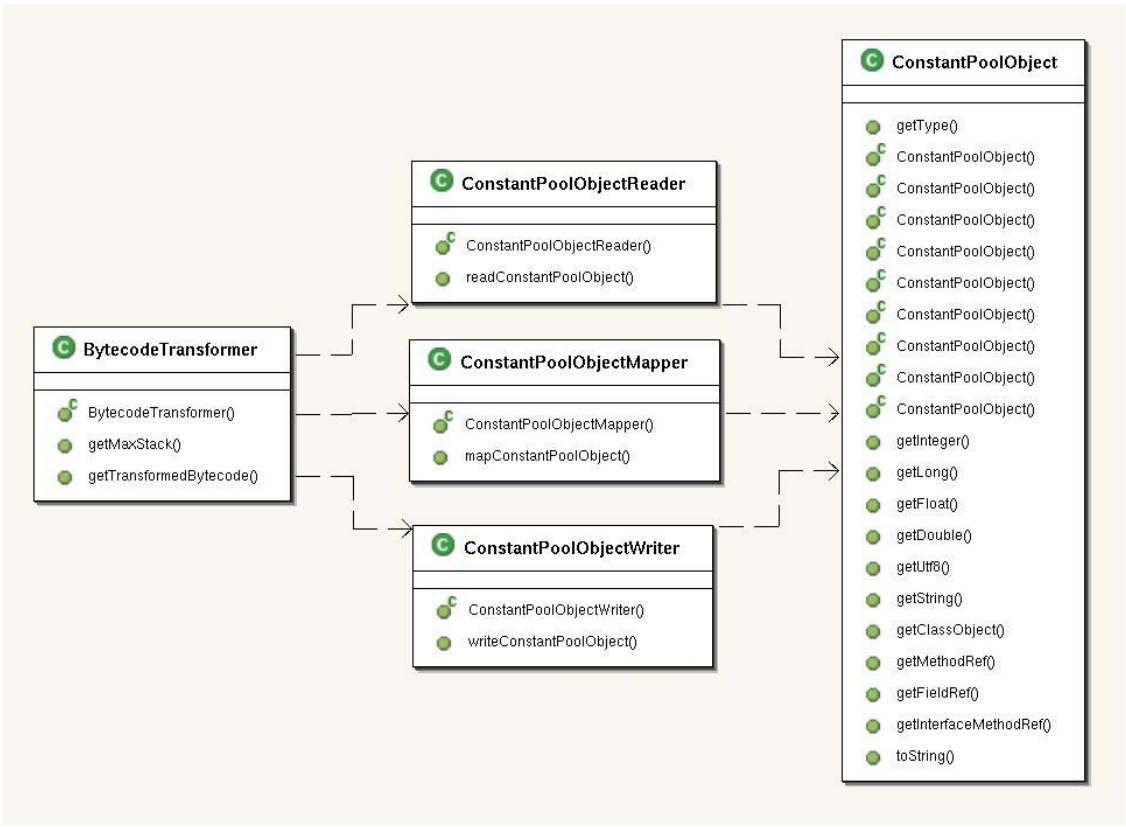


Abbildung 24 Bytecode Transformer

Abbildung 24 zeigt die zentralen für Bytecodecopy verwendeten Klassen. Im folgenden soll mit Hilfe des Klassendiagramms das Prinzip der implementieren Bytecodetransformation erklärt werden.

Innerhalb der Methode `generateCode()` wird anstelle der Bytecode-Generierung aus Statements, eine auf Bytecodecopy basierende Bytecode-Generierung durchgeführt. Sämtliche für diese Generierung notwendigen Informationen (auch den transformierten Bytecode) liefert der `BytecodeTransformer`. Der `BytecodeTransformer` holt sich den zu kopierenden Bytecode und den zugehörigen Constant-Pool (ebenfalls nur in Form von Bytecode). Dann werden alle im Bytecode enthaltenen Referenzen (`int`-Wert) auf den Constant-Pool herausgesucht. Für jede im Bytecode gefundene Referenz wird mit dem `ConstantPoolObjectReader` das referenzierte `ConstantPoolObject` instanziiert. Dieses `ConstantPoolObject` stellt einen Wrapper für alle möglichen Datentypen (`MethodBinding`, `FieldBinding`, `String`, `int`, usw) dar, welche durch diese Referenz referenziert werden können. Die Liste aller möglichen Datentypen ist in Abschnitt 3.2.1 definiert. Der `ConstantPoolObjectMapper` führt jetzt nach den in Abschnitt 4.1.6 beschriebenen Regeln das Mapping aller Referenzen vom Super-Team in das aktuelle Sub-Team durch und liefert als Ergebnis ein transformiertes `ConstanPoolObject` zurück. Dieses neue `ConstantPoolObject` wird jetzt mit dem `ConstantPoolObjectWriter` in den Ziel-Constant-Pool geschrieben (`writeConstantPoolObject`). Als Rückgabe liefert die Methode die neue Referenz auf den zuvor erzeugten Constant-Pool-Eintrag zurück. Als letzter Schritt wird die zu transformierende Referenz im Bytecode durch die neue Referenz ersetzt.

5.3.3 Generisches Austauschen von AstNodes

Während der AST-Transformation müssen einige `AstNodes` ausgetauscht werden. Dieses betrifft z.B. die Konstruktoraufrufe (`new Role(args)`) welche im Rahmen von Copy-Inheritance gegen Konstruktor-Methodenaufrufe (`_OT$createRole(args)`) ausgetauscht werden müssen. Um dieses so wenig wie möglich invasiv zu realisieren, bedarf es einem speziellen Mechanismus. Der Grund dafür ist, dass der AST in Eclipse einen Baum darstellt, in dem zwar jeder `AstNode` seine sämtlichen Knoten und Blätter (Childs) kennt, nicht aber seinen Container (Parent). Ohne den Parent zu kennen, kann jedoch in Java kein Austausch durchgeführt werden. Der erste für das Austauschen notwendige Schritt, ist also die Suche nach dem umschließenden Parent-`AstNode`. Diese Suche wird über einen `AstNodeParentSearchVisitor` erledigt. Dieser traversiert den AST und speichert alle `AstNodes` in einem Stack. Sobald der Visitor den auszutauschenden Child-AstNode erreicht, wird der Parent-`AstNode` aus dem Stack geholt, und die Suche beendet.
Aber auch mit Hilfe dieses Parents ist es immer noch nicht leicht, einen Austausch vorzunehmen. Denn jeder `AstNode` hat eine unterschiedliche interne Struktur, außerdem enthält ein `AstNode` in der vorliegenden Implementierung keine Funktionalität, um alle enthaltenden Childs zu iterieren. Dadurch wäre für jeden auszutauschenden `AstNode`-Typen, eine Sonderbehandlung für alle potentiell in Frage kommenden Parents notwendig. Da dies im vorliegenden AST recht viele sein können, wurde nach einem Mechanismus gesucht, diesen Austausch ohne großen Implementierungsaufwand durchführen zu können. Die Lösung des Problems wird über Reflections (`java.lang.Reflec`) erreicht. Reflections ermöglichen einen Zugriff auf alle Attribute (und die darin enthaltenden Objektinstanzen) einer Klasse in generischer Weise. Dadurch kann die gesuchte `AstNode`-Instanz gefunden und ausgetauscht werden.

5.3.4 Implementierung von Translation Polymorphism

5.3.4.1 LiftingEnvironment

Das `LiftingEnvironment` stellt Funktionalität zur Verfügung, um die für das Smart Lifting benötigten Rollen- und Basis-Vererbungshierarchien aufzubauen. Ausgehend von diesen erzeugten Hierarchien wird dann für jede gebundene Rollenklasse die benötigte LiftTo-Methode sowie der notwendige Rollenkonstruktor im AST generiert.

5.3.4.2 LiftTo-Methode

Die während der Compiler-Transformation durch das `LiftingEnvironment` generierte LiftTo-Methoden stellen zur Laufzeit Funktionalität zur Verfügung, um ein Mapping von Basisobjekten auf Rollenobjekte durchzuführen (`RoleType r = liftToRole(BaseType b)`). Da ein gleiches Mapping immer auf eine identische Objektidentität hinauslaufen muss, wird das innerhalb dieser Methode erzeugte Rollenobjekt in einer HashMap (`java.util.WeakHashMap`) gespeichert und bei einem folgenden Aufruf der LiftTo-Methode aus diesem wieder ausgelesen. Falls noch kein Eintrag in der HashMap existiert wird die eigentliche Erzeugung des Rollenobjektes innerhalb einer `switch/case`-Anweisung durchgeführt. Diese `switch/case`-Anweisung entscheidet in Abhängigkeit von einem Integer-Wert (dem *Base-Tag*), welche Rollenklasse für das übergebene Basisobjekt instanziiert und zurückgeliefert werden muss. Nicht unterstützte `switch`-Werte resultieren in einer `LiftingFailedException`. Das Base-Tag wird zur Laufzeit aus einem Feld der übergebenen Basis ausgelesen. Spannend an der Sache ist, dass dieses Feld während des Kompilierungsvorganges noch gar nicht in der Basis existiert. Dieses Feld wird vom ObjectTeams-RE erst unmittelbar vor dem Lifting in das Basisobjekt hineingewebt. Für den ObjectTeams/Java-Compiler bedeutet diese Vorgehensweise, dass Bytecode und eine Constant-Pool-Referenz für eine Referenz auf ein Attribut erzeugt werden muss, welches nicht existiert. Erreicht wird dies durch ein Base-Tag-Fake.

5.3.4.3 Base-Tag-Fake

Anstelle einer üblichen `QualifiedNameReference` wird im AST eine spezielle Base-Tag-Referenz (`QualifiedFakedBaseReference`) generiert. Dieses ist notwendig, da in Phase 2.4 (resolve) für eine `QualifiedNameReference` überprüft wird, ob das referenzierte Ziel existiert. Sollte dies nicht der Fall sein, wird vom Compiler eine Fehlermeldung ausgegeben. Da für das referenzierte Base-Tag diese Überprüfung nicht durchgeführt werden darf, ist in der `QualifiedFakedBaseReference` dieser Mechanismus entfernt worden. Zusätzlich wird innerhalb dieser Referenz das für die Bytecodegenerierung und für den Constant-Pool-Eintrag benötigte Binding erzeugt.

5.3.4.4 Rollenkonstruktoren

Für jede gebundene Rollenklasse wird zusätzlich zur LiftTo-Methode ein Konstruktor der Form (`RoleName(BaseType b)`) erzeugt. Dieser Konstruktor fügt das übergebene Basisobjekt in einen Cache ein. Außerdem wird eine Referenz auf die übergebene Basis im Team gespeichert.

6 Zusammenfassung

6.1 Fazit

Erst während der praktischen Arbeit am System entsteht intuitives Wissen über die Vorgänge in einem Softwaresystem. Querverbindungen und Seiteneffekte können mit Hilfe dieses Wissens schneller erfasst werden. Fehlerquellen werden erahnt und fehlerhafte Stellen im System können schneller aufgefunden werden. Da sich dieses Wissen jedoch erst im Laufe des Implementierungsprozesses entwickelt, müssen bereits durchgeführte Arbeiten immer wieder durchdacht werden. Nicht selten ergaben sich daraus notwendige Korrekturarbeiten. Ein positiver Aspekt ergab sich aus dem Spiegel, der einem durch eigens implementierte und auch eigens dokumentierte Programmfunktionalität vorgehalten wurde. Deshalb wurden Korrekturarbeiten oftmals begleitet von Refactoring und dem Erweitern vorhandener Dokumentation.

Ein weiterer „Entwicklungszyklus“ ergab sich durch die Niederschrift der Entwürfe und Implementierungen in dieser Diplomarbeit. Beim Schreiben tauchten Fragen und Probleme auf, die bisher nicht beantwortet und somit in der Software nicht berücksichtigt wurden. Teilweise konnten diese Probleme noch gelöst werden, zogen dann aber eine Korrektur bis dahin entstandener Texte mit sich. Zusammenfassend wurden folgende Arbeiten erledigt.

6.1.1 Erledigte Arbeiten

Grammatik

Die ObjectTeams Grammatik wurde vollständig umgesetzt. Der aus dieser Grammatik erzeugte Parser/Scanner ist vollständig implementiert und erzeugt aus einer ObjectTeams-Quelldatei einen um ObjectTeams-Elemente erweiterten AST.

Copy-Inheritance

Der ObjectTeams-Compiler kopiert sämtliche Rollen aus dem Super-Team ins Sub-Team. Auf Signaturebene wird dies vollständig durchgeführt. Berücksichtigt wird dabei die Erweiterung der Signaturen um Marker-Argumente. Rollenkonstruktoren-Methoden werden auf Team-Ebenen für jeden Rollenkonstruktor erzeugt. Im AST werden sämtliche Rollen-Instanziierungsaufrufe durch den Aufruf dieser Rollenkonstruktor-Methoden ersetzt. Das Splitting der Rollenklassen wird vollständig durchgeführt. Gesplittete Rollenklassen werden ebenfalls mit geerbten Methoden erweitert. Die für Copy-Inheritance notwendigen Vererbungsbeziehungen zwischen den Rolleninterfaces und Rollenklassen werden erzeugt. Aufrufe von `tsuper` werden korrekt transformiert. Ein Team erbt implizit immer von `org.objectteams.Team`, falls keine Superklasse

angegeben ist. Das Within-Statement wird korrekt transformiert. Der Bytecode von implizit geerbten Rollenmethoden wird korrekt kopiert.

Lifting

Die für Lifting notwendigen Hierarchien für Rollenklassen und Basisklassen werden zur Laufzeit des Compilers vollständig aufgebaut (ohne Berücksichtigung von Externalized Roles). Es werden für die an eine Basis gebundenen Rollenklassen, LiftTo-Methoden und die zugehörigen Konstruktoren erzeugt. Das innerhalb der LiftTo-Methoden referenzierte Base-Tag wird ohne Fehlermeldung des Compilers als Bytecode erzeugt.

Neben den Erledigten Arbeiten gibt es noch eine Reihe offener Arbeiten am Compiler. Die im folgenden Abschnitt dokumentiert werden.

6.1.2 Offene Arbeiten

Die im Rahmen dieser Diplomarbeit durchgeführte Implementierung stellt eine Grundlage für die vollständige Implementierung des Eclipse-ObjectTeams/Java-Compilers dar. Im Quellcode wurden an unterschiedlichen Stellen Hinweise über noch ausstehende Arbeiten eingetragen. Diese in Eclipse als Task sichtbaren TODOs müssen geprüft und implementiert werden. Dabei handelt es sich teilweise um Aufgaben, die analog zu bereits Implementiertem durchgeführt werden können, und kein oder nur wenig intuitives Wissen erfordern. Aber auch bewusst ausgelassene, unfertige oder fehlerhafte Stellen wurden auf diese Weise dokumentiert. Als Beispiel sei hier die Implementierung von Bytecodecopy für Rollenkonstruktoren analog zu Bytecodecopy von Rollenmethoden genannt. Neben diesen kleineren Programmieraufgaben gibt es jedoch noch größere Aufgaben die nur mit entsprechendem Fachwissen ausgeführt werden können. Diese offene Aufgaben sollen im Folgenden kurz skizziert werden.

Hierarchieanalyse

Die Implementierung der unter [14] beschriebenen Hierarchieanalyse bezüglich Smart-Lifting steht noch aus.

Parameter-Mapping

Die Implementierung von Parametermapping für Callin- und Callout-Bindings ist noch nicht konzeptionell erarbeitet, kann aber Ansatzweise in der Diplomarbeit von C.Binder [20] nachgelesen werden.

Deklariertes Lifting

Die Implementierung und konzeptionelle Erarbeitung von deklariertem Lifting ist noch offen. Deklariertes Lifting stellt eine Möglichkeit dar, den Lifting-Prozess explizit auszulösen. Team-Methoden können spezielle Parameter mit sowohl einem Basistypen als auch einem Rollentyp deklarieren. Die Syntax für diesen Lifting-Type lautet `BaseType`

`as RoleType`. Durch dieses deklarierte Lifting ist es möglich, eine Team-Methode, welche intern ein Rollenobjekt vom Typ `RoleType` erwartet, von ausserhalb des Teams aufzurufen, indem der Methode ein Objekt vom Typ `BaseType` übergeben wird.

Ersetzung der base-calls

Base-calls müssen vom Compiler transformiert werden. Die dafür notwendige Regel lautet: wenn eine mittels `callin`-Modifier markierte Rollen-Methode einen base-call der Form `base(args)` enthält, und die umschließende Rolle an eine Basis gebunden ist, dann muss der base-call in einen rekursiven Aufruf der Rollen-Methoden umgewandelt werden. Diese Rekursion wird später vom ObjectTeams-RE durch einen Basis-Methodenaufruf ersetzt. Alternativ dazu könnte der Aufruf der nicht existenten Basis-Methode analog zum Base-Tag gefaked werden.

Fehlermeldungen

Aussagekräftige Fehlermeldungen müssen vom Compiler erzeugt werden. Der Mechanismus zur Erzeugung von Fehlermeldungen ist leicht verständlich und wird zur Erzeugung einfachen Fehlermeldungen bereits verwendet. Das `within`-Statement gibt eine Fehlermeldung aus, wenn das übergebene Objekt keine Team-Instanz ist. Eine Superklasse für ein Team wird nur akzeptiert, wenn die Superklasse ebenfalls ein Team ist. Das Werfen einer Fehlermeldung geschieht über einen `ProblemReporter`, welcher über den Scope erreichbar ist. Als Beispiel sei hier die Erzeugung einer Fehlermeldung in der Klasse `WithinStatement` für einen oben beschriebenen Fehler gezeigt:

```
scope.problemReporter().needTeamInstance(condition);
```

Externalized Roles

Externalized Roles werden in der Implementierung noch nicht berücksichtigt. Zu berücksichtigen sind Externalized Roles während des Aufbaus der Hierarchie. Intern wird für Rollen ein spezieller nicht explizit setzbarer Role-Modifier verwendet. Dieser wird vom Parser automatisch für Rollenklassen gesetzt. Dieser Modifier muss ebenfalls für Externalized Roles gesetzt werden.

6.1.3 Optimierungen

Parser

Der Eclipse-Compiler hat einen Fehler beim Einlesen von Binaries, welche ein „$“ enthalten. Dieser Bug wurde zwar für ObjectTeams-spezifische Namen behoben, gehört aber eigentlich auf die Seite der Eclipse-Bugliste. Das Problem basiert auf einem zu einfachen Mechanismus zur Generierung des internen Namens für innere Klassen. Der interne Name ist in Eclipse immer der nach dem letzten „$“ folgende Rest.

Visitor

Des weiteren gibt es im Eclipse-Compiler eine AST-Visitor Implementierung die vermutlich nachträglich unsauber optimiert wurde. Eclipse implementiert innerhalb jedes `AstNodes` eine rekursive `traverse()`-Methode, in welcher der Visitor aufgerufen wird. Für diese Implementierung gibt es einen existierenden Mechanismus zum Verhindern weiterer Rekursionen. Offensichtlich war das aber demjenigen nicht klar. Denn er implementierte zusätzlich einen zusätzlichen Mechanismus zum Verhindern der Traversierung (`ignoreFurtherInvestigation`). Die Traversierung wird z.B. verhindert, wenn bereits einmal ein Compiler-Fehler aufgetreten ist. Dies sollte nicht sein. Die Traversierungen sollten immer ausgeführt werden sofern und soweit der Visitor dieses wünscht. Ein daraus resultierendes Problem war ein schwer auffindbarer Fehler, welcher durch einen Seiteneffekt hervorgerufen wurde. Nach dem Einfügen eines expliziten Role-Modifiers durch den Parser und vor allem durch eine dritte Person (ohne die dritte Person hiermit kritisieren zu wollen), funktionierte das während der Compilertransformation notwendige Auswechseln eines `AstNodes` innerhalb eines Statements nicht mehr. Für das generische Austauschen wird ein Visitor verwendet, welcher zu einem `AstNode` seinen Parent findet. Der Visitor erreichte einfach die Stelle im AST nicht mehr und konnte daraufhin auch keinen Austausch vornehmen. Da ein Visitor im Kontrollfluss schwer zu verfolgen ist, dauerte es eine Weile, bis herausgefunden war, an welcher Stelle und warum der Visitor hier abbrach. Aber damit war die Ursache noch nicht gefunden. Fest Stand nur, dass `ignoreFurtherInvestigation` auf true gesetzt war. Die Suche nach Schreibzugriffen auf diese Variable (Eclipse sei Dank) lieferte einen Zugriff, wenn der Modifier einer inneren Klasse nicht korrekt ist. Aber die Rollenklasse hatte im Sourcecode doch keinen sichtbar falschen Modifier, sondern war nur als „`public class Role {...}`“ definiert. Nach einer Analyse der innerhalb der Klasse gesetzten Modifier (Bitfeld) stellte sich heraus, dass ein Role-Modfier gesetzt war. Erst jetzt fiel es mir wie Schuppen von den Augen, dass genau dieses Vorhaben bereits mit mir persönlich besprochen war. Der Fehler war jetzt schnell gefunden und beseitigt.

Speicher

Für die Realisierung von Bytecodecopy muss eine Referenz auf den zu kopierenden Bytecode gespeichert werden. Dieses geschieht momentan innerhalb eines MethodBindings. Diese Referenz benötigt Speicher und sollte deshalb sobald wie möglich freigegeben werden. Das gleiche Problem betrifft einen verwendeten Scope.

AstNodes

Die im Kontext von Copy-Inheritance erzeugten Methoden, für die zur Codegenerierungsphase Bytecode kopiert werden muss, sollten ein eigenes von `MethodDeclaration` oder `ConstructorDeclaration` erbendes AST-Element bekommen. Dadurch wäre der Eingriff in vorhandene Eclipse-Strukturen weniger invasiv. Fallunterscheidungen könnten dann aufgrund der Polymorphie während der Codegenerierung entfallen.

6.2 Resümee der Vorgehensweise

Die im Rahmen dieser Diplomarbeit angewandte Vorgehensweise, stellt einen strukturierten Prozess dar. Zu Beginn des Prozesses wurde dessen genereller Ablauf definiert und anhand der gegebenen Aufgabenstellung verfeinert. Auf diese Weise entstand ein Zeitplan, in welchem die bis dahin ausfindig gemachten Aufgaben und deren Abfolge festgelegt waren. Die Liste der Aufgaben war zu diesem Zeitpunkt jedoch keinesfalls vollständig. Vielmehr enthielt dieser Plan allgemeine in der Softwaretechnik verwendete Vorgänge wie Analyse, Entwurf, Implementierung. Ebenfalls war bereits der zeitliche Rahmen für die Niederschrift der vorliegenden Diplomarbeit abgesteckt. Zusätzlich wurden Meilensteine gesetzt. Der auf diese Weise definierte Plan erwies sich jedoch in der Praxis als zu ungenau und musste im Verlauf des Prozesses mehrmals verfeinert werden. Neben dem üblichen Verfeinerungsprozess durch Zerlegen der Aufgaben in Teilaufgaben, mussten bereits abgeschlossene Phasen erneut durchlaufen werden. Ein Grund für diesen iterativen Prozess war, dass keine strikte Trennung von Analyse, Entwurf und Implementierung erreicht werden konnte. Dies hing damit zusammen, dass das zu verändernde Softwaresystem und die mit der Veränderung im Zusammenhang stehenden Technologien, eine so hohe Komplexität haben, dass es innerhalb einer Diplomarbeit nicht möglich ist, diese vollständig analytisch zu erfassen. Daraus resultiert, dass nur soviel Wissen erarbeitet wurde, wie zur Realisierung (Entwurf/Implementierung) des Vorhabens notwendig war. Vor Beginn einer Realisierung steht jedoch noch gar nicht genau fest, wie viel Wissen das in Wirklichkeit sein wird. Aus diesem Grund wurde vorab geschätzt, welches das minimal benötigte Wissen darstellt, um gezielte und effiziente Analysen durchführen zu können. Tauchten dann zu einem späteren Zeitpunkt während der Realisierung Probleme auf, die mit dem bisherigen Wissen nicht gelöst werden konnten, musste eine weitere gezielte Analyse in den Arbeitsablauf eingeschoben werden.
Dieses iterative Vorgehen war insofern problematisch, da Aufgrund der eingeschobenen Analysephasen, der Entwicklungsprozess unterbrochen und zu einem späteren Zeitpunkt wieder aufgenommen werden musste. Damit der Entwicklungsprozess genau dort wieder aufgenommen werden konnte, wo er unterbrochen wurde, diente eine spezielle Junit-Testklasse als Merkhilfe. Diese Testklasse definierte zu jeder Zeit das gerade zu implementierende ObjectTeams-Sprachfeature und wurde deshalb im Laufe des Entwicklungsprozesses an den jeweils zu implementierenden Concern angepasst.

6.3 Ausblick

Die Entwicklungsumgebung Eclipse bietet aufgrund ihres durchdachten Konzeptes die Möglichkeit, unterschiedliche Werkzeuge zur Softwareentwicklung zu verwenden. Obwohl Eclipse erst seit einigen Jahren existiert, reicht die Palette der existierenden Werkzeuge von Programmiersprachen (z.B. Java), über Werkzeuge zur Qualitätssicherung (z.B. JUnit), bis hin zu vollständigen Modellierungswerkzeugen (z.B. Omondo). All diese Werkzeuge unterstützen den Softwareentwicklungsprozess, wodurch ein enormer Produktivitätsgewinn erreicht wird.

Mit der Integration des ObjectTeams/Java-Compilers für die Eclipse-IDE, wurde die Grundlage für eine vollständige ObjectTeams-Entwicklungsumgebung geschaffen. Bevor jedoch diese Entwicklungsumgebung in der Praxis effizient eingesetzt werden kann, sind noch unterschiedliche Erweiterungen durchzuführen. Eine Anpassung von Outline und Refactoring an das ObjectTeams-Programmiermodell muss erfolgen. Außerdem ist es notwendig, Syntaxhighlighting und CodeCompletion zu unterstützen. Als große Aufgabe ist noch ein Debugger zu realisieren, mit dem es möglich ist, ObjectTeams Programme sinnvoll zu debuggen.

Neben der Fertigstellung des Eclipse-ObjectTeams/Java-Plugins (Eclipse-OTDT), sollte ein weiterer Schwerpunkt die Entwicklung oder Erweiterung von Werkzeugen (Eclipse-Plugins) sein, mit deren Hilfe die Entwicklung aspektorientierter Softwaresysteme unterstützt wird.

In naher Zukunft wird eine vollständige ObjectTeams Entwicklungsumgebung verfügbar sein. Zur Demonstration der Leistungsfähigkeit sollte ein Bootstrapping, als Herausforderung im Rahmen einer Diplomarbeit, in Angriff genommen werden.
Um dafür einen guten Einstieg zu gewährleisten, ist nachfolgend für die Klasse `org.eclipse.jdt.internal.compiler.ast.AstNode` beispielhaft ein Callin-Binding definiert.

Bootstrapping

```
01 class otAstNode implements IConstants playedBy AstNode {
02
03   modifiersString <- replace modifierString;
04
05   callin String modifiersString(int modifiers){
06     String s = base.modifiersString(modifiers);
07     if ((modifiers & AccTeam) != 0)
08       s = s + "team ";
09     if ((modifiers & AccRole) != 0)
10       s = s + "role ";
11     if ((modifiers & AccCallin) != 0)
12       s = s + "callin ";
13     return s;
14   }
15 }
```

Beispiel 30

7 Anhang

7.1 Verwendete Arbeitsmittel

Werkzeuge	**Beschreibung**
Eclipse 2.1.1	IDE für Entwicklung des ObjectTeams/Java-Plugins
Eclipse Sourcen 2.1.1	Eclipse-Quellcode und Grundlage für das ObjectTeams/Java Plugin
Suse 8.1	Das Suse Linux-Betriebssystem wurde für den kompletten Entwicklungsablauf verwendet
Jad / jadclipse	Dekompiliert Bytecode und erzeugt daraus Sourcecode
Eclipse-Omondo	UML-Plugin für Eclipse
Awk / sed / grep	Unix Kommandozeilentools
Jikespg-1.2	Parsergenerator zum übersetzen der ObjectTeams-Grammatik
Listclass / BCEL	Listet basierend auf der BCE-Library Bytecodeinformationen einer Klasse auf
hexdump	Einfacher Hexdump mit dem der reine Bytecode eines Binaries angesehen werden kann.
JUnit	Unit-Test für vorgefertigte Testfällen / Szenarien
Debugger	Java-Debugger zur Zustandsabfrage von Objekten zur Laufzeit
generateOTParser.sh	Script welches die Ersetzungen im ObjectTeams/Java Plugin nach einer Grammatikänderung durchführt
OpenOffice.org_1.1.0	Dokumentationsmittel und Textprogramm für Diplomarbeit
Poseidon UML	Standalone UML-Tool

Tabelle 11

7.2 Abbildungsverzeichnis

Abbildung 1 Compileraufbau [8] 9
Abbildung 2 Suchbaum 11
Abbildung 3 Reduce 13
Abbildung 4 ImplicitInheritance 17
Abbildung 5 Eclipse Perspektive 18
Abbildung 6 Eclipse SDK 19
Abbildung 7 Eclipse PDE 21
Abbildung 8 Extensionpoints 22
Abbildung 9 Hello World View 24
Abbildung 10 JDT-Core 25
Abbildung 11 JavaBuilder 26
Abbildung 12 Source Delta Schema 28
Abbildung 13 IncrementalImageBuilder 28
Abbildung 14 Sequenzdiagramm Compiler 29
Abbildung 15 AST Composite 38
Abbildung 16 Eigene Bindings 40
Abbildung 17 Bindinggraph 40
Abbildung 18 Compiler Phasenmodell 42
Abbildung 19 Phasenablauf 43
Abbildung 20 Copy-Inheritance 52
Abbildung 21 AST Erweiterung 70
Abbildung 22 Zustandsmodell 74
Abbildung 23 Zustandsautomat 80
Abbildung 24 Bytecode Transformer 83

7.3 Quellenverzeichnis

[1] Eclipse
http://www.eclipse.org

[2] Ian Sommerville
Software Engineering
Addison-Wesley 2001

[3] Stanley M. Sutton Jr.* and Isabelle Rouvellou
Modeling of Software Concerns in Cosmos
AOSD 2002

[4] CVS
Concurrent Versions System
http://www.cvshome.org

[5] JUnit
Java Unit Testing
http://www.junit.org

[6] Robert v. Binder
Testing Object-Oriented Systems
Addison-Wesley 2000

[7] Javadoc
Java Documentation
http://java.sun.com/j2se/javadoc

[8] Wilhelm Weissweber
Basis von Programmiersprachen und Systemen
Einführung und Syntax
Vorlesungsscript 2000
http://flp.cs.tu-berlin.de/~ww/lehre/pss

[9] Jikespg
Jikes Parsergenerator
http://www-124.ibm.com/developerworks/oss/jikes

[10] Dick Grune and Ceriel J.H. Jacobs
Parsing Techniques - A Practical Guide, 1990,
http://www.cs.vu.nl/~dick/PTAPG.html

[11] Sean E. O Connor
Review of LR(k) and LALR(k) Parsing Theory
http://www.seanerikoconnor.freeservers.com/ComputerScience/Compiler/
ParserGeneratorAndParser/QuickReviewOfLRandLALRParsingTheory.html

[12] JVM
Java Virtual Machine Specification Second Edition
http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[13] ObjectTeams
http://www.ObjectTeams.org

[14] Stephan Herrmann
Translation Polymorphism in Object Teams
Technische Universität Berlin, 2003
http://www.objectTeams.org
[15] BNF
What is BNF notation?
http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html
[16] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Patterns
Elements of Reusable Object-Oriented Software
Addison Wesley 2002,
[17] listclass
BCEL Example
http://cvs.sourceforge.net/viewcvs.py/bcel/BCEL/listclass.java
[18] BCEL
Byte Code Engineering Library
http://cvs.sourceforge.net/viewcvs.py/bcel/BCEL
[19] Generate Eclipse Java Parser
http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/jdt-core-home/howto/
generate parser/generateParser.html?rev=1.1&content-type=text/html
[20] Christoph Binder
Aspectual Collaborations:
Erweiterung des Java-Compilers für verbesserte Modularität
durch aspekt-orientierte Techniken
Diplomarbeit 2002
http://www.objectTeams.org
[21] JLS
The Java Language Specification
http://java.sun.com/docs/books/jls/first_edition/html/index.html
[22] Jad
Java Decompiler
http://kpdus.tripod.com/jad.html
[23] Omondo
UML Tool für Eclipse
http://www.omondo.com