Diplomarbeit

Entwicklung der Werkzeugunterstützung für die Spracherweiterung Generic Object Teams

Jan Marc Hoffmann
Mat.-Nr. 221614
Berlin, 31. März 2011

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

Diplomarbeitsbetreuer: Dipl.-Inform. Andreas Mertgen
Erstgutachter: Prof. Dr. Ing. Stefan Jähnichen
Zweitgutachter: Dr. Ing. Steffen Helke

Eidesstattliche Erklärung

Die selbstständige und eigenhändige Ausfertigung versichert an Eides statt.
Berlin, den 31. März 2011

. . . . . .
Unterschrift

Zusammenfassung

Generic Object Teams (GOT) ist eine Spracherweiterung der aspektorientierten Programmiersprache Object Teams. Sie erweitert die Sprache um das Konzept von Metavariablen, welche es ermöglichen, generische Aspekte zu erzeugen. Diese generischen Aspekte ergänzen die Möglichkeiten von Object Teams *Crosscutting Concerns* zu modularisieren. Metavariablen sollen an Stelle von Sprachelementen wie Referenzen zu Methoden, Klassen und Feldern in den Quellcode von Object Teams/Java (OT) eingebettet werden können. Diese Metavariablen werden dann, in einem weiteren Schritt mit Hilfe einer Faktenbasis über das Basisprogramm mit Elementen des Basisprogramms belegt. Abschließend wird aus dem aufgelösten Quellcode OT Code generiert.

In dieser Arbeit wird für die Sprache Generic Object Teams eine entsprechende Syntax und Werkzeugunterstützung entwickelt. Dafür wird das Object Teams Development Tooling (OTDT) als Basis genommen und für Generic Object Teams ergänzt. Die Erweiterungen umfassen die Anpassung von Compiler und Userinterface. Außerdem soll ein Typsystem für Metavariablen entwickelt werden, um eventuelle Fehler bereits bei der Entwicklung von Generic Object Teams Code zu erkennen. Spezielle in Verbindung mit dem neuen Konzept von Metavariablen stehende Fehler sollen dem Nutzer angezeigt und wenn möglich Lösungsvorschläge unterbreitet werden.

Die gesamte Adaption des OTDT soll mit Hilfe von Aspekten erfolgen, um die Evolutionsfähigkeit und Wiederverwendung zu steigern. Alle Veränderungen werden in Module gebündelt und zu einer eigenen, separaten Werkzeugunterstützung zusammengefasst. Diese Werkzeugunterstützung soll bei Bedarf als Eclipse Plugin in eine Eclipse Installation integriert werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

1.1 State of the Art

Die aspektorientierte Programmierung wurde geschaffen um die Grenzen der Objektorientierung auszudehnen. Sie soll Wartbarkeit, Wiederverwendbarkeit und Evolution von komplexer Software durch höhere Modularität verbessern.

Komplexe Software muss eine Vielzahl an Aufgaben und Problemen bewältigen. Darunter fallen unter anderem Anforderungen, Features, Varianten, Datenstrukturen und Patterns. Um die Komplexität dieser Aufgaben in den Griff zu bekommen, können Aufgaben und Probleme, genannt Concerns, in einzelne Module aufgeteilt werden. Diese Modularisierung der einzelnen Concerns nennt man *Seperation of Concerns*.

Die Objektorientierung bietet für die Modularisierung bereits diverse Konzepte wie Objekte, Packages und Vererbung. Es existieren jedoch Anforderungen bei denen eine Modularisierung mit den Mitteln der Objektorientierung nicht möglich ist. Ein Großteil dieser Anforderungen sind *Crosscutting Concerns*. Crosscutting Concerns erfordern entgegen des verwendeten Programmiermodells querschneidende Veränderungen. So kann ein Concern die Veränderung an einer Vielzahl von bereits gebildeten Modulen erfordern. Die bekanntesten Vertreter für derartige Concerns sind Logging, Seralisierung und Rechtevergabe. In all diesen Concerns müssen eine Vielzahl von Programmelementen verändert werden. Um diese Concerns zu modularisieren, kann auf aspektorientierte Programmiersprachen wie OT zurückgegriffen werden.

1.1.1 Object Teams/Java

Object Teams vereint Konzepte der Objektorientierung mit dem Konzept von Teams und Rollen.

Die Objektorientierung abstrahiert die Struktur und das Verhalten von realen Objekten eines Systems. Dabei werden die wichtigsten strukturellen Eigenschaften realer Objekte des Systems in Klassen zusammengefasst. Klassen dienen dann als Vorlage zur Erzeugung von Objekten. Ähnlich den realen Objekten des Systems kollaborieren auch die abstrahierten Objekte in vielseitigen Beziehungen miteinander. Je nach Aufgabe des Systems erfolgt dies auf unterschiedliche Weise. Das Verhalten der Objekte passt sich der aktuellen Aufgabe an. Die Objektorientierung hat Möglichkeiten, die Struktur der Objekte mit Hilfe von Vererbung zu modularisieren.

Object Teams erlaubt es weitergehend, das Verhalten von Objekten mit Kontexten in Verbindung zu bringen. Dafür wird ein neues Konzept Team (Kontext) als Sprachkonstrukt ähnlich einer Klasse eingeführt, welches eine Menge von inneren Klassen (Rollen) beinhalten kann. Jede Rolle erweitert eine Klasse des zugrunde liegenden Basisprogramms, ähnlich der aus der Objektorientierung bekan-

nten Vererbung. Die Beziehung zwischen Rolle und Basis ermöglicht, ähnlich der herkömmlichen Vererbung, eine Adaptierung der Methoden und Felder der Basis durch die Rolle. Mehrere Rollen können in Teams gebündelt und als Aspekt bei Bedarf aktiviert oder deaktiviert werden. Je nach Kontext kann das Team aktiviert und damit das Verhalten aller darin adaptierten Objekte verändert werden. Eine tiefere Einführung in Object Teams bietet Herrmann (2003) und die offizielle Webseite unter dem Eclipse Projekt, Object Teams (2011a).

OT ermöglicht mit Hilfe des Konzeptes von Teams und Rollen die Modularisierung von *Crosscutting Concerns*. Dabei werden Crosscutting Concerns mit Hilfe von Teams modularisiert.

1.1.2 LogicAJ

LogicAJ verwendet einen unterschiedlichen Ansatz zur Modularisierung von *Crosscutting Concerns*. LogicAJ erstellt eine logische Faktenbasis vom Basisprogramm, die detaillierte Strukturinformationen über das Basisprogramm enthält. Auf Basis dieser Strukturinformationen können logische Anfragen formuliert werden. Diese Anfragen wählen Programmelemente der Basis zur Adaption durch Aspektcode aus. LogicAJ ermöglicht es, die definierten Anfragen-Statements beliebig zu erweitern und wiederzuverwenden. Die Formulierung der Anfragen erfolgt in einer an die logische Programmiersprache Prolog angelehnten Syntax. Die Auswertung und damit die Auswahl von Elementen der Basis erfolgt mit Hilfe einer Prolog Inferenzmaschine. Aufgrund der umfangreichen Faktenbasis und der generischen Formulierung von Joinpoints, den Punkten an denen Aspektcode und Basiscode zusammentreffen, ist eine robuste Modularisierung von *Crosscutting Concerns* möglich. Nähere Informationen zu LogicAj bietet die offizielle Webseite LogicAj (2011).

1.2 Problembeschreibung

Um *Crosscutting Concerns* zu modularisieren müssen oft eine Vielzahl von Joinpoints definiert werden. In OT besteht die Definition eines Joinpoints mindestens aus einem Team, einer Rolle mit einer Basisbindung und dem eigentlichen Joinpoint, einer Callinbindung zwischen Basis- und Rollenmethode. Jede Definition eines Joinpoints erfolgt explizit, statisch durch den Entwickler. Mit dieser Art von Definition lässt sich eine Vielzahl von Problemen elegant lösen. Es gibt allerdings Fälle, in denen man sich einen generischen Ansatz wünschen würde. Als Beispiel sei eine Menge von Klassen `{A,B,...}` gegeben, die alle die gleiche Methode `m` enthalten, aber nicht durch eine Vererbungshierarchie vereinbar sind. Wollte man die Methode `m` in allen diesen Klassen adaptieren, wäre es notwendig für jede Klasse nahezu identische Rollen zu erstellen, die sich ausschließlich in ihrer Bindung zur Basisklasse unterscheiden. Eine generische Definition von Rollen und Bindungen, die derartige Probleme lösen könnten, wäre wünschenswert. Eine generische Rolle könnte dann eine Menge von Basisklassen adaptieren.

Neben der Wiederverwendung wird die Evolutionsfähigkeit von Teams und Rollen durch die Definition von statischen Bindungen eingeschränkt. Rollen, Methoden- und Feldbindungen werden ausschließlich über die Angabe des entsprechenden Featurenamens gebunden. Beispielsweise würde eine einfache Umbenennung einer Basisklasse, die Anpassung aller an die Basisklasse

gebundenen Rollen erfordern. Hier könnte eine generische Bindung, die zusätzliche Elemente des Basisprogramms einbezieht, robustere Aspekte hervorbringen.

Für die Modularisierung von *Crosscutting Concerns* wäre eine generische Definition von Rollen, Methoden, Feldern und Bindungen, die repräsentativ für eine Menge von Elementen verwendet werden können, von Vorteil. Diese generische Definition würde eine bessere Quantifizierung und Wiederverwendung von Rollen erlauben. Außerdem kann die Evolutionsfähigkeit der Aspekte gesteigert werden.

1.3 Zielsetzung

Um die im vorherigen Abschnitt beschriebenen Einschränkungen von OT aufzulösen, soll eine generische Version von OT entwickelt werden. Diese Version soll die Definition von generischen Feldern, Methoden, Rollen und Bindungen ermöglichen. Dafür wird eine Querysprache benötigt, die dem Entwickler die notwendigen Informationen für die Definition von generischen Sprachelementen liefert. Die Querysprache dient als Anfragesprache für eine logische Faktenbasis, die Strukturinformationen über das zu adaptierende Programm enthält. Der Entwickler hat damit die Möglichkeit generische Sprachelemente zu definieren, die auf alle verfügbaren strukturellen Informationen über das Basisprogramm zugreifen können.

Für die Integration der Querysprache muss OT um zusätzliche Sprachelemente erweitert werden. Es müssen Elemente für die Formulierung von Anfragen und Elemente für die Verwendung der Ergebnisse der Anfragen entworfen werden. Um die Zahl der neuen Sprachelemente möglichst gering zu halten, erfolgt ein großer Teil der Anfragen und Ergebnisse mit Hilfe von Metavariablen. Metavariablen erfüllen zwei Rollen. Sie dienen als Terme zur Formulierung einer logischen Anfrage an eine Inferenzmaschine und als Träger für die Ergebnisse dieser Anfrage. Metavariablen werden wie in logischen Programmiersprachen üblich, gematcht und belegt. Dieses System wird in der logischen Programmierung Unifikation genannt. Die logische Querysprache, aufbauend auf Metavariablen, ermöglicht eine quantifizierte Auswahl von Joinpoints, mit einer geringen Anzahl an zusätzlichen Sprachelementen.

Die beschriebene Erweiterung von Object Teams kann in zwei Teilbereiche untergliedert werden. Zum einen der theoretische Entwurf einer den Anforderungen genügenden Syntax und zum anderen die Integration der neuen Sprachfeatures in das OTDT.

1.3.1 Entwurf einer Syntax für Generic Object Teams

An die Verschmelzung von OT mit Metavariablen und der Entwicklung einer logischen Querysprache werden folgende Anforderungen gestellt:

- Es soll untersucht werden welche Sprachelemente sich für die Verwendung von Metavariablen eignen. Metavariablen können an einer Vielzahl von Sprachelementen eingesetzt werden. Dazu zählen z. B. Rollendefinitionen, Methodendefinitionen, Felddefinitionen und Bindungen. Allerdings sind Metavariablen nicht für alle Sprachelemente sinnvoll. Elemente für die kein

sinnvoller Anwendungsfall gefunden werden kann, sollten nicht in die Sprache aufgenommen werden, um zusätzliche Komplexität zu vermeiden.

- Die Formulierung von Queries soll neben der Verwendung von Metavariablen auch darüber hinaus gehende Anfragen an die Faktenbasis ermöglichen, um deren Wissen voll auszuschöpfen. Queries werden vorwiegend mit Hilfe von Metavariablen gebildet. In vielen Fällen ist es allerdings notwendig Informationen, die nicht als Metavariablen repräsentiert werden können, in einen Query mit aufzunehmen. Solche Informationen über das Basisprogramm könnten z. B. Packages, Konstruktoren und Strings sein, für die entsprechende Datentypen gefunden werden müssen (z. B. String, int).
- Die Syntax muss sich in das Sprachkonzept von OT eingliedern. Die Syntax der Querysprache soll eher an die Syntax von OT, als die von Prolog angelehnt sein. Die Anzahl der zusätzlichen Sprachelemente soll gering gehalten werden.
- Queries können leicht an Komplexität gewinnen. Daher muss ein Sprachmittel gefunden werden, welches dem Entwickler die Erstellung von eigenen Queries ermöglicht. Dabei soll eine größtmögliche Wiederverwendung ohne eventuelle Textkopien gewährleistet werden. Zu erwägen wäre hierbei eine Trennung von Definition und Instanzierung von Queries. Des Weiteren soll untersucht werden, in welcher Form die Definition von Queries vorgenommen werden kann. Die Definition könnte innerhalb von OT `.java` Dateien, in eigenen Dateien oder inline erfolgen.
- Es soll eine statische Typüberprüfung der definierten Metavariablen ermöglicht werden, um den Benutzer vorzeitig über Typfehler zu informieren. Dafür muss untersucht werden wie sich die untypisierten Metavariablen mit dem streng typisierten OT verbinden lassen und wie der Nutzer frühzeitig vor Fehlern gewarnt werden kann.
- Bei der Verwendung von herkömmlichen und generischen OT Mitteln kann es zu Fällen kommen in denen die Struktur des Programms ein logisches Constraint impliziert. Als Beispiel könnte eine nicht generische Rolle, generische Methoden und Bindungen enthalten. Das implizierte Constraint wäre dann die Verknüpfung des Selektion-Statements mit der Basisklasse. Diese impliziten Constraints sollen dem Nutzer zur Verfügung gestellt werden. Denkbar wären semi-automatische Hilfestellungen wie Quickfixes.

1.3.2 Integration der Querysprache in Object Teams und das OTDT

Die auf dem Papier entwickelte Querysprache, samt aller zusätzlichen Sprachelemente soll in das OTDT integriert werden. Die Integration umfasst die Einbettung der Spracherweiterung in das OTDT, sowohl in das User Interface (UI) als auch den Compiler. Es soll ein Tooling für die Sprache Generic Object Teams mit folgenden Anforderungen entwickelt werden.

- Der OT Quellcode Editor und der dahinter liegende Abstrakte Syntax Baum muss um die zusätzliche Syntax erweitert werden.
- Beim Verändern von Quellcode im Generic Object Teams Tooling (GOTDT) Editor wird eine Typüberprüfung der Metavariablen gestartet und eventuell vorhandene Typfehler dargestellt.

- Für die Auflösung von Typfehlern soll exemplarisch ein Quickfix implementiert werden.
- Neben Typfehlern soll auch für die impliziten Constraints ein Quickfix implementiert werden.
- Am Ende des Compilerprozesses soll eine extern entwickelte Schnittstelle aufgerufen werden, die eine Auflösung und Belegung der Metavariablen durchführt und Generic Object Teams Code in OT Code umwandelt.

Optional können neben dem Quellcode Editor noch Elemente wie Outline und Binding Editor an die neuen Sprachfeatures angepasst werden. Neben Quickfix können auch andere semi-automatische Hilfesysteme evaluiert werden.

1.4 Einordnung und ähnliche Arbeiten

Das Konzept von generischen Aspekten ist nicht neu. Es gibt einige aspektorientierte Sprachen, für die Erweiterungen entwickelt wurden, die eine generische, statische Definition von Aspekten mit Hilfe von Querysprachen ermöglichen. Diese sollen im Folgenden kurz umrissen werden und in Bezug zu Generic Object Teams gestellt werden.

Für die .NET Plattform wurde die aspektorientierte Sprache Compose*, um eine neue Querysprache erweitert. Diese Querysprache ähnelt in ihrer Syntax der von Generic Object Teams. Sie basiert ebenfalls auf der logischen Unifikation. Metavariablen sind jedoch, im Gegensatz zu GOT, auf die Definition des Queries beschränkt. Eine Typisierung und eine Integration dieser in das Tooling findet nicht statt (siehe Havinga (2005)).

Der engste Verwandte von Generic Object Teams ist die auf AspectJ aufbauende Sprache LogicAJ. Sie verwendet ähnlich zu Generic Object Teams Metavariablen, um über eine Prolog ähnliche Querysprache generische Aspekte zu erstellen. Auch hier existiert kein Konzept für eine Typisierung von Metavariablen. Im Zuge der Entwicklung von LogicAJ 2 liegt der Fokus auf der Ausdrucksstärke der Querysprache, um feingranulare Möglichkeiten zu bieten Joinpoints auszuwählen und entsprechende Advices definieren zu können (siehe Rho u. a. (2006)).

Vorausgehend zu der Entwicklung von Generic Object Teams wurde bereits eine Querysprache für Object Teams entwickelt. Dabei war es ein Hauptziel eine erweiterte Quantifizierung von Joinpoints für Object Teams zu ermöglichen, um Crosscutting Concerns besser abbilden zu können. Die Auswertung von Queries und Generierung von OT Code wurde mit Hilfe des Metamodells des Basisprogramms und Modell-Transformationen durchgeführt. Die für die Abfragen notwendigen Queries wurden in einer eigens definierten Sprache erstellt. Die Verbindung zu Object Teams erfolgt ausschließlich über spezielle Callin- und Calloutbindungen. Die generischen Anteile eines solchen OT Quellcodes sind also im Gegensatz zu GOT auf die Bindungen beschränkt (siehe Mertgen (2007)).

Eine Übersicht über weitere generische aspektorientierte Sprachen und deren Gegenüberstellung bietet Kniesel und Rho (2006).

Generic Object Teams vereint einige der Konzepte der in diesem Abschnitt vorgestellten Sprachen. So wird eine an die Querysprachen von Compose* und LogicAJ angelehnte Syntax für die Definition

von Queries verwendet. Als Basis für die Auswertung von Queries dient die gleiche logische Faktenbasis, die auch von LogicAJ verwendet wird. Die Transformationen von GOT nach OT Code benutzt die selben Mechanismen wie LogicAJ (Conditional Transformation). Metavariablen können ähnlich zu LogicAJ im gesamten Aspekt, an unterschiedlichsten Stellen verwendet werden. Die hervorstechende Neuerung zu den vorgestellten Sprachen ist die enge Orientierung an den Konzepten von Object Teams. So werden auch Metavariablen mit einem Typsystem versehen und in das bestehende Tooling von Object Teams integriert. Implizite Constraints werden dem Nutzer als Lösungsvorschläge vorgeschlagen.

1.5 Aufbau der Arbeit

Im ersten Teil der Arbeit (Kapitel 2) wird ein Überblick über die Sprache GOT gegeben, welche die generischen Erweiterungen wie Metavariablen und Queries von OT beschreibt. Dabei werden die Konstrukte der Sprache erläutert und eine mögliche Syntax vorgestellt. Darauf folgend wird eine kleine Einleitung in das OTDT gegeben. Die für die Entwicklung des auf dem OTDT basierenden GOTDT relevanten Strukturen werden herausgestellt. Da das GOTDT eine Werkzeugunterstützung für eine Programmiersprache darstellt, sind Datenstrukturen, wie der AST eine wichtige Grundlage für das gesamte Tooling. Diese Datenstrukturen, die durch das gesamte GOTDT verwendet werden, werden in Kapitel 4 vorgestellt.

Den Hauptteil der Arbeit macht die Beschreibung des GOT Compilers aus. Die Anpassungen des OT Compilers für GOT werden in Kapitel 5 erläutert. Dafür müssen Grammatik, Scanner, Parser und das Typsystem angepasst werden.

Im letzten Teil der Arbeit (Kapitel 6) wird die Entwicklung des Userinterface samt der Integration des GOTDT in Eclipse erläutert. Den Abschluss macht ein Fazit und der Ausblick für mögliche aufbauende Arbeiten in Kapitel 7 .

2 Die Sprache Generic Object Teams

Ziel dieser Arbeit ist die Erweiterung der Sprache Object Teams um weitere Generizität. Es soll mit Hilfe einer Querysprache und dem Einsatz von Metavariablen eine Möglichkeit geschaffen werden, Joinpoints für OT zu generieren. Die für dieses Ziel erforderlichen Erweiterungen bündeln sich unter der neuen, auf OT aufbauenden Programmiersprache Generic Object Teams.

Generic Object Teams ist eine auf Object Teams basierende Sprache. Sie besitzt zusätzlich zu den Sprachelementen von OT, neue, spezifische Elemente. Der OT Compiler und die OT Werkzeugunterstützung sollen um die neuen Sprachfeatures erweitert werden, um eine GOT Werkzeugunterstützung zu schaffen. Für dieses Vorhaben müssen die neuen Features von Generic Object Teams in Hinblick auf ihre Umsetzung in Compiler und Tooling identifiziert werden. Dieses Vorhaben wird im folgenden Abschnitt beschrieben. Im Rahmen dieser Arbeit steht die Entwicklung von Compiler und Tooling im Vordergrund. Das Sprachdesign wird vernachlässigt und als gegeben angesehen. Eine detaillierte Sprachbeschreibung ist aus der Generic Object Teams Sprachdefinition zu entnehmen (siehe Mertgen (2010)).

Um möglichst viel Funktionalität des OT Compilers wiederzuverwenden, ist es sinnvoll GOT Sprachelemente möglichst mit Hilfe von OT Sprachelementen abzubilden. Daher wird im Folgenden auch stets erläutert wie sich das beschriebene Sprachelement mit Hilfe von OT Sprachelementen darstellen lässt. In diesem Zusammenhang wird oft der GOT AST erwähnt. Der GOT AST ist ein zum regulären OT AST paralleler Abstrakter Syntax Baum, der spezielle GOT Knoten enthält. Dabei sind diese Knoten im eigentlichen Sinne OT Rollen auf bestehende Elemente im OT AST. Der GOT AST dient zum Speichern von Informationen, die im Repräsentativen Java Element nicht gespeichert werden können. Die Bedeutung und eine nähere Erläuterung des GOT AST findet im Kapitel 4 statt.

Grundsätzlich lassen sich alle neuen Sprachelemente in zwei Gruppen einteilen. Es gibt Elemente, die Teil der Querysprache sind und Elemente, die dazu dienen Metavariablen in OT Code zu integrieren. Als erstes wird der Aufbau der Querysprache und die dafür benötigten Elemente erläutert.

2.1 Querysprache

Die Kernfunktionalität von Generic Object Teams ist die Verwendung einer logischen Faktenbasis als Quelle für Strukturinformationen über das zu adaptierende Basisprogramm. Mit Hilfe von diesen Informationen können OT Elemente wie Rollen, Methoden, Callins und Callouts generiert werden. Um aus der teils riesigen Menge an Informationen in der Faktenbasis die Teile auszuwählen,

die für OT relevant sind, müssen Abfragen (Queries) formuliert werden, die die gewünschten Informationen auswählen. In Prolog werden Queries mit Hilfe von logischen Gleichungen mit Metavariablen definiert. Diese Ausdrücke werden dann per Unifikation so belegt, dass eine gültige Gleichung entsteht. Als Ergebnis entsteht eine Menge von Tupeln, die alle gültigen Belegungen der Query Gleichung wiedergeben. Da die Sprache Generic Object Teams den Anspruch hat, sich nah an der Konzeption von Object Teams zu orientieren, sollen Queries in einer Prolog-freien Form, nah an den Konzepten von Java und OT, definierbar sein. Außerdem soll bei der Umsetzung darauf geachtet werden, dass Modularisierung in Form von Bibliotheken von Queries möglich ist.

Um Veränderungen und damit Aufwand und Fehlerquellen am OTDT möglichst gering zu halten, wird probiert eine Syntax für die Querysprache zu finden, die möglichst nah an OT und Java angelehnt ist. Eine Wiederverwendung von OT und Java Sprachelementen bietet sich an.

2.1.1 Queryklassen

Um die Definition eines eigenen Dateityps für Queries zu umgehen und nah an bestehenden Java Mitteln zu bleiben, werden Queries mit Java Klassen abgebildet. Diese speziellen Queryklassen können analog zu herkömmlichen Klassen in eigenen Java Dateien, oder zusammen mit anderen Klassen in einer Java Datei verwendet werden. Exemplarisch sei in Listing 2.1 eine solche Queryklasse gezeigt.

Queryklassen werden mit dem vorangestellten Modifier `otquery` gekennzeichnet. Da Overriding von Querymethoden semantisch wenig Sinn macht, ist der `abstract` Modifier auf Queryklassen nicht zulässig. Auch der `team` Modifier ist für Queryklassen ungültig. Queryklassen dürfen ausschließlich Querymethoden enthalten. Sie beinhalten keine weiteren Features.

Queryklassen können im Compiler als eingeschränkte Javaklassen abgebildet werden. Der vorangestellte `otquery` Modifier dient zur Unterscheidung von herkömmlichen Klassen und muss separat gespeichert werden. Es bietet sich dafür ein der Klasse zugeordneter Knoten im GOT AST an.

```
1 package myquerypackage;
2
3 public otquery class MyQuery {
4   public otquery classOfName(?Type ?t, String s) {
5     isClass(?t) && equalsName(?t, s)
6   }
7 }
```

Listing 2.1: Das Listing zeigt eine Queryklasse, die in einem eigenen Package `myquerypackage` innerhalb einer Java Datei definiert ist. Die Klasse beinhaltet einen Query `classOfName`, der die Metavariable `?t` mit allen Typen in Form von Klassen mit dem Namen `s` belegt. Dass es sich um eine Queryklasse handelt, ist an dem vorangestellten Modifier `otquery` zu erkennen.

2.1.2 Querymethoden

Querymethoden kapseln den eigentlichen Query und ermöglichen eine von Funktionen bekannte Funktionalität-Bündelung. Innerhalb von Querymethoden kann ein komplexer Query definiert werden, der sich nach außen für den Nutzer als Funktion mit Parametern repräsentiert. Querymethoden werden ähnlich wie Klassen durch den `otquery` Modifier gekennzeichnet. Da Overriding bei Querymethoden wenig Sinn macht, sind alle Querymethoden implizit *statisch*.

Die Parameterliste einer Querymethode entspricht der Syntax einer regulären Javamethode mit der Einschränkung, dass ausschließlich Metatypen, `int` und `String` als Parametertypen verwendet werden dürfen. Zusätzlich müssen alle Parameter mit Metatypen Metavariablen sein.

Querymethoden haben im Gegensatz zu Java Methoden keinen Rückgabetyp. Sie dienen nur als Informationsquelle für den *JTransformer* und sind im übersetzten Quellcode nicht mehr vorhanden.

Der Body von Querymethoden darf lokale Deklaration aus dem gleichen Bereich wie die Parameterliste aufweisen. Dabei wird jede Deklaration mit einem Semikolon beendet. Zentrales Element des Bodies ist das Querystatement, welches wiederum eine Queryexpression beinhaltet. Die Queryexpression repräsentiert den eigentlichen Query. Das Statement wird nicht mit einem Semikolon beendet. Listing 2.1 zeigt eine Querymethode innerhalb einer Queryklasse.

Zur Abbildung im Compiler können für Querymethoden gewöhnliche Javamethoden verwendet werden. Die Javamethode bekommt den `static`-Modifier und als Rückgabewert `boolean`. Da Querymethoden wiederum in Queries verwendet werden sollen, ist ein logischer Rückgabewert für die Typisierung von Vorteil. Er ermöglicht die problemfreie Verwendung innerhalb von logischen Javaexpressions. Der `otquery` Modifier identifiziert eine Methode als Querymethode und wird im GOT AST gespeichert.

2.1.3 Queries

Der eigentliche Query wird als logischer Ausdruck (Expression) dargestellt. Diese Queryexpression ähnelt einer aus Java bekannten logischen Expression. Sie ist jedoch auf die von Java bekannten Operatoren `!`, `&&` und `||` beschränkt. Neben den Atomen `true` und `false` können Queries Aufrufe von Querymethoden beinhalten. Jede Queryexpression wird von einem Querystatement umgeben, welches sich in eine Querymethode einbetten lässt. Da nur eine Queryexpression pro Querymethode zugelassen ist, wird das Expression Statement *nicht* mit einem Semikolon beendet. Listing 2.1 zeigt einen in eine Queryklasse und Methode eingebetteten Query.

Eine Queryexpression entspricht einer eingeschränkten Javaexpression und kann auch als solche abgebildet werden. Es müssen keine weiteren Anpassungen vorgenommen werden, da die Queryexpression eine Teilmenge einer Javaexpression ist. Die Information, dass es sich bei der besagten Javaexpression um eine Queryexpression handelt, wird im GOT AST gespeichert.

2.2 Metavariablen für Object Teams

Generic Object Teams basiert auf Object Teams. Die zusätzlich einzuführenden Sprachelemente sollen im folgenden erläutert werden. Kern der neu eingeführten Elemente bilden die sogenannten Metavariablen, welche in Abschnitt 2.2.1 beschrieben werden. In den darauf folgenden Abschnitten werden weitere Sprachelemente vorgestellt, die die Typisierung und Belegung von Metavariablen sowie eine Transformation in OT Code ermöglichen.

2.2.1 Metavariablen

Ein Object Teams Programm und die darin enthaltenen Teams, Rollen und reguläre Javaklassen bestehen aus einer Vielzahl kleinerer Sprachelemente. Die Idee von Generic Object Teams ist es, diese Sprachelemente wahlweise mit Metavariablen zu ersetzen. So können Typnamen, Methodennamen und Feldnamen durch Metavariablen ersetzt werden. Dabei dient eine Metavariable als Platzhalter für eine beliebige Menge an Elementen. Diese Menge an Elementen wird durch die Auswertung, der im Abschnitt 2.1 beschriebenen Queries bestimmt. Die Menge kann Elemente von unterschiedlichem Typ enthalten. So könnte eine Metavariable mit zwei Methoden und drei Klassen belegt werden. Sofern die besagte Metavariable als Ersatz für einen Methodennamen verwendet wird, würde es bei einer Umsetzung in OT augenscheinlich zu Fehlern kommen. Es liegt daher nahe sich an der strikten Typisierung von Object Teams zu orientieren und ein Typsystem für Metavariablen einzuführen, um Fehler, die erst bei der Generierung auftreten würden, schon beim Entwickeln zu erkennen.

Dafür werden Metavariablen mit einem neuen Typ versehen. Ein sogenannter Metatyp. Dieser Typ liegt nicht auf der selben Metaebene wie ein Interface oder eine Klasse, sondern vielmehr auf der nächst höheren Metaebene. Ein Metatyp beschreibt OT Sprachelemente. Er bestimmt für eine Metavariable welche Elemente in der durch sie dargestellten Menge enthalten sein dürfen. So kann beispielsweise die Menge einer aufgelösten Metavariable je nach Metatyp nur noch Methoden *oder* Klassen enthalten. Alle möglichen Metatypen werden im folgenden aufgelistet.

- **?Class** konform zu `java.lang.Class`
- **?Type** konform zu `java.lang.reflect.Type`
- **?Method** konform zu `java.lang.reflect.Method`
- **?Field** konform zu `java.lang.reflect.Field`
- **?Modifier** konform zu `java.lang.reflect.Modifier`
- **?String** konform zu `java.lang.String`
- **?int** konform zu int / `java.lang.Integer`
- **?boolean** konform zu boolean / `java.lang.Boolean`

Metatypen und Metavariablen werden als solche durch ein vorangestelltes `?` gekennzeichnet. Für Metavariablen existieren außerdem die Präfixe + und -. Dabei handelt es sich um gebundene und

```
1 ...
2 protected class -Sysout playedBy ?baseInPackageData {
3   void writeToDatabase(Object data) <- after void ?setter(?dataType data);
4
5   void writeToDatabase(Object data) {
6     db.commit(data);
7   }
8 }
9 ...
```

Listing 2.2: Der Codeausschnitt beschreibt einen Aspekt, der für jede setter Methode in allen Klassen in einem bestimmten package, die übergebenen Objekte in einer Datenbank speichert. Erreicht wird dies mit Hilfe der Rolle `-Sysout` die für jede Klasse in der durch die Metavariable `?baseInPackageData` erzeugten Menge, generiert wird. Ähnlich wird für jede Methode aus der `?setter` Menge eine Bindung zur Methode `writeToDatabase` erzeugt.

freie Metavariablen. Freie Metavariablen sind für die Generierung notwendig. Sie dienen als frei vom Transformer belegbarer Name. Soll beispielsweise eine Rolle `-Sysout` für jede Basisklasse `?baseInPackageGOT` erzeugt werden, so muss für jede Rolle ein eigener Name gefunden werden. Da dieser Name bei der Erzeugung frei belegt werden muss, muss der Rollenname durch eine freie Metavariable gekennzeichnet werden. Ein Beispiel für den Einsatz von Metavariablen wird in Listing 2.2 erläutert.

Für alle Metavariablen gilt eine Konformität zu existierenden Javatypen des Java Reflection API. Dadurch kann auf die Funktionalitäten des Application Interface (API) zugegriffen werden und es ist eine einfache Abbildung von Metatypen im Compiler möglich. Da Metavariablen und Typen als Ersatz für eine Vielzahl von Elementen verwendet werden können, werden sie auch im Compiler durch verschiedene Elemente abgebildet. Dabei wird ihre Identität als Metavariable oder Typ im GOT AST gespeichert.

Verwendung von Metavariablen

Die verschiedenen Metatypen wurden im vorherigen Abschnitt eingeführt. Im Folgenden wird der Einsatz von Metavariablen näher erläutert.

Eine Metavariable, die zum Einsatz kommt, muss zuvor mit einem Metatyp deklariert werden. Hinter diesem Metatyp verbirgt sich ein, wie in Abschnitt 5.4.1 beschrieben, normaler Javatyp. Die Metavariable kann also analog zu einer regulären Javavariable eingesetzt werden. Die Metavariable `?Method ?m` entspricht genau ihrem Java Pendant `java.lang.reflect.Method m`. Daraus folgt, dass alle Methoden der Klasse `java.lang.reflect.Method` auch über die Metavariable erreichbar sind.

Bis zu diesem Punkt leistet die Metavariable nicht mehr als ihr Java Pendant. Sie entspricht einem Alias. Ohne eine zusätzliche Bedeutung wäre es also kaum sinnvoll sie einzuführen.

Daher soll es zusätzlich erlaubt sein, Metavariablen überall dort einzusetzen, wo eine ihrem Metatyp entsprechende Referenz im Code verwendet wird. Eine solche Referenz wäre exemplarisch für den

Metatyp `?Class` der Name einer Klasse. Für den Metatyp `?Method` die komplette Signatur oder der Aufruf einer Methode. In den meisten Fällen ist eine Referenz der (qualifizierte) Name des durch die Metavariable repräsentierten Sprachelementes.

Bei den Metatypen `?String`, `?boolean`, `?int` können weitergehend die eventuell durch die Auswertung des Queries erzeugten Werte, verwendet werden. Tabelle 2.1 zeigt eine Auswahl an möglichen Werten von Metavariablen und deren potentielle Verwendung.

Metavariable	Möglicher Wert nach Auswertung durch einen Query	Potentielle Verwendung im OT Quellcode
?Class ?c	`somepackage.SomeClass` `SomeClass`	`playedBy ?c` `instanceof ?c`
?Method ?m	`somepackage.SomeClass.someMethod` `void someMethod(String s)`	`myMethod <- replace ?m` `void myMethod() <- replace ?m`

?Type ?t	`somepackage.SomeClass`
	`somepackage.SomeInterface`
	`java.lang.Boolean`
		`playedBy ?t`
		`instanceof ?t`
		`void do(?t finished)`

?Field ?f	`somepackage.SomeClass.someField` `someField`	`?f.getClass()` `long getID() -> get ?f`

?Modifier ?mo	`public`
	`static`
	`final`
		`?mo class -GenRole {}`
		`public ?mo String CONST`
		`public ?mo void someMethod()`

?String ?s	`"Some String"`	`printPackageName(?s)`
?int ?i	`1234`	`public static final METHODCOUNT = ?i`
?boolean	`true` `false`	

Tabelle 2.1: Die Tabelle zeigt alle Metatypen von Generic Object Teams und deren mögliche Belegung nach dem Ausführen eines Queries. Die dritte Spalte beschreibt mögliche Stellen im OT Code an denen Metavariablen des entsprechenden Typs verwendet werden könnten.

Metavariablen können also entweder, entsprechend ihres zum Metatyp zugewiesenen Java Pendant, oder als Referenz oder Wert verwendet werden. Durch diese Mehrdeutigkeit entstehen neue Probleme.

So kann ein Aufruf an einer Feld Metavariable `?f.`, sowohl ein Feature des Javatyps als auch des durch die Metavariable bestimmten, aufgelösten Typs adressieren. Um diese Doppeldeutigkeit aufzulösen gibt es verschiedene Ansätze. So könnte man eine Priorisierung einführen, die eventuell durch spezielle Sprachelemente gesteuert werden kann. Außer wäre die Einführung eines neuen Operators z. B. „->“ denkbar.

Eine weiteres Problem entsteht beim direkten Aufruf einer Methode mit Hilfe einer Methoden Metavariable „`x.?m.n`“. Hier kann die Metavariable für ein Objekt des Typs `java.lang.reflect.Method`, oder eine Referenz auf die durch `?m` repräsentierte Menge von Methoden stehen. Es stellt sich somit die Frage ob `?m` einen Methodenaufruf darstellt, oder einen Zugriff auf das `java.lang.reflect.Method` Objekt ist. Da dieser Aufruf ausschließlich im Fall einer leeren Parameterliste der Methode zu einem gültigen Generat führen kann, könnte man in diesem Fall, bei einem Methodenaufruf durch eine Metavariable, die nachfolgenden Klammern `()` forcieren. „`x.?m().n`“ würde `?m` also als aufgelöste Referenz auf eine Methode betrachten und „`x.?m.n`“ als Objekt vom Typ `java.lang.reflect.Method`.

Der Fokus von Generic Object Teams liegt auf Rollen und damit Callins und Callouts. Die eben beschriebenen, sehr speziellen Fälle, werden daher nicht gesondert behandelt. Es wird das durch das Tooling gegebene Grundverhalten beibehalten. Dieses Verhalten wird in der folgenden Aufzählung erläutert.

- Aufrufe über den Punkt Operator auf einer Metavariable werden ausschließlich als Aufrufe auf dem durch die Metavariable repräsentierten Javaobjekt interpretiert.
- Ebenso wird bei Aufrufen der Form `x.?m` die Metavariable `?m` als Objekt des durch den Metatyp definieren Javaobjekts gesehen.

2.2.2 Die Sprachelemente `match` und `declare`

Metavariablen müssen deklariert und einem Query zugeordnet werden. Diese Aufgabe übernehmen die im folgenden vorgestellten Elemente. Die neuen Sprachelemente werden an die Definition eines Teams angehängt. Das `match` und `declare` Element dienen als Verbindung zwischen OT Code und Queries. Beide Elemente sind in ihrer Syntax identisch. Sie unterscheiden sich ausschließlich in ihren Bezeichnern und selbstverständlich in ihrer Semantik. Daher wird nur auf das `match` Element eingegangen. Ein `match` Element wird optional in den Header eines Teams als letztes Element nach dem optionalen `playedBy` und `Guard` eingesetzt. Die Syntax des `match` Sprachelementes ähnelt der in Abschnitt 2.1.2 vorgestellten Querymethode. Hauptunterschiede sind der feste, vorbelegte Name und die fehlenden Modifier.

Die Parameterliste und die darin verwendbaren Elemente sind identisch zu denen einer Querymethode. Auch der Body eines `match` Konstrukts hat die selben Inhalte, wie der einer Querymethode. Es kann ein Querystatement und mehrere lokale Deklarationen enthalten. Listing 2.3 zeigt ein Team mit dem `match` Sprachelement.

Da das neue Element zur Deklaration von Metavariablen für das Team dient, müssen alle deklarierten Metavariablen aus der Parameterliste des `match` Elements, ähnlich wie Felder, im ganzen Body der Klasse verfügbar sein.

```
1 public team class SomeTeam
2   match(?Class ?c, ?Method ?m, String s) {
3     ?Field ?f;
4     isMemberOfClass(?m, ?c) && isMemberOfClass(?f, ?c) && hasName(?f, s)
5   }
6   {
7 }
```

Listing 2.3: Das Listing zeigt eine Teamdeklaration mit einem `match` Element. Über die Parameterliste des `match` Elementes werden Metavariablen deklariert, die später im Body des Teams verwendet werden können. Innerhalb des Bodies des `match` Elementes werden die in der Parameterliste deklarierten Metavariablen, mit Hilfe eines Queries im Zuge der Codegenerierung belegt.

Die Abbildung im Compiler erfolgt ähnlich dem in Abschnitt 2.1.2 beschriebenen Vorgehen. Die Methode bekommt als Namen entweder `match` oder `declare`. Außerdem wird sie mit dem `static` Modifier versehen und erhält als Rückgabewert `boolean`. Die so entstandene Java Methode wird dann als reguläres Member einer Teamklasse geführt. Um auch innerhalb der Teamklasse Zugriff auf die über das `match` Statement deklarierten Metavariablen zu haben, werden die Parameter der `match` Methode in Felder umgewandelt und in das Team kopiert.

Die Information, ob es sich bei einem Team um ein spezielles GOT Team handelt, welches `match` oder `declare` Elemente enthält, wird im GOT AST gespeichert. Dort wird zur Vereinfachung des Zugriffs auch eine Referenz auf das zugehörige neue Element abgelegt.

2.2.3 Der `per` Block

Wie in Abschnitt 2.2.1 bereits erläutert, stehen Metavariablen als Platzhalter für eine Menge von Sprachelementen. Durch den Einsatz von Metatypen wird diese Menge auf bestimmte Sprachelement Typen beschränkt. Eine Metavariable `?c` vom Typ `?Class` wird also nur Klassen und keine Methoden enthalten. Die Anzahl der Elemente, die durch die Metavariable repräsentiert werden, ist beliebig. Dieser Umstand führt bei der Generierung von OT Code zu Mehrdeutigkeiten. Als Beispiel dafür sei Listing 2.4 gegeben. Aus Sicht der Generierung ist die dort gezeigte Rolle nicht eindeutig. Es gibt zwei verschiedene Interpretationen.

1. Es wäre möglich, dass für jede Klasse, die in der Menge der Metavariable `?setOfBases` enthalten ist, eine Rolle generiert wird. Darin wird für jede Basismethode, die in der Menge der Metavariable `?setOfMethods` enthalten ist, eine Callinbindung erzeugt.
2. Ebenso wäre es jedoch möglich, dass für jedes Callin eine eigene Rolle erzeugt wird.

Um diese Mehrdeutigkeit zu bändigen, ist die Einführung des `per` Blocks von Nöten. Mit Hilfe des `per` Elementes kann die Eindeutigkeit der Generierung gewährleistet werden. Dafür fungiert das `per` Element als umschließendes Element, welches den Bereich kennzeichnet, an denen pro gültiger Belegung der ihm übergebenen Metavariablen, ein neues Object Teams Element generiert wird.

Das `per` Sprachelement kann innerhalb von Klassen und Methoden verwendet werden. Der `per` Block hat nur Einfluss auf die Generierung. Er erstellt *keinen* neuen Namensraum. Ansonsten

```
1 ...
2 protected class -GenericRole playedBy ?setOfBases {
3   doSomething <- after ?setOfMethods;
4 }
5 ...
```

Listing 2.4: Das gezeigte Listing repräsentiert eine generische Rolle mit einer generischen Callinbindung. Die Rolle und Bindung werden in Abhängigkeit der Metavariablen `?setOfBases` und `?setOfMethods` erzeugt. Das hier gezeigte Fragment ermöglicht, ohne die fehlenden `per` Elemente, keine eindeutige Generierung von Rollen und Callinbindungen.

```
1  ...
2  per(?setOfBases) {
3    protected class -GenericRole playedBy ?setOfBases {
4      per(?setOfMethods) {
5        doSomething <- after ?setOfMethods;
6      }
7    }
8  }
9  ...
10 per(?setOfBases, ?setOfMethods) {
11   protected class -GenericRole playedBy ?setOfBases {
12     doSomething <- after ?setOfMethods;
13   }
14 }
15 ...
```

Listing 2.5: Die hier gezeigten Rollen zeigen zwei unterschiedliche Interpretationen der in Listing 2.4 gezeigten generischen Rolle. Die Eindeutigkeit wird durch die Verwendung von `per` Blöcken sichergestellt. Im ersten Fall wird für jede Basis der Menge `?setOfBases` eine Rolle `-GenericRole` erzeugt, die für jede Methode der Basis und der Menge `?setOfMethods` eine Callinbindung enthält. Im Zweiten Fall wird für jede Belegung von `?setOfBases` und `?setOfMethods` eine eigene Rolle erzeugt.

verhält er sich wie ein normaler Block, der beliebigen Kontext bündeln kann. So kann `per` innerhalb von Klassen um Felder, Methoden, Callinbindungen und Calloutbindungen gelegt werden. Und innerhalb von Methoden um Statements. Die Syntax des `per` Blocks ähnelt einer Methode mit festem Namen ohne Modifier. Er beinhaltet keine Parameterliste, sondern eine Argumentliste. So wird eine beliebige Anzahl von Komma separierten Metavariablen aufgelistet, die in der `match / declare` Deklaration eines umgebenden Teams deklariert sind. Für jede gültige Belegung der in der `per` Argumentliste aufgeführten Metavariablen, wird der im Body befindliche OT Code für erzeugt.

Das gezeigte Beispiel 2.4 wird in Listing 2.5 ohne Mehrdeutigkeiten für die Generierung gezeigt. Der obere Codeblock zeigt die erste Interpretation, der untere die Zweite.

Die Umsetzung des `per` Blocks mit Java Mitteln gestaltet sich etwas schwieriger als die der bisherigen Elemente. Da `per` innerhalb von Klassen *und* Methoden verwendet werden kann, entstehen einige Probleme. Das Hauptproblem ist es einen allgemeingültigen Java oder OT Repräsentant für einen `per` Block zu finden. Es gibt in Java oder OT kein Block-artiges Element welches sowohl an beliebigen Stellen innerhalb von Klassen, als auch von Methoden vorkommen kann. Daher bleibt nur die Möglichkeit das `per` Element, je nach Kontext, mit zwei verschiedenen Java Elementen darzustellen. Im Kontext einer Klasse wird `per` als Klasse und im Kontext einer Methode als Block dargestellt.

Problematisch bleibt dennoch die Unterbringung der Argumentliste. Keines der Elemente besitzt eine Argumentliste. Daher muss, um das `per` Element als Block abzubilden, der herkömmliche Block um eine Argumentliste erweitert werden. Bei bisherigen Realisierungen von Generic Object Teams Sprachelementen mit Hilfe von Java Elementen, war die Ausdrucksstärke der Java Elemente größer als die der GOT Elemente. Es konnte also ein Teil eines Java Elementes für ein GOT Element

verwendet werden. In diesem Fall ist es umgekehrt. Es muss zusätzlich eine Argumentliste in dem Java Block oder der Klasse gespeichert werden. Da diese Elemente keinen Platz für eine Argumentliste bieten, werden diese Informationen in Knoten des GOT AST abgelegt.

Da `per` Blöcke in Klassen als Klassen abgebildet werden, wird als Klassenname dieser Blöcke *per* verwendet.

Da `per` Blöcke keinen neuen Namensraum aufspannen sollen, werden `per` Blöcke in dem normalen OT Compiler unbekannten, GOT AST gespeichert. Dort werden sie ihrer umschließenden Klasse oder Methode zugeordnet.

2.2.4 Callin- und Callout-Bindungen

Die Definition einer Callin- oder Calloutbindung kann in Object Teams auf zwei unterschiedliche Arten dargestellt werden. Dabei ist zwischen einer kurzen und langen Darstellung zu unterscheiden.

In der kurzen Darstellung wird eine Bindung ausschließlich mit Hilfe des Namens der beteiligten Rollen- und Basismethoden definiert. In der langen Darstellung wird die komplette Methodensignatur beider Varianten angegeben. Die lange Darstellung ist vor allem dann sinnvoll, wenn zwei Basis oder Rollenmethoden mit gleichem Namen aber unterschiedlicher Signatur existieren. Ebenso bietet die lange Darstellung durch die Offenlegung der Parameter die Möglichkeit ein Parametermapping zu definieren.

Um dem Benutzer eine einfache Möglichkeit zu bieten, Rollenmethoden anhand der Signatur auszuwählen und dennoch nicht auf eine einfache Auswahl der Basismethoden mit Hilfe einer Metavariable zu verzichten, wird für GOT eine neue Mischform angeboten. So kann die linke Seite der Bindung die komplette Methodensignatur der Rollenmethode aufweisen und die rechte Seite nur eine Metavariable, die über die Definition von Queries spezifiziert wird. Diese Mischform ermöglicht die explizite Auswahl einer Rollenmethode mit Hilfe der kompletten Methodensignatur. Außerdem kann damit bei Calloutbindungen, wie sonst nur in der langen Darstellung, eine Rollenmethoden Definition in der Bindung stattfinden.

Listing 2.6 zeigt die Verwendung der neuen Mischform „long - short“ gegenüber den herkömmlichen Varianten.

```
1  ...
2  protected class MyRole {
3    // short - short
4    myRoleMethodShort -> replace ?baseMethod;
5
6    // long - long
7    String myRoleMethodLong(String s) -> replace String ?baseMethod(String s);
8
9    // long - short
10   String myRoleMethodLong(String s) -> replace ?baseMethod;
11 }
12 ...
```

Listing 2.6: Die abgebildete Rolle `MyRole` zeigt die verschiedenen Varianten für die Definition von Callin- und Calloutbindungen in Generic Object Teams. In GOT wurde die letzte Darstellung hinzugefügt, welche eine Mischform der beiden oberen bildet.

3 Das Generic Object Teams Development Tooling (GOTDT)

Generic Object Teams erweitert Object Teams. Es liegt daher nahe, dass auch die Werkzeugunterstützung von GOT, die von OT erweitern sollte. Eine Auseinandersetzung mit dem Object Teams Development Tooling ist daher unabdingbar.

Das OTDT basiert auf der Eclipse Plattform. Es ist eine direkte Erweiterung des Java Development Toolings (JDT) (siehe Eclipse JDT (2011)), welches eine umfangreiche Toolunterstützung für die Entwicklung von Java Programmen bietet. Das Java Development Tooling (JDT) und damit auch das OTDT sind in einzelne Module, genannt Plugins aufgeteilt. Diese Plugins basieren auf dem Equinox Plugin System, welches auf der OSGI Spezifikation in Version 4 basiert (siehe OSGi Alliance (2011) und Eclipse Equinox (2011)). Diese Plugins haben die Möglichkeit die Plattform und andere Plugins um Funktionalität zu erweitern. Dafür werden Schnittstellen in Form von *Extension Points* bereitgestellt, die von anderen Plugins erweitert werden können. Eine Erweiterung an einem Extension Point wird *Extension* genannt.

Grundsätzlich lassen sich die dem JDT und OTDT zugehörigen Plugins für diese Arbeit in zwei Kategorien aufteilen. Plugins, die die Kernfunktionalitäten wie den Compiler beinhalten und Plugins, die für das Userinterface zuständig sind.

Der Object Teams Compiler, welcher sich in dem Plugin `org.eclipse.jdt.core` befindet, ist eine abgeänderte Version des JDT Compilers, die zusätzlich OT Code compilieren kann. Die dafür notwendigen Veränderungen wurden nicht, wie vielleicht zu erwarten wäre, mit Object Teams selbst durchgeführt. Es wurde der Quellcode gebrancht und direkt für die Bedürfnisse von OT angepasst. Die direkte Veränderung des JDT Compilers im `org.eclipse.jdt.core` Plugin ist untypisch für die Entwicklungen auf der Eclipse Plattform. Im Normalfall werden bestehende Plugins über die von ihnen angebotenen Extension Points verändert. Hier wurde jedoch direkt der Quellcode eines bestehenden Plugins verändert. Dabei wurde der Hauptbezeichner des Plugins nicht verändert, um mögliche Abhängigkeiten nicht zu stören. Denn das angepasste Core Plugin besitzt die gesamte Funktionalität des JDT Plugins, plus die für die Compilierung von OT Code notwendige Funktionalität. Eine auf diese Weise durchgeführte Veränderung wirkt sich negativ auf die Evolutionsfähigkeit des Plugins aus. Denn jede Änderung am Basisplugin muss in das angepasste Plugin übernommen werden, um die gesamten Features der Basis aufweisen zu können.

Eine elegantere Lösung würde eine Adaption des bestehenden Core Plugins mit Hilfe von OT/Equinox darstellen. OT/Equinox ist eine Erweiterung des Equinox Plugin Systems, um aus einem Plugin heraus andere Plugins mit Hilfe von OT zu adaptieren. Eine Adaption außerhalb der definierten Extension Points ist damit möglich (siehe Herrmann und Mosconi (2007)). Dieses

Verfahren zur Adaption soll für das GOTDT verwendet werden. Für den GOT Compiler muss das `org.eclipse.jdt.core` Plugin adaptiert werden.

Für die UI Plugins des OTDT wurde bereits Object Teams und OT/Equinox, welches Adaptionen mit OT über Plugingrenzen erlaubt, eingesetzt. Es musste also kein Plugin kopiert und verändert werden. Die OT UI Plugins erweitern mit Aspekten das JDT Plugin `org.eclipse.jdt.ui`.

Die Basis für die Erweiterung des OT Toolings für die Sprache GOT bieten für den Compiler das Plugin `org.eclipse.jdt.core` und für das Userinterface die Plugins `org.eclipse.jdt.ui`, bzw. `org.eclipse.objectteams.ui`.

3.1 OTDT Core

Der Kern des OTDT ist der Compiler. Er ist ein inkrementeller Compiler, der einzelne CompilationUnits, Methoden, Blöcke oder auch ganze Programme in Form von einer Vielzahl von CompilationUnits übersetzen kann. Eine *CompilationUnit* ist die Repräsentation einer (Java)-Datei im Eclipse Compiler. In Eclipse sind Programme und CompilationUnits in Projekte eingebettet. Diesen Projekten können dann *Natures* zugeordnet werden. Natures geben an, welche Art von Quelltext sich in dem zugehörigen Projekt befinden. Für Object Teams wurde eine eigene Nature entwickelt, die eine Java-Datei mit OT Quellcode beschreibt. Jeder Nature können in Eclipse mehrere *Builder* zugeordnet werden, die für den Buildprozess und damit den Compiler zuständig sind. Da Object Teams eine Erweiterung von Java ist und der Java Compiler kopiert und um Funktionalität erweitert wurde, kann der Object Teams Builder auch Java Quellcode übersetzen. Sofern das OTDT installiert wurde, wird jede Java-Datei, die sich in einem Projekt mit einer Java-Nature befindet mit dem OT Compiler übersetzt, da der JDT Compiler durch den OT Compiler ersetzt wurde. Der Builder hat nicht die alleinige Kontrolle über den Compiler. Neben dem Builder, der über das Userinterface aufgerufen wird, greifen noch Reconciler und Elemente des Toolings direkt auf den Compiler zu.

Der Compiler selbst besteht aus Scanner, Parser, Checker (Resolver) und dem eigentlich Compiler, welcher Bytecode erzeugt. Teile des Parsers sind mit Hilfe eines Parsergenerators aus einer Grammatik Definition erzeugt worden. Überraschend ist allerdings die Vielzahl der durch den Compiler verwendeten Datenstrukturen. Zu Erwarten wäre ein einziger Abstrakter Syntax Baum gewesen, der eine für den Compiler effizientere Darstellung des Quellcodes bietet, um darauf Operationen wie die Typüberprüfung auszuführen. Im OT Compiler werden jedoch eine Vielzahl von Datenstrukturen verwendet. Es gibt *drei wichtige ASTs* und zwei zusätzliche Datenstrukturen, die für das Resolving (Checker) benötigt werden.

Der OT (Java) Compiler ist nicht für eine Erweiterung ausgelegt worden. Die Mehrzahl der Klassen befindet sich undokumentiert in internen Packages, die strenge Sichtbarkeitsregeln vorweisen. Es gibt kaum dokumentierte APIs, die eine einfache Erweiterung ermöglichen würden.

3.2 Object Teams Userinterface

Der Quellcode im Userinterface Plugin des OTDT ist weitaus lesbarer und besser dokumentiert. Begründet ist dies durch die Verwendung der Rich Client Platform (RCP), welche nicht nur als Basis des JDT und OTDT Userinterfaces, sondern als weit verbreitete Basis für das Userinterface Eclipse basierter Anwendungen verwendet wird. Dementsprechend existiert ein breites Spektrum an Dokumentation und genau spezifizierte APIs (siehe Eclipse RCP (2011) für mehr Informationen zur RCP).

Da das OTDT Userinterface weitgehend dem des JDT entspricht, wurden die Vorzüge von OT/Equinox und der Aspektorientierung verwendet und das JDT UI mit Hilfe von Teams adaptiert. Außerdem wurde für Object Teams eine entsprechende Nature und Builder über Extension Points hinzugefügt. Über diese Extension Points wurden auch Wizards integriert, die das Erstellen von OT Projekten ermöglichen. OT spezifische Compilerfehler werden ohne Anpassungen im Editor angezeigt und über den Quickfix Extension Point mit semi-automatischen Lösungsvorschlägen verbunden.

3.3 Vom OTDT zum GOTDT

Die Erweiterung des OTDT zum GOTDT erfordert Anpassungen im Core des OTDT und im Userinterface. Jedoch ist ein Großteil der Anpassungen im Core zu erwarten. Es müssen Scanner und Parser für Generic Object Teams angepasst werden. Da der Großteil des Parsers über einen Generator aus einer Grammatikdefinition erzeugt wurde, muss diese Definition erweitert werden. Außerdem muss eine Anpassung des Checkers für das neue Metavariablen Typsystem vorgenommen werden. Zusätzlich muss für die durch GOT hinzugekommenen Elemente eine neue Datenstruktur (AST) erstellt, oder eine bisherige erweitert werden. Um Quellcode im GOTDT einigermaßen komfortabel entwickeln zu können, muss das Syntax Highlightning angepasst, sowie eine Möglichkeit geschaffen werden, GOT Projekte zu erstellen. Implizite Constraints sollen durch den Compiler erkannt und dem Nutzer mit Hilfe von Quickfixes als Lösungsvorschläge unterbreitet werden. Außerdem muss dafür gesorgt werden, dass das GOTDT nur für Projekte verwendet wird, die ausdrücklich auf GOT Features zugreifen wollen.

Die in dieser kurzen Analyse angerissenen Anpassungen werden in den folgenden Kapiteln, beginnend mit den im gesamten Tooling benötigten Datenstrukturen, eingehend erläutert.

4 Datenstrukturen

Der AST ist die wichtigste Datenstruktur im OTDT Tooling. Sie repräsentiert eine Quellcode Datei in einer für das Tooling nutzbaren Form. Die Repräsentation einer Quellcode Datei wird im folgenden als CompilationUnit bezeichnet.

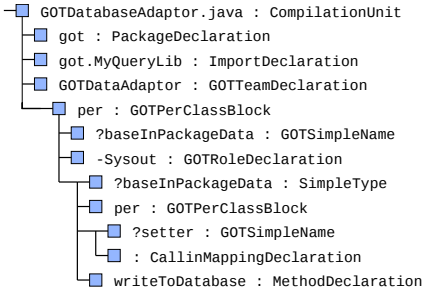
Der AST einer CompilationUnit bietet die Grundlage für eine Vielzahl von Funktionen innerhalb der Entwicklungsumgebung. Im Allgemeinen wird er aus dem Quellcode einer CompilationUnit erstellt. Der AST dient als Repräsentation des Quellcodes für den Compiler. So läuft das Resolving und die Codeerzeugung ausschließlich über den AST. Außerdem wird der AST im Eclipse JDT außerhalb des Compilers für Benutzer unterstützende Operationen verwendet. So bietet er unter anderem die Grundlage für die Outline, Package Explorer, Navigator und Debugger. Zusätzlich werden alle Refactorings auf AST Ebene umgesetzt und in einem späteren Schritt aus diesem in Quelltext umgewandelt. Ebenso bietet die Eclipse JDT Plattform Plugin Entwicklern die Möglichkeit über verschiedene Schnittstellen den AST zu verändern.

In Abbildung 4.1 ist zur Veranschaulichung eine Generic Object Teams CompilationUnit in textueller und AST Repräsentation dargestellt. Die AST Repräsentation ist stark vereinfacht. Der komplette AST hat 90 Knoten.

Zu den wichtigen ASTs in Eclipse zählen der Compiler AST, DOM AST, und das Java Model. Erwähnenswerte weitere Datenstrukturen sind Bindings und Scopes. Alle diese Strukturen und ihre Verbindungen zueinander, sowie ihre Relevanz für das GOTDT werden in diesem Kapitel behandelt.

```
1  package got;
2
3  import otquery got.MyQueryLib;
4
5  public team class GOTDatabaseAdaptor
6    ...
7  {
8    per(?baseInPackageData)
9    {
10     protected class -Sysout
11       playedBy ?baseInPackageData
12     {
13       per(?setter{
14         void writeToDatabase(Object data)
15           <- after
16           void ?setter(?dataType data);
17
18         void writeToDatabase(Object data)
19         {
20           DB.getDefault().commit(data);
21         }
22       }
23     }
24   }
25 }
```

(a) Das Listing zeigt einen Aspekt der alle setter Methoden aller Klassen innerhalb eines Packages adaptiert und das ihnen übergebene Objekt in einer Datenbank speichert.



(b) Die gezeigte Abbildung zeigt eine vereinfachte AST Repräsentation des in Listing 4.1a dargestellten Code Fragments.

Abbildung 4.1: Die beiden dargestellten Abbildungen zeigen den gleichen Aspekt `GOTDatabaseAdaptor` in verschiedenen Repräsentationen. Links ist die textuelle und rechts die AST Darstellung gezeigt. Zu beachten ist dabei, dass das benötigte `match` Element nicht dargestellt ist. Außerdem ist der AST stark vereinfacht.

4.1 Compiler AST

Der Compiler AST ist die zentrale Datenstruktur des Compilers. Alle für den Compilierungsprozess einer CompilationUnit relevanten Operation verwenden den Compiler AST. In der Regel wird der AST mit Hilfe des Parsers aus dem Quelltext einer Java / OT CompilationUnit erstellt. Alternativ ist es möglich, den Compiler AST aus anderen ASTs im Eclipse JDT zu erstellen. Dafür werden Converter verwendet, die zwischen verschiedenen AST Typen konvertieren können. Nähere Details zu den Convertern sind in Abschnitt 4.6 zu finden.

Der Compiler AST befindet sich im Package `org.eclipse.jdt.internal.compiler.ast`, im Plugin `org.eclipse.jdt.core`. An dem Keyword *internal* lässt sich erkennen, dass der Compiler AST nur für Entwickler des JDT gedacht ist und keine APIs nach außen anbietet. Dementsprechend gestaltet sich die öffentliche Dokumentation als „nicht existent“. Unter allen vorzustellenden ASTs ist der Compiler AST bei Weitem der komplexeste. Dies ist darin begründet, dass sich in den Knoten des AST, ein Großteil der für das Resolving und die Code Analyse befindliche Funktionalität befindet.

Der Basistyp aller AST Knoten ist die Klasse `ASTNode`. Sie stellt für alle Knoten notwendige Daten zur Verfügung. Dazu zählen Quellcode Positionen der textuellen Darstellung des Knotens und eine Menge von Konstanten, die in einer Bitmaske im Feld `bits` kodiert werden können. Diese

Konstanten beinhalten eine Vielzahl von internen Flags die für die Verarbeitung im Compiler benötigt werden.

Eine CompilationUnit wird im Compiler AST mit der Klasse `CompilationUnitDeclaration` darstellt.

4.1.1 Erweiterung des Compiler AST für Generic Object Teams

Generic Object Teams enthält zu Object Teams zusätzliche Sprachelemente. Diese Sprachelemente müssen dem Compiler AST hinzugefügt werden. Es wird davon ausgegangen des der Quelltext des OT Compilers direkt verändert werden kann. Eine Erweiterung des AST erfolgt, indem seine Knotenmenge erweitert wird. Dafür werden neue Knoten erstellt, die entweder bestehende AST Knoten spezialisieren oder direkt vom Knoten `ASTNode` abgeleitet werden. Die so entstandenen, neuen Knoten können dann vom angepassten Parser verwendet werden, um einen GOT AST aufzubauen. Für OT Knoten ist die benötigte Logik zur Erstellung dieser Knoten schon vollständig vorhanden. In den meisten Fällen unterscheiden sich GOT von OT Knoten nur an wenigen Stellen. Infolgedessen ist auch die verwendete Logik zu Erstellung der GOT Knoten ähnlich. Um diese Logik wiederverwenden zu können, müssen an einigen Stellen im Parser zusätzliche Bedingungen eingeführt werden. Außerdem bietet sich stellenweise der Austausch von Konstruktoraufrufen durch ihre spezialisierten GOT Varianten an.

Folge dieser Eingriffe wäre eine Vermischung von GOT-spezifischem und OT-spezifischem Code im Parser. Um dies zu vermeiden und GOT-spezifischen Code im GOT Parser zu bündeln, ohne dabei Wiederverwendung von bestehender Funktionalität zu vernachlässigen, wird ein aspektorientierter Ansatz für die Realisierung des GOT AST verwendet.

Im Folgenden soll das beschriebene Vorgehen auf einen aspektorientierten Ansatz mit Object Teams übertragen werden. GOT Knoten werden mit Hilfe von Rollen auf bestehende OT Knoten definiert. Anstelle einer Superklasse sucht man für den GOT Knoten eine passende Basis. Da jede Rolle nur innerhalb eines Teams existieren kann, müssen alle GOT Rollen in ein Team eingebettet werden. Dieses Team wird im Verlauf dieser Arbeit als GOT AST bezeichnet, da es alle GOT Knoten (Rollen) enthält. Im Gegensatz zum OT AST existiert nicht eine GOT AST Instanz pro CompilationUnit, sondern eine Einzige global für alle CompilationUnits im Compiler. Dies ist möglich, da jede Rolle an eine Basis gebunden wird und in diesem Fall die Basis ein OT Knoten ist, der wiederum einer AST Instanz zugeordnet ist. Über diese Instanz wird der Lifecycle der Basis *und* der Rolle kontrolliert.

Ein visuelle Darstellung des beschriebenen Konzepts zeigt Abbildung 4.2. Die Abbildung zeigt die AST Repräsentation einer Queryklasse mit einer Querymethode. Die rechte Seite der Abbildung zeigt hierbei den vom Parser generierten OT AST mit einer Klasse und einer Methode. Die Klasse hat eine Referenz auf ihre enthaltene Methode. Um beide Knoten als Queryklasse und Querymethode zu identifizieren, wird im GOT AST (links) für beide jeweils eine Rolle (Knoten) erzeugt. Neben der Markierung als GOT Elemente dienen die Rollen als Möglichkeit zur Manipulation des Verhaltens der beiden OT Knoten. Die Struktur des AST wird in fast allen Fällen ausschließlich im OT AST gebildet. Im GOT AST sind nur selten Abhängigkeiten zwischen GOT Knoten definiert.

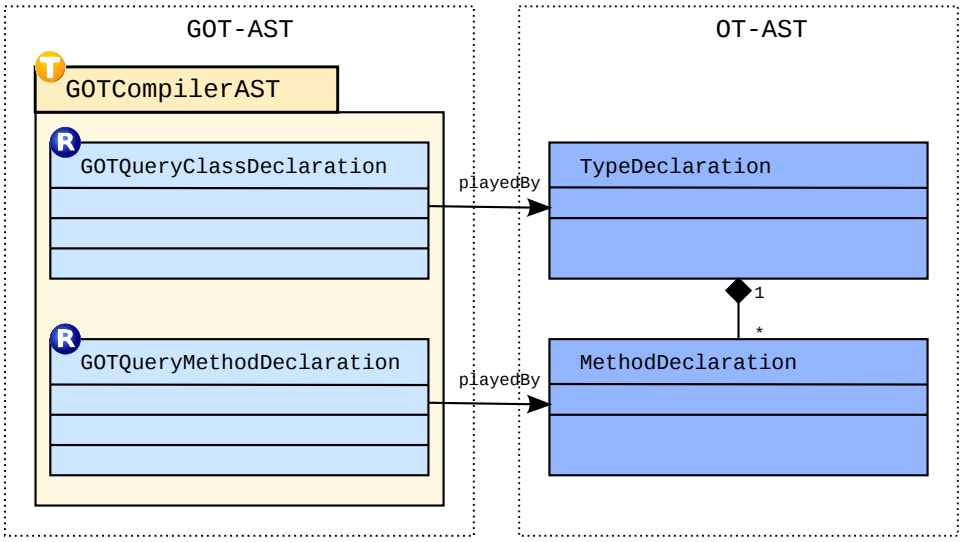


Abbildung 4.2: Die Abbildung zeigt die AST Repräsentation einer GOT Queryklasse mit einer Querymethode. Im OT Compiler AST wird eine Klasse mit Hilfe einer `TypeDeclaration` dargestellt. Eine Methode wird durch eine `MethodDeclaration` dargestellt. Die auf der linken Seite, im GOT AST definierten Rollen markieren die rechts dargestellten OT Knoten als GOT Knoten und damit Queryklasse und Querymethode.

Auf den ersten Blick erscheint dieser Ansatz sehr logisch und einfach. Im Detail entstehen jedoch einige Komplikationen. Das größte Problem ist die dynamische Bindung zwischen Rolle und Basis. Standardmäßig binden sich Rollen an jedes Basisobjekt, welches durch eine Rolle adaptiert wurde und implizit, z. B. durch ein Callin geliftet wird. Dies ist jedoch nicht in allen Fällen erwünscht. Nicht jeder AST Knoten soll ein GOT Knoten werden. So existieren normale Javaklassen und Queryklassen, die jeweils als Basis eine `TypeDeclaration` haben. Es ist wichtig, dass nur einige spezielle Basisknoten zu GOT Knoten werden. Eine Basis darf also nur geliftet werden, wenn sie eine Basis für einen GOT Knoten ist. Diese Lifting Entscheidung kann mit Hilfe von Baseguards an der Rolle getroffen werden. Exemplarisch kann man davon ausgehen, dass ein Identifier einer Basisreferenz, der mit einem '?' beginnt, auf eine Metavariable deutet und es sich somit um eine Basis eines GOT Knoten handelt. Ist dies der Fall wird die Basis geliftet und der GOT Knoten ist einsatzbereit.

Listing 4.1 zeigt den GOT Knoten `GOTSingleTypeReference`. Die Basis des Knotens ist eine `SingleTypeReference`, wie sie hinter dem Keyword `playedBy` für die Definition einer Basis für eine Rolle verwendet wird. Die GOT Knoten Rolle wird nur aktiv, sofern es sich bei der Basis um eine `SingleTypeReference` handelt, deren Name mit einem „?“ beginnt. Somit ist ausgeschlossen, dass eine `SingleTypeReference` zu einer `GOTSingleTypeReference` geliftet wird, die keine Metavariable darstellt.

Diese Erkennung der Basis als Kandidat für eine GOT Knoten Rolle funktioniert nicht in allen Fällen. Oft ist es nicht ohne Weiteres möglich das Basisobjekt als Basisobjekt für die GOT Knoten Rolle zu erkennen. Ein Beispiel dafür ist die Queryklasse. Eine Queryklasse mit einem leeren Body, lässt sich nicht von einer normalen Javaklasse unterscheiden. Beide Klassen werden im Compiler AST durch

```
1 protected class GOTSingleTypeReference playedBy SingleTypeReference
2   base when (base.token[0] == '?') {
3 }
```

Listing 4.1: Das Listing zeigt den GOT Knoten `GOTSingleTypeReference`. Er ist eine Rolle auf die Basis `SingleTypeReference`. Die Besonderheit hier ist der Baseguard, welcher ein Lifting der Basis nur zulässt, sofern der Basisknoten eine Metavariable als Namen verwendet.

eine `TypeDeclaration` dargestellt. Ihr einziger Unterschied existiert im Quelltext als Keyword `otquery`. Dieses Keyword kennt nur der Parser, der daraus entstehende AST Knoten ermöglicht keine Rückschlüsse mehr. Abbildung 4.3 zeigt eine Gegenüberstellung der Queryklasse und einer ähnlichen Javaklasse.

```
1 public otquery class
2   MyQueryClass {
3 }
```

(a) Eine eine Queryklasse mit leerem Body. Das es sich um eine Queryklasse handelt, zeigt der `otquery` Modifier.

```
1 public class
2   MyQueryClass {
3 }
```

(b) Eine einfache Javaklasse mit leerem Body.

Abbildung 4.3: Die Abbildung zeigt die Ähnlichkeit einer Queryklasse zu einer herkömmlichen Javaklasse. Es lassen sich neben dem `otquery` Modifier keine syntaktischen Unterschiede ausmachen.

Informationen, wie das `otquery` Keyword gehen in der Parserphase verloren. Es sollte also schon im Parser die Möglichkeit geben zu entscheiden, ob ein Java AST Knoten als Basis für einen GOT Knoten fungieren wird. Die Eröffnung dieser Möglichkeit impliziert, dass die Bindung zwischen Rolle und Basis explizit festgelegt werden muss. Es soll möglich sein eine GOT Rolle für eine OT Basis explizit zu erstellen. Diese Funktionalität wird in OT als „Explicit role creation“ beschrieben (siehe Object Teams (2011c)).

Innerhalb eines Teams kann der Lifting Konstruktor verwendet werden, um eine neue Rolle für eine Basis zu erstellen. Der Lifting Konstruktor nimmt als erstes Argument immer die der Rolle zugeordneten Basis. Beim manuellen Aufruf des Lifting Konstruktors muss darauf geachtet werden, dass nicht für eine Basis mehrere Rollen erstellt werden.

Alternativ zum direkten Aufruf des Lifting Konstruktor können Rollen mit Hilfe von „Declared Lifting“ erstellt werden (siehe Object Teams (2011b)). Dafür wird eine Team Methode erstellt, die als Parameter die Basis mit Hilfe des `as` Keywords, beim Aufruf der Methode liftet. Es wird entweder eine existierende Rolle für den Parameter, oder eine neue Rolle verwendet. Von außerhalb des Teams kann die Methode mit dem Basisobjekt als Argument aufgerufen werden. Innerhalb der Methode steht das zur Rolle geliftete Rollenobjekt zur Verfügung. Dieser Mechanismus kann ausgenutzt werden um Rollen explizit für eine Basis zu erstellen. Dafür wird im GOT AST für alle so erstellbaren GOT Knoten eine Team Methode dieser Form bereitgestellt. Diese Methoden sind

```
1 public GOTQueryClassDeclaration createGOTQueryClassDeclaration(
2   TypeDeclaration as GOTQueryClassDeclaration qcd) {
3   return qcd;
4 }
```

Listing 4.2: Die dargestellte Methode ermöglicht das explizite Erstellen einer Rolle. Dies geschieht mit Hilfe von *Declared Lifting*.

```
1 protected class GOTSingleTypeReference playedBy SingleTypeReference
2   base when (GOTCompilerAST.this.hasRole(base, GOTSingleTypeReference.class)
3 {
4 }
```

Listing 4.3: Der dargestellte GOT Knoten (Rolle) kann nur implizit geliftet werden, sofern bereits eine Rolle für die Basis vom Typ `GOTSingleTypeReference` im umschließenden Team `GOTCompilerAST` existiert. Die Überprüfung im Baseguard auf Existenz einer Rolle im Team bildet die Grundlage für das *Object Registration* Pattern.

durch die vorangestellte Kennung `create` gekennzeichnet. Listing 4.2 zeigt eine solche `create` Methode für eine `GOTQueryClassDeclaration`.

Die explizite Erstellung von GOT Knoten ist nun möglich. Jedoch muss für die für die explizite Erstellung vorgesehenen Knoten noch die implizite Erstellung deaktiviert werden. Dafür wird mit Hilfe eines speziellen Baseguards das Lifting der entsprechenden Rolle nur erlaubt, sofern das Team bereits eine Rolleninstanz für die Basis besitzt. Dieser Baseguard wird in Listing 4.3 gezeigt. Er bewirkt das die Rolle nur geliftet wird sofern in der Rollenmenge des Teams eine Rolle für die aktuell zu liftende Basis existiert. Mit Hilfe der Angabe einer Klasse, kann diese Abfrage auf Rollen des gewünschten Typs beschränkt werden.

In Object Teams ist das vorgestellte Verfahren zur Kontrolle der Rolle Basis Bindung als *Object Registration* Pattern bekannt.

Alle GOT Knoten des GOT Compiler AST sind im Team `GOTCompilerAST` gebündelt. Neben den GOT Knoten Rollen und den vorgestellten `create` Methoden, enthält der `GOTCompilerAST` noch notwendige Getter und Setter Methoden auf die GOT Knoten. Hierbei wird die Funktionalität der GOT Knoten über Team Methoden nach außen gegeben. Dieses System erspart eine Externalisierung der Rollen und ermöglicht einen leichten Zugriff auf AST Funktionalität; auch für OT unerfahrene Nutzer. Des Weiteren verbindet alle GOT Knoten eine gemeinsame Superklasse `GOTCompilerASTNode`, welche es ermöglicht gemeinsame GOT spezifische Funktionalität für alle Knoten bereitzustellen. Darunter fällt z. B. eine Methode zur Bestimmung des Knotennamens. Listing 4.4 zeigt die einen Ausschnitt aus dem `GOTCompilerAST`.

```
1  public team class GOTCompilerAST {
2    ...
3    protected class GOTQueryClassDeclaration extends GOTCompilerASTNode
4      base when
5      (GOTCompilerAST.this.hasRole(base, GOTQueryClassDeclaration.class) {
6      ...
7    }
8
9    public GOTQueryClassDeclaration createGOTQueryClassDeclaration(
10     TypeDeclaration as GOTQueryClassDeclaration qcd) {
11     return qcd;
12   }
13
14   public boolean isGOTQueryClassDeclaration(org.eclipse.jdt.internal.
         compiler.ast.TypeDeclaration td) {
15     if(td != null && this.hasRole(td, GOTQueryClassDeclaration.class))
16       return true;
17     return false;
18   }
19
20   public org.eclipse.jdt.internal.compiler.ast.MethodDeclaration
         getGOTMatchDeclaration(
21     TypeDeclaration as GOTTeamDeclaration declr) {
22     return declr.getMatch();
23   }
24   ...
25 }
```

Listing 4.4: Der `GOTCompilerAST` ist die Haupt-Datenstruktur des GOT Compilers. Hier gezeigt ist ein Ausschnitt mit der Definition eines Queryklassen Knotens und der zugehörigen Methode zur Erstellung von selbigem. Außerdem ist eine Methode gezeigt, die dem Aufrufer mitteilt, ob die von ihm übergebene Basis, teil des GOT AST ist. Die letzte Methode ermöglicht es direkt anhand einer übergebenen Basis für eine `GOTTeamDeclaration` die Basis der zugehörigen Match Deklaration zu erhalten.

GOT Compiler AST Knoten

Tabelle 4.1 zeigt alle Knoten im GOT Compiler AST und deren zugehörige Basisknoten im OT AST, samt Kontext. Die Besonderheiten und Aufgaben der GOT Knoten werden im Folgenden kurz erläutert. Die Knoten sind als Rollen im `GOTCompilerAST` Team eingebettet.

Alle Knoten

Alle Compiler AST Knoten erben von der Klasse `ASTNode`. Die Superklasse aller GOT AST Knoten ist die Rolle `GOTASTNode` mit der Basis `ASTNode`. Diese Rolle identifiziert einen GOT Knoten und stellt gemeinsame Funktionalität zur Verfügung. Wie z. B. eine Methode, die überschrieben werden kann und den Namen des aktuellen Knotens liefert. Außerdem ist die `GOTASTNode` Rolle zuständig für die Entkapselung der für eine Mehrzahl von GOT Knoten benötigten Felder und Methoden der `ASTNode` Klasse.

Knoten	Basisknoten	**Zugehörigkeit**
GOTASTNode	ASTNode	Oberklasse aller GOT Knoten
GOTCompilationUnitDeclaration	CompilationUnitDeclaration	CompilationUnit
GOTTeamDeclaration	TypeDeclaration	GOT Team
GOTMatchDeclaration	MethodDeclaration	GOT Team
GOTDeclareDeclaration	MethodDeclaration	GOT Team
GOTMetaFieldDeclaration	FieldDeclaration	GOT Team
GOTClassDeclaration	TypeDeclaration	per Block
GOTMethodDeclaration	GOTMethodDeclaration	per Block
GOTPerBlock	/	per Block
GOTPerClassBlock	TypeDeclaration	per Block
GOTAbstractMethodMappingDeclaration	AbstractMethodMappingDeclaration	Callin- und Calloutbindungen
GOTCallinMappingDeclaration	CallinMappingDeclaration	Callin- und Calloutbindungen
GOTCalloutMappingDeclaration	CalloutMappingDeclaration	Callin- und Calloutbindungen
GOTMethodSpec	MethodSpec	Callin- und Calloutbindungen
GOTFieldAccessSpec	FieldAccessSpec	Callin- und Calloutbindungen
GOTQueryClassDeclaration	TypeDeclaration	Queryklassen
GOTQueryMethodDeclaration	MethodDeclaration	Querymethoden
GOTQueryExpression	Expression	Queries
GOTSingleTypeReference	SingleTypeReference	Metatypen
GOTSingleNameReference	SingleNameReference	Metavariablen
GOTQualifiedNameReference	QualifiedNameReference	Metavariablen

Tabelle 4.1: Die Tabelle zeigt alle GOT Compiler AST Knoten und deren Zugehörigkeit zu den Sprachelementen der Sprache Generic Object Teams.

CompilationUnit

Jede GOT CompilationUnit wird durch den Knoten `GOTCompilationUnitDeclaration` repräsentiert. Dieser Knoten hat als Basis eine CompilationUnitDeclaration. Aufgabe des Knotens ist die Kontrolle der Code Erzeugung. Beim Aufruf der `generateCode` Methode des Knotens werden rekursiv alle Kinder aufgefordert sich selbst in Bytecode umzuwandeln. Da die Code Erzeugung bei CompilationUnits die eine `GOTTeamDeclaration` enthalten und somit für die Source-to-Source Transformation vorgesehen ist, keinen Sinn macht, wird diese hier abgebrochen.

Das GOT Team

Ein GOT Team bietet die Grundlage für die Verwendung von Metavariablen. Eine solches Team wird durch den Knoten `GOTTeamDeclaration` dargestellt. Dieser Knoten enthält eine Referenz auf entweder eine `GOTMatchDeclaration` oder `GOTDeclareDeclaration`, welche im Scope des Teams verwendbare Metavariablen deklarieren und mit Hilfe von Queries belegen. Diese deklarierten Metavariablen sind vom Typ `GOTMetaFieldDeclaration`. Die für ein GOT Team verwendeten Knoten werden, dank den beteiligten Metavariablen anders aufgelöst als herkömm-

liche OT AST Knoten. Ein Teil der dafür benötigten Funktionalität befindet sich ebenfalls in den GOT Knoten. Das Resolving und die Rolle der aufgelisteten Knoten wird in Abschnitt 5.4 detailliert erläutert.

Der `per` Block

Der `per` Block ermöglicht feingranulare Kontrolle über die Erzeugung von OT Code aus GOT Code. Er wird mit Hilfe von zwei verschiedenen Knoten, wie in Abschnitt 2.2.3 erläutert, dargestellt. Diese Knoten lauten `GOTPerMethodBlock` und `GOTPerClassBlock`. Klassen, die einen per Block enthalten, werden über den Knoten `GOTClassDeclaration` realisiert. Methoden dazu analog mit dem Knoten `GOTMethodDeclaration`. Da jeder `per` Block noch zusätzlich eine Menge von Argumenten beinhalten kann und diese sowohl bei der Klassen- als auch der Methodenrepräsentation gleich ist, wird dafür eine gemeinsame Superklasse `GOTPerBlock` verwendet. Auch der `per` Block und seine beteiligten AST Knoten enthalten Resolving spezifische Funktionalität.

Callin- und Callout-Bindungen

Neben der Anpassung von bestehenden Callin- und Callout-Bindungen für GOT, muss auch die in Abschnitt 2.2.4 erläuterte, neue Darstellungsform im AST widergespiegelt werden können. Die Gemeinsamkeiten zwischen Callin- und Calloutbindungen werden mit Hilfe einer gemeinsamen Superklasse `GOTAbstractMethodMappingDeclaration` abgebildet. Die rechte Seite einer Callin- Calloutbindung, die Metavariablen enthält, wird mit Hilfe der Knoten `GOTMethodSpec` und `GOTFieldAccessSpec` abgebildet. Durch die mögliche Verwendung von Metavariablen, muss das Resolving in den besagten Knoten angepasst werden. Mehr dazu ist in Abschnitt 5.4 zu finden.

Queryklassen

Die Repräsentation von Queryklassen übernimmt der Knoten `GOTQueryClassDeclaration` im GOT AST. Mit Hilfe dieses Knotens können existierende OT Klassen Knoten als Queryklassen markiert werden.

Querymethoden

Neben Queryklassen müssen auch Querymethoden von regulären Javamethoden unterscheidbar sein. Für diese Aufgabe beinhaltet der GOT Compiler AST den `GOTQueryMethodDeclaration` Knoten.

Queries

Queries werden ebenfalls mit ihrem eigenen Knoten im GOT Compiler AST repräsentiert. Der Knoten für eine Queryexpression heißt `GOTQueryExpression`.

Metavariablen

Den Schluss der GOT Knoten bilden jene, die für die Darstellung von Metavariablen zuständig sind. Metatypen werden durch die `GOTSingleTypeReference` dargestellt. Metavariablen werden durch die `GOTSingleNameReference` repräsentiert. Die Kombination von Metavariablen und herkömmlichen Namen zu qualifizierten Metareferenzen übernimmt der GOT Knoten `GOTQualifiedNameReference`. All diese aufgezählten Referenz Knoten verwenden ein angepasstes Resolving für Metavariablen (siehe Abschnitt 5.4).

4.2 DOM AST

Der Compiler AST ist das Herz des Compilers. Für große Teile des Toolings und die Verwendung von außerhalb des Compilers steht jedoch der DOM AST zur Verfügung. Der Compiler AST ist sehr komplex und mächtig und in seiner Struktur auf die Bedürfnisse des Compilers und vor allem des Resolvings angepasst. Der Compiler AST wird im Parser erstellt und danach nur wenig in seiner Struktur verändert.

Für Veränderung, Transformation und externen Zugriff wurde der DOM AST gemäß des *Document Object Model* entwickelt. Der DOM AST bietet ein umfangreiches API nach außen.

Jeder AST Knoten hat als Superklasse die Klasse `ASTNode`. Eigner eines jeden DOM AST ist die Klasse `org.eclipse.jdt.core.dom.AST`. Die Klasse `AST` besitzt eine beliebige Anzahl an `ASTNode` Knoten, die jeweils einen Link auf die für sie zuständige AST Instanz haben. Dabei ist die Struktur der Knoten beliebig. Eine AST Instanz kann beliebige Unterbäume haben. Die Klasse AST dient außerdem als Factory um Knoten zu erzeugen. Der Root Knoten eines jeden Unterbaumes ist über seine Kinder mit der Methode `ASTNode.getRoot()` abrufbar. In den meisten Fällen ist der Root Knoten eines DOM AST ein Knoten vom Typ `CompilationUnit`. Er repräsentiert mit seinen Kindern eine Java / OT Quellcode Datei.

Eine Besonderheit des DOM AST sind seine Strukturmöglichkeiten. Im Gegensatz zum Compiler AST sind die Beziehungen zwischen Vater- und Kindknoten und seine Eigenschaften nicht ausschließlich statisch definiert. Über ein generisches Konzept ist die Definition von neuen Beziehungen und Eigenschaften möglich. So ist es denkbar zur Laufzeit einer bestehenden `TypeDeclaration` neue Kindknoten zuzuweisen und diese im Knoten für Aufrufer zu registrieren. Diese generischen Beziehungen werden Properties genannt. Das Konzept der Properties spielt vor allem für die Adaption für GOT eine wesentliche Rolle (siehe Abschnitt 4.2.2).

Parallel zu Properties bieten zusätzlich alle Knoten über Methoden direkten Zugriff auf statisch festgelegte Beziehungen. So liefert die Methode `TypeDeclaration.getMethods()` die alle Methoden eines Typs.

Der DOM AST implementiert das Visitor Pattern, welches es ermöglicht den gesamten AST abzulaufen und für jeden Knoten eine vom Nutzer zu bestimmende Funktion auszuführen. Nutzerdefinierte Visitor sollten vom `ASTVisitor` abgeleitet werden.

4.2.1 Erzeugung und Manipulation

Die Erzeugung eines DOM AST kann über verschiedene Wege erfolgen. Er und seine Knoten können direkt über Factory Methoden der Klasse `AST` erzeugt werden. Oder er wird mit Hilfe der Klasse `ASTParser` aus einer Quelldatei erzeugt. Der `ASTParser` startet dafür den Compiler, lässt einen Compiler AST erstellen und wandelt diesen dann in einen DOM AST um. Der dafür verwendete Converter wird in Abschnitt 4.6 beschrieben.

Veränderung des AST erfolgt entweder direkt auf dem `AST`, oder über die Klasse `ASTRewriter`. Die direkte Manipulation ist am komfortabelsten auf einer zuvor durch den `ASTParser` erstellten `CompilationUnit` durchzuführen. Dafür wird mit der Methode `recordModifications` ein Mechanismus auf der `CompilationUnit` in Gang gesetzt, der alle Änderungen auf der `CompilationUnit` und deren Kinder protokolliert. Das so erstellte Protokoll wird mit Hilfe der Methode `rewrite` verwendet, um die Änderungen auf den Quellcode anzuwenden.

Der `ASTRewriter` ist jedoch die zu bevorzugende Variante zur Veränderung des AST. Er arbeitet Kopien des AST und protokolliert Änderungen auf diesen. Es ist daher möglich auch mehrere Änderungen auf dem gleichen AST in separaten Logs zu bündeln. Erzeugt wird der `ASTRewriter` aus einer `AST` Instanz. Alle Veränderungen auf dem `ASTRewriter` AST müssen mit speziellen Logmethoden protokolliert werden.

4.2.2 Erweiterung des DOM AST für Generic Object Teams

Der DOM AST enthält alle Informationen des Compiler AST, ist von außerhalb des JDT leicht zugänglich und verfügt über ein ausgereiftes API. Er bietet damit die richtige Grundlage, um als Datenstruktur für das durch den GOT Compiler erzeugte Compilier Endergebnis zu dienen. Mit Hilfe des durch den Compiler erzeugten DOM AST kann die Umwandlung von GOT in OT Code durchgeführt werden.

Leider ist auch der DOM AST, ähnlich dem Compiler AST nicht für die Erweiterung um neue Knoten entwickelt worden. Eine Adaption mit herkömmlichen Java Mitteln gestaltet sich daher ähnlich kompliziert, wie die Adaption des Compiler AST. Daher wird für die Adaption des DOM AST analog zum Compiler AST der aspektorientierter Ansatz mit Object Teams gewählt.

Die Umsetzung mit OT erfolgt analog zu der des Compiler AST. Alle DOM AST GOT Knoten werden als Rollen innerhalb des Teams `GOTDomAST` realisiert. Analog zu den in Abschnitt 4.1.1 beschriebenen Konzept, wird auch für DOM AST Knoten das *Object Registration* Pattern eingesetzt. Da der DOM AST aus dem Compiler AST erzeugt wird und er den gleichen Quellcode darstellen soll, wie der Compiler AST, sind die Knoten der ASTs ähnlich. Tabelle 4.2 zeigt alle Compiler Knoten und deren Äquivalent im DOM AST. Nicht alle Knoten müssen in beiden ASTs vorkommen. Die ASTs sind auf ihren jeweiligen Verwendungszweck ausgerichtet und sind daher nicht exakt Deckungsgleich. Exemplarisch dafür sind die `GOTMethodDeclaration` und `GOTClassDeclaration`. Da im GOT AST das direkte Hinzufügen von eigens definierten Kindern möglich ist, kann der per Block direkt in die entsprechende `MethodDeclaration` oder `TypeDeclaration` eingefügt werden. Die Container `GOTMethodDeclaration` und `GOTClassDeclaration` sind daher überflüssig.

GOTCompilerAST	GOTDomAST
GOTASTNode	GOTASTNode
GOTCompilationUnitDeclaration	–
GOTTeamDeclaration	GOTTeamDeclaration
GOTMatchDeclaration	GOTMatchDeclaration
GOTDeclareDeclaration	GOTDeclareDeclaration
GOTMetaFieldDeclaration	GOTMetaFieldDeclaration
GOTClassDeclaration	–
GOTMethodDeclaration	–
GOTPerBlock	–
GOTPerClassBlock	GOTPerClassBlock
GOTPerMethodBlock	GOTPerMethodBlock
GOTAbstractMethodMappingDeclaration	–
GOTCallinMappingDeclaration	–
GOTCalloutMappingDeclaration	–
GOTMethodSpec	–
GOTFieldAccessSpec	–
GOTQueryClassDeclaration	GOTQueryClassDeclaration
GOTQueryMethodDeclaration	GOTQueryMethodDeclaration
GOTQueryExpression	GOTQueryExpression
GOTSingleTypeReference	–
GOTSingleNameReference	GOTSimpleName
GOTQualifiedNameReference	GOTQualifiedName
–	GOTMetaType
–	GOTRoleTypeDeclaration

Tabelle 4.2: Die Tabelle zeigt alle GOT Compiler AST Knoten und deren Äquivalent im GOT DOM AST. Nicht jeder Knoten, der in dem einen AST benötigt wird, muss auch im anderen existieren. Beide ASTs sind auf ihren Verwendungszweck ausgelegt und beinhalten die dafür notwendigen Knoten.

Properties

Im Gegensatz zum Compiler AST, bietet der DOM AST das Konzept von Properties. Dieses ermöglicht eine vom Compiler AST unterschiedliche Adaption. Eine Property ist eine Eigenschaft eines AST Knotens. Alle Eigenschaften, darunter auch Kindbeziehungen eines DOM AST Knotens werden in einer Liste von Properties im Knoten gespeichert. Diese Liste kann zur Laufzeit um neue Elemente ergänzt werden. Dieser Umstand ermöglicht die direkte Manipulation eines OT DOM AST Knotens auf die Struktur eines GOT Knotens. Daraus folgt, dass Plugins, die die Properties Funktionalität verwenden auch direkt auf GOT Eigenschaften zugreifen können. Es muss dafür nicht der Umweg über den `GOTDomAST` gegangen werden. Der Aufrufer benötigt *keine* Kenntnis von GOT. Exemplarisch für solch ein Programm ist das `org.eclipse.jdt.astview` Plugin. Es wurde für die Darstellung eines Java AST entwickelt, kann jedoch selbst einen GOT AST vollständig und korrekt darstellen.

Für die Verwendung von Properties werden einige Anpassung notwendig. Der Aufrufer, der alle Eigenschaften eines Knotens abfragt, muss falls es sich bei diesem Knoten um eine Basis eines DOM AST Knotens handelt, eine veränderte Propertyliste erhalten. Diese Liste muss die zusätzlich für GOT enthaltenen Informationen enthalten.

Alle Properties eines Knotens werden über die Methode `ASTNode.structuralProperties ForType` abgerufen. Diese Methode gibt eine Liste mit `StructuralPropertyDescriptor` Objekten zurück. Der Aufruf wird mit Hilfe eines Callins abgefangen und die Ergebnisliste um GOT Eigenschaften erweitert. Um den kompletten Baum, samt GOT Elementen mit dem Visitor Pattern ablaufen zu können, muss ein weiteres Callin die Methode `ASTNode.accept0(ASTVisitor)` adaptieren. Dabei wird der `accept` Aufruf an alle Kinder des GOT Knotens weitergegeben.

4.3 Java Model

Der Compiler und DOM AST sind komplexe, schwergewichtige Datenstrukturen. Für viele Anwendungen sind diese ASTs zu komplex und träge. Als Beispiel seien der Package Explorer, Navigator und die Outline genannt. Diese Elemente benötigen nur einen geringen Teil des gesamten AST. Für sie wird ein vereinfachter, leichtgewichtiger, schnell anzupassender AST benötigt. Diese Anforderung erfüllt das Java Model. Der Großteil einer Java Datei besteht aus Statements. Für die Struktur einer Java Datei sind diese jedoch nicht von Bedeutung. Sie werden daher vom Java Model ignoriert. Dafür bietet der Parser einen speziellen Modus, genannt *diet-Parsing*, der alle Statements überliest. Details dazu bietet Abschnitt 5.3.3.

Für die Anzeige im Package Explorer sind neben den Sprachelementen die auch der DOM AST bietet, Strukturen oberhalb einer CompilationUnit von Bedeutung. Darunter Fallen z. B. Packages und Projekte. Die Knoten des JavaModel werden über Interfaces identifiziert. Zur Auswahl stehen folgende Typen:

- IJavaProject
- IPackageFragmentRoot
- IPackageFragment
- ICompilationUnit
- IClassFile
- IImportDeclaration
- IType
- IField
- IInitializer
- IMethod

Die Erzeugung des Java Model erfolgt mit Hilfe des Parsers und über Converter, die zwischen den verschiedenen AST Strukturen umwandeln können (siehe Abschnitt 4.6).

Das Java Model wurde nicht für Generic Object Teams adaptiert. GOT verändert die Struktur nur innerhalb einer CompilationUnit. Eine Anpassung des Java Model wäre also vorrangig für die Outline von Nutzen. Da diese jedoch eher eine zusätzliche, nicht zwingend notwendige Funktionalität bietet, ist eine Adaption zu vernachlässigen. Begründet durch die nicht-invasive Veränderung des

Compiler AST ist das daraus generierte Java Model gültig, zeigt jedoch nicht alle Features von GOT an. Diese Einschränkung ist für den in dieser Arbeit erstellten Prototyp vertretbar.

4.4 Bindings

Mit den in vorherigen Abschnitten beschriebenen Strukturen können alle Sprachelemente einer Quellcode Datei abgebildet werden. Für die weitere Compilierung und eine statisch Typisierung müssen jedoch zusätzlich zu dieser Abbildung Namen eindeutig Typen, Variablendeklarationen oder bereits compilierten Typen zugeordnet werden. Diese Zuordnungen werden im Eclipse JDT Bindings genannt. Bindings sind im Gegensatz zum Compiler und DOM AST nicht als Baum, sondern als Graph modelliert, der auch zyklische Referenzen zulässt. Sie ermöglichen damit eine gegenseitige Referenzierung, die die Navigation und Überprüfung von Typen erleichtert.

Neben den in der CompilationUnit enthaltenden Typen, werden in den meisten CompilationUnits weitere Typen referenziert. Darunter fallen zum einen ebenfalls im Quelltext vorliegende Typen und zum anderen bereits compilierte Typen. Da diese beiden Typen sehr unterschiedlich sind, werden dafür zwei verschiedene Bindings bereitgestellt: `SourceTypeBinding` und `BinaryTypeBinding`. Für Features wie Methoden (`MethodBinding`) und Felder (`FieldBinding`), Imports (`ImportBinding`) und lokalen Variablen (`LocalTypeBinding`) stehen ebenfalls Knoten im Bindinggraph zur Verfügung.

Die Erzeugung der `SourceTypeBindings` und darin enthaltenden Bindings wird vom Compiler nach der vollständigen Parsierung in der Resolvingphase vorgenommen. Die Auswertung der erkannten Imports, führt zur Erzeugung der `BinaryTypeBindings`. Der entstandene Bindinggraph dient zur Auswertung aller in der CompilationUnit enthaltenen Referenzen, wie z. B. `TypeReference` und `NameReference`. Sowohl Knoten des Compiler AST als des DOM AST referenzieren den Bindinggraph, um eine eindeutige Referenzierung zu ermöglichen. Das Resolving ist beendet, wenn für alle Elemente der zu compilierenden CompilationUnits Bindings gesucht wurden.

Für den aktuellen Generic Object Teams Prototyp ist es nicht notwendig spezielle Bindings zu definieren, da alle GOT Sprachelemente mit Java Elementen abgebildet und deren Bindings verwendet werden können. In einer weiteren Entwicklungsstufe des GOTDT könnte es bei einem Einzug der Codetransformation in den Compiler durchaus sinnvoll sein, eigene Bindings für Metatypen hinzuzufügen. Diese könnten als Repräsentant für eine Familie von Typen auftreten.

4.5 Scopes

Um Namen in Abhängigkeit ihres Sichtbarkeitbereichs aufzulösen und entsprechende Bindings zu erstellen, bietet das Eclipse JDT eine zusätzliche Datenstruktur. Die sogenannten Scopes. Scopes beschreiben einen Sichtbarkeitsbereich oder auch Namespace. Scopes ermöglichen es in einem Java (OT) Programm den selben Namen mit unterschiedlicher Bedeutung, abhängig vom jeweiligen Kontext, zu verwenden.

Für Scopes im JDT gibt es die Oberklasse (`Scope`), von der diverse spezielle Scopes erben. Dazu zählen unter anderem der `CompilationUnitScope`, `ClassScope`, `MethodScope`, `BlockScope`, `OTClassScope` und `CallinCalloutScope`. Diese Scopes bieten die Grundlage um Bindings für Namen zu erstellen. Die verschiedenen Scopes werden ineinander geschachtelt. Dies ermöglicht die Suche nach einer Zuordnung eines Namens im speziellsten Scope. Wird keine Zuordnung im Scope gefunden wird der umgebene Scope durchsucht. Diese Suche wird rekursiv fortgesetzt, bis kein umgebener Scope mehr existiert.

Typen werden über das `LookupEnvironment` gesucht. Dieses bietet sowohl Source als auch Binarytypen. Ein Cache ermöglicht einen schnelleren wiederholten Zugriff.

4.6 Converter

Neben der Möglichkeit einen AST mit Hilfe des Parsers aus dem Quelltext zu erstellen, ist es ebenso möglich, einige Strukturen aus anderen abzuleiten. So wird der DOM AST mit Hilfe eines Converters aus dem Compiler AST und den darin enthaltenen Bindings erstellt. Auch das Java Model wird über eine Art Converter aus dem Compiler AST erstellt.

Für GOT wichtig, ist vor allem der Converter, der aus dem Compiler AST, den DOM AST erstellt. Dieser Converter (`ASTConverter`) hat eine ähnliche Stellung zum DOM AST, wie der Parser zum Compiler AST. Er ist hauptverantwortlich für die Erzeugung von AST Knoten. Um aus dem GOT Compiler AST einen GOT DOM AST zu erzeugen, muss der Converter Kenntnis von GOT haben. Daher muss der `ASTConverter` für GOT adaptiert werden. Diese Adaption wird am Ende des Compiliervorgangs in Abschnitt 5.5.1 beschrieben.

5 Compiler

Der Compiler ist der Kern des GOTDT. Er ist verantwortlich für die Umwandlung von Sourcecode in Bytecode. Im Fall von GOT ist das gewünschte Endprodukt kein Bytecode, sondern ein vollständig aufgelöster, detaillierter DOM AST. Dieser AST kann dann in weiteren Schritten als Quelle zum Generieren von OT Code verwendet werden.

Die Umwandlung von Sourcecode in den vollständigen DOM AST erfolgt in mehreren Teilschritten. Eine vereinfachte Übersicht über die einzelnen Teilschritte wird in Abbildung 5.1 gezeigt. Die Abbildung gibt einen Überblick über den für GOT relevanten Compilierungsprozess. Detailliertere Informationen über den JDT- und OT-Compilierungsprozess gibt die Diplomarbeit von Markus Witte (siehe Witte (2003)).

Abbildung 5.1 zeigt eine Übersicht über den Compilierungsprozess. Eine CompilationUnit besteht aus mindestens einem Typen. Beim Compilieren dieser CompilationUnit wird zuerst eine Typhierarchie aufgebaut. Dafür werden alle Typen und Imports, die in der CompilationUnit enthalten sind, mit Ausnahme von Methodenrümpfen, in Schritt 1.1 geparst. Im folgenden Schritt 1.2 werden alle bekannten Typen, Imports und Features aufgelöst und mit anderen Typen aus CompilationUnits oder `class` Dateien referenziert. Werde dabei neue Typen gefunden, werden für diese ebenfalls Schritt 1.1 bis 1.3 ausgeführt. Im Laufe der Erstellung der Typhierarchie werden etwaige Typfehler erkannt und für den Aufrufer gesammelt.

In Phase 2 werden alle ausstehenden Methodenrümpfe der sich in der CompilationUnit befindlichen Klassen geparst. Die somit hinzugekommenen Statements und die darin enthaltenen Namen werden in Schritt 2.2 aufgelöst und auf eventuelle Typfehler überprüft.

In Phase 3 werden auf dem bis dahin vollständig aufgelösten AST zusätzliche semantische Analysen ausgeführt. In Schritt 3.2 wird, wenn möglich aus dem erzeugten AST Java Bytecode erzeugt. Im letzten Schritt wird der DOM AST aufgebaut.

Die in den einzelnen Compilerphasen beschriebenen Mechanismen (Scanner, Parser, Resolving, DOM AST Erzeugung) und deren Erweiterung für die Sprache GOT werden in diesem Kapitel näher erläutert.

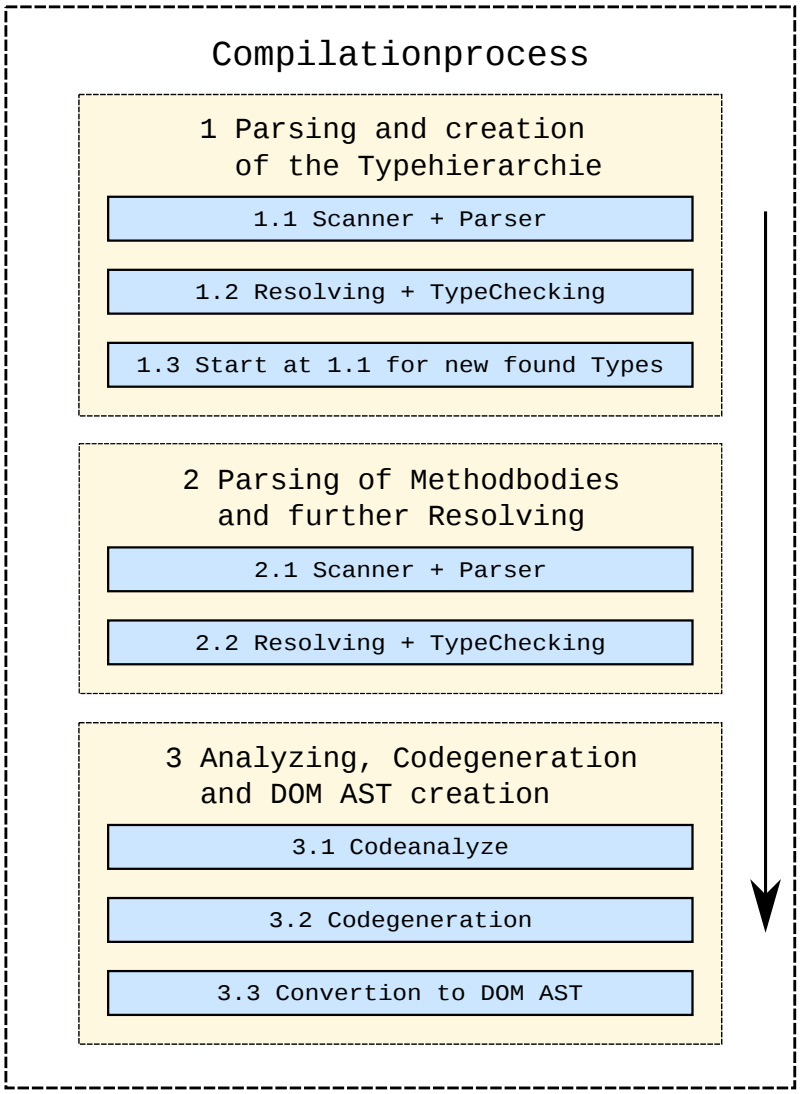


Abbildung 5.1: Die Abbildung zeigt den vereinfachten Compilierungsprozess im OTDT (GOTDT).

5.1 Grammatik

Die ersten Schritte vom Quelltext zum AST übernehmen der Scanner und Parser, welche im Compilierungsprozess in den Phasen 1.1 und 2.1 zum Einsatz kommen (siehe Abbildung 5.1). Sie sind für die initiale Erstellung des AST verantwortlich. Basis für Scanner und Parser bildet eine Grammatik, die die Syntax der zu parsierenden Sprache beschreibt. OT ist eine Teilmenge von GOT. Es ist daher sinnvoll, die Object Teams Grammatik für Generic Object Teams zu erweitern. Die dafür notwendigen Erweiterungen werden im Folgenden beschrieben.

```
1  $Terminals
2    a
3    b
4  $Start
5    S
6  $Rules
7    S -> C
8    C -> D
9    C ::= C 'a'
10   D ::= 'b'
```

Listing 5.1: Das Listing zeigt eine einfache Grammatik in LPG Parsergenerator Syntax. Gezeigt werden die drei wichtigen Bereiche `$Terminals`, `$Start` und `$Rules` der Grammatik. Die hier gezeigte Grammatik erkennt Wörter der Form „b+a{0,1}“ (Posix Regex).

5.1.1 Syntax der Grammatik

Object Teams verwendet den LALR(1) Parsergenerator LPG (siehe Philippe Charles (2011)), um aus einer Grammatik-Definition einen Parser zu erzeugen. Dabei handelt es sich um einen Bottom-Up Shift-Reduce Parser. Diese Parserart parst vom Speziellen zum Allgemeinen, dem Startsymbol. Ein LALR(1) Parser ist ein spezieller LR(1) Parser. Er fasst im Gegensatz zum LR(1) Parser Zustände, die die gleichen Follow Mengen haben, zusammen. Diese Zusammenfassung beschleunigt das Parsen, da weitaus weniger Zustände abgelaufen werden müssen, um das Startsymbol zu erreichen.

Um einen gültigen, deterministischen Parserautomaten zu erzeugen, ist es notwendig, dass die zugrunde liegende Grammatik LALR(1) konform ist. Sie darf somit keine Shift-Reduce, Reduce-Reduce oder Shift-Shift Konflikte erzeugen. Dies bedeutet, dass der Parser keinen Zustand erreichen darf von dem aus es mehrere Wege zum Startsymbol gibt. Eine Besonderheit von LR(1) Parsern im Gegensatz zu LL(1) Parsern ist die Möglichkeit, Linksrekursionen in der Grammatik-Definition zu verwenden. Nähere Details zur Funktionsweise des Parser und Scanner, gibt Abschnitt 5.3.

LPG Syntax

Der LPG Parsergenerator verwendet eine eigene Syntax für die Darstellung der Grammatik. Diese Syntax soll im folgenden beschrieben werden.

Eine LPG Grammatik besteht im wesentlichen aus drei wichtigen Abschnitten. Dazu zählen die Abschnitte `$Terminals`, `$Rules` und `$Start`. Im Abschnitt `$Terminals` werden alle in der Grammatik verwendeten Terminalsymbole aufgelistet. Der Abschnitt `$Rules` beschreibt alle Nonterminals, bzw. Regeln der Grammatik. Im Bereich `$Start` wird die Regel aufgeführt, die zum Startsymbol führt und damit den Parsierungsprozess erfolgreich beendet. Listing 5.1 zeigt eine einfache Grammatik in LPG-Syntax. Besonders zu beachten ist dabei, dass per freier Konvention Terminals in einfache Anführungszeichen gebettet werden. Ebenso werden Regeln, die eine AST Produktion hervorrufen und nicht nur Weiterleitungen auf andere Regeln sind, mit einem „::=“ Operator anstelle eines „->“ Operators gekennzeichnet.

Ziel eines Parsergenerators ist die Erzeugung eines Parsers. Der LPG Parsergenerator erzeugt eine Logik und bietet einen Mechanismus für den Benutzer, um eine eigene Verbindung zum nicht

```
1 GOTQueryClassDeclaration ::= GOTQueryClassHeader GOTQueryClassBody
2 /.$putCase consumeClassDeclaration(); $break ./
```

Listing 5.2: Dargestellt ist eine Grammatikproduktion, die eine `GOTQueryClassDeclaration` erzeugt. Der AST Knoten für die Deklaration wird mit Hilfe der `consumeClassDeclaration` Methode erzeugt. Die Methode wird in der Parserklasse definiert.

generierten Quellcode des Parsers herzustellen. Dafür kann zusätzlich an jeder Produktion der Grammatik ein Makro definiert werden, welches zum Erzeugen von Quelltext verwendet werden kann. In der OT Grammatik wird dieses Makro verwendet, um ein umfangreiches Switch-Statement aufzubauen, welches die in der Logik des Parsers verwendeten Zustand-IDs an Methoden bindet. Jede Produktion erzeugt für dieses Switch Statement mit Hilfe der Makros einen `case` Block. Die dort aufgerufenen Methoden können dann später im Parser deklariert und implementiert werden. Eine solche Regel mit Methodenaufruf wird in Listing 5.2 gezeigt.

5.1.2 Generic Object Teams Grammatik

Aufbauend auf der eingeführten Syntax der LPG Grammatik, sollen im Folgenden die für GOT notwendigen Veränderungen an der OT Grammatik erläutert werden. Es werden sowohl zusätzliche Regeln als auch Erweiterungen bestehender Regeln vorgenommen.

Eine Problematik bei der Grammatik Erweiterung für GOT ist die Einschränkung. Zwar erweitert GOT die Sprache OT im Gesamten, innerhalb der Elemente findet jedoch teils eine Einschränkung statt. So werden für GOT neue Sprachelemente eingeführt, die eine Teilfunktionalität eines bestehenden OT Elementes besitzen. Da das neue GOT Element jedoch nicht die komplette Funktionalität des OT Elementes erben soll, wird es notwendig eine Einschränkung vorzunehmen. Ein Beispiel für solch ein Element wäre eine Expression. Java Expressions sind ausdrucksstärker als GOT Expressions. Leider bietet die Grammatik keine Möglichkeit, ein bestehendes Element in Teilen zu übernehmen. Die einzige Möglichkeit wäre die Einführung eines gemeinsamen Oberelementes, welches den Schnitt des OT und GOT Elementes an Funktionalität bietet. Diese Einführung würde jedoch mit einem immensen Umbau der Grammatik einhergehen und ist daher nicht praktikabel. Nicht zu Letzt, da Veränderungen der Java oder OT Grammatik komplizierter auf GOT übertragen werden können.

Alternativ könnte eine Einschränkung außerhalb der Grammatik im Parser vorgenommen werden. Damit würde jedoch die Makrosyntax Überprüfung zum großen Teil in die Parserlogik verlagert werden. Dies würde zusätzlichen programmatischen Aufwand und Rechenzeit beanspruchen. Um dieser Verlagerung aus dem Weg zu gehen, werden teils sehr zu OT ähnliche GOT Elemente von Grund auf neu definiert. Die Grammatik dadurch etwas länger und komplizierter, jedoch werden weder bestehende Elemente verändert, noch Syntax Einschränkungen aus der Grammatik in den Parser verlagert.

```
1 MetaIdentifier
2 UnBoundMetaIdentifier
3 BoundMetaIdentifier
4 match
5 declare
6 otquery
7 per
```

Listing 5.3: Das Listing zeigt alle für GOT zusätzlich definierten Terminalsymbole. Diese Symbole müssen im Abschnitt `$Terminals` in der Grammatik definiert werden.

Terminals

Die OT Grammatik musste für GOT um einige Terminalsymbole erweitert werden (siehe Listing 5.3). Hervorzuheben sind dabei die Unterscheidungen zwischen `Identifier`, `MetaIdentifier`, `UnBoundMetaIdentifier` und `BoundMetaIdentifier`. Die Einführung der verschiedenen Typen von Metaidentifiern als Terminals erlaubt eine feinere Unterscheidung im Parser. Alternativ wäre es denkbar gewesen, ausschließlich die Definition von Identifiern im Scanner zu erweitern.

Generic Object Teams Produktionen

Im Folgenden werden die für GOT notwendigen Produktionen erläutert. Eine neue Produktion muss definiert und in einem zweiten Schritt in die bestehende OT Grammatik eingebunden werden. Die auf die eigentliche Regel folgend definierten Makros, rufen die entsprechenden Parsermethoden zur AST-Erstellung auf. Dabei werden, wenn möglich, bestehende OT Parsermethoden wiederverwendet. Falls eine solche Wiederverwendung nicht möglich ist, werden neue Parsermethoden für GOT erstellt.

Namen

In der Grammatik sind alle Referenzen auf Variablen oder Typen mit Hilfe von Namen dargestellt. In OT sowie in Java werden zwei verschiedene Typen von Namen verwendet. Entweder handelt es sich um nicht-qualifizierte, einfache Namen (`SimpleName`), oder um qualifizierte Namen (`QualifiedName`). In OT und Java werden diese Namen aus Identifiern gebildet.

In Generic Object Teams ist die Definition von Namen zusätzlich mit Hilfe von Metaidentifiern möglich. Es müssen daher die Definition von einfachen und qualifizierten Namen angepasst werden. Die entsprechenden Regelerweiterungen, sind in Listing 5.4 zu sehen. Die dort gezeigten Erweiterungen machen sich zu Nutze, dass qualifizierte Namen aus einfachen Namen gebildet werden. Es muss daher nur die Definition für einfache Namen angepasst werden.

Imports

Um Bibliotheken von Queries erstellen zu können, die sich über verschiedene Packages und Dateien erstrecken, ist es sinnvoll eine Form von Imports speziell für CompilationUnits, die Queryklassen enthalten, zu unterstützen. Diese Imports sollen sich klar von bisherigen Java und OT-Imports abgrenzen. Dies erfolgt mit Hilfe des vorangestellten `otquery` Modifiers. Um diesen Modifier zu

```
SimpleName -> 'Identifier'
-- {Generic Object Teams
SimpleName -> 'MetaIdentifier'
SimpleName -> 'UnBoundMetaIdentifier'
SimpleName -> 'BoundMetaIdentifier'
-- Jan Marc Hoffmann}

QualifiedName ::= Name '.' SimpleName
/.$putCase consumeQualifiedName(); $break ./
```

Listing 5.4: Das Listing zeigt die für GOT benötigten Änderungen an der Definition von einfachen Namen. Namen in der Grammatik resultieren im AST meist als Typ- oder Variablenreferenzen.

```
----
-- GOTImport Definition

GOTSingleQueryImportDeclaration ::= GOTSingleQueryImportDeclarationName ';'
/.$putCase consumeGOTImportDeclaration(); $break ./

GOTSingleQueryImportDeclarationName ::= 'import' 'otquery' Name
/.$putCase consumeSingleTypeImportDeclarationName(); $break ./

----
-- GOTImport Integration

ImportDeclaration -> StaticImportOnDemandDeclaration
--{Generic Object Teams: otquery import:
ImportDeclaration -> GOTSingleQueryImportDeclaration
-- Jan Marc Hoffmann}
```

Listing 5.5: Gezeigt ist die Definition eines neuen für GOT spezifischen Import Elementes. Das neue Import Element `GOTSingleQueryImportDeclaration` wird den bisherigen Import Elementen hinzugefügt. Der Aufruf der speziellen `consumeGOTImportDeclaration` Methode erlaubt die spezielle Behandlung des GOT Import Elementes bei der AST Knoten Erstellung im GOT Parser.

erkennen, muss die OT Grammatik um ein zusätzliches GOT Import Element erweitert werden. Diese Erweiterung ist in Listing 5.5 zu sehen. Die Erweiterung besteht aus zwei Schritten. Zum einen muss eine neue Regel für einen GOT Import definiert werden und zum anderen muss diese Regel den bestehenden Import Regeln hinzugefügt werden.

Queryklassen

Auf den ersten Blick unterscheiden sich Queryklassen von herkömmlichen Klassen nur in ihrem vorangestellten `otquery` Modifier. Queryklassen dürfen jedoch nur ausschließlich Querymethoden enthalten. Denkbar wäre in der grammatikalischen Umsetzung daher eine einfache Erweiterung der bisherigen Klasse um einen zusätzlichen Modifier. Die Beschränkung auf Querymethoden müsste dann im Parser direkt programmatisch vorgenommen werden und wäre nicht in der Grammatik abgebildet. Wie in der Einleitung dieses Abschnittes beschrieben, soll dieses Vorgehen aus den genannten Gründen jedoch vermieden werden. Daher steht alternativ dazu eine Neudefinition der gesamten Queryklasse in der Grammatik. Dafür müssen zusätzliche Grammatik Regeln eingeführt werden, aber auf eine Einschränkung im Parser kann verzichtet werden.

Ähnlich regulären Klassen sollen auch Queryklassen in Java CompilationUnits definiert werden können. Dies erfolgt über die Erweiterung der `TypeDeclaration` Regel. Die `TypeDeclaration`

```
1  ----
2  -- Queryclass Definition
3
4  GOTQueryClassDeclaration ::= GOTQueryClassHeader GOTQueryClassBody
5  /.$putCase consumeClassDeclaration(); $break ./
6
7  GOTQueryClassHeader ::= GOTQueryClassHeaderName
8  /.$putCase consumeClassHeader(); $break ./
9
10 GOTQueryClassHeaderName ::= GOTQueryModifiersopt otquery class Identifier
11 /.$putCase consumeGOTQueryClassHeaderName(); $break ./
12
13 GOTQueryClassBody ::= '{' GOTQueryClassBodyDeclarationsopt '}'
14
15 GOTQueryClassBodyDeclarationsopt ::= $empty
16 /.$putCase consumeEmptyClassBodyDeclarationsopt(); $break ./
17 GOTQueryClassBodyDeclarationsopt -> GOTQueryClassBodyDeclarations
18
19 GOTQueryClassBodyDeclarations -> GOTQueryClassBodyDeclaration
20 GOTQueryClassBodyDeclarations ::= GOTQueryClassBodyDeclarations GOTQueryClassBodyDeclaration
21 /.$putCase consumeClassBodyDeclarations(); $break ./
22
23 GOTQueryClassBodyDeclaration -> GOTQueryMethodDeclaration
24 /.$putCase consumeClassBodyDeclaration(); $break ./
25
26 ----
27 -- Queryclass Integration
28
29 TypeDeclaration -> EnumDeclaration
30 TypeDeclaration -> AnnotationTypeDeclaration
31 -- {Generic Object Teams: Queryclass
32 TypeDeclaration -> GOTQueryClassDeclaration
33 -- Jan Marc Hoffmann}
```

Listing 5.6: Abgebildet sind die notwendigen Grammatik-Produktionen für die Erweiterung der OT Grammatik um Queryklassen. Eine Queryklasse besteht aus einem Header und einem Body. Ihre Integration in die OT-Grammatik erfolgt über die Erweiterung der Regel `TypeDeclaration` in Zeile 32.

Regel beinhaltet alle Typen, die in einer CompilationUnit vorkommen können. Darunter fallen unter anderem Klassen, Interfaces, Enums und Annotation Deklarationen. Listing 5.8) zeigt die für die Einführung von Queryklassen notwendigen Grammatik-Regeln.

Querymethoden

Der Body einer Queryklasse darf nur Querymethoden enthalten, deren Grammatik-Produktionen folgend beschrieben werden. Eine Querymethode ähnelt einer eingeschränkten Javamethode. Um eine eindeutige Identifizierung schon in der Parserlogik zu ermöglichen, wurde eine eigene Definition für sie entwickelt. Dank des vorangestellten `otquery` Modifiers ist sie eindeutig als Querymethoden zu erkennen. Der Body der Querymethode darf nur lokale Deklarationen und Queries beinhalten. Diese Elemente sind als `GOTQueryBlockStatements` zusammen gefasst. Auf diese Statements wird im nächsten Abschnitt (5.1.2) näher eingegangen. Um diet-Parsing (siehe 5.3.3) auch für Querymethoden zu ermöglichen, wird eine spezielle Regel `GOTQueryNestedMethod` eingeführt, um den Parser auf das Überspringen des Inhalts des Methodenbodies vorzubereiten. Die Grammatik Regeln für Querymethoden sind in Listing 5.7 zu sehen.

```
1  ----
2  -- Querymethod Declaration
3
4  GOTQueryMethodDeclaration ::= GOTQueryMethodHeader GOTQueryMethodBody
5  /.$putCase consumeGOTQueryMethodDeclaration(); $break ./
6
7  GOTQueryMethodHeader ::= GOTQueryMethodHeaderName GOTFormalParameterListopt
       GOTQueryMethodHeaderRightParen
8  /.$putCase consumeMethodHeader(); $break ./
9
10 GOTQueryMethodHeaderName ::= GOTQueryModifiersopt otquery Identifier '('
11 /.$putCase consumeGOTQueryMethodHeaderName(); $break ./
12
13 GOTQueryMethodHeaderRightParen ::= ')'
14 /.$putCase consumeMethodHeaderRightParen(); $break ./
15
16 GOTQueryMethodBody ::= GOTQueryNestedMethod GOTQueryMethodOpenBlock
       GOTQueryMethodBlockStatementsopt GOTQueryMethodCloseBlock
17 /.$putCase consumeMethodBody(); $break ./
18
19 GOTQueryNestedMethod ::= $empty
20 /.$putCase consumeNestedMethod(); $break ./
21
22 GOTQueryMethodOpenBlock ::= '{'
23
24 GOTQueryMethodCloseBlock ::= '}'
```

Listing 5.7: Das Listing zeigt die Grammatik-Regeln für Querymethoden. Eine Querymethode besteht aus einem Header und einem Body. Ein besonderes Augenmerk liegt auf der Regel `GOTQueryNestedMethod` in Zeile 16 (19). Sie ermöglicht es dem Parser, für eventuelles diet-Parsing, den Scanner anzuweisen alle Statements im Body zu überspringen.

```
1  ----
2  -- Querymethod-Body
3
4  GOTQueryMethodBlockStatementsopt ::= $empty
5  /.$putCase consumeEmptyBlockStatementsopt(); $break ./
6  GOTQueryMethodBlockStatementsopt -> GOTQueryMethodBlockStatements
7
8  GOTQueryMethodBlockStatements -> GOTQueryMethodBlockStatement
9  GOTQueryMethodBlockStatements ::= GOTQueryMethodBlockStatements GOTQueryMethodBlockStatement
10 /.$putCase consumeBlockStatements(); $break ./
11
12 GOTQueryMethodBlockStatement -> GOTQueryLocalVariableDeclarationStatement
13 GOTQueryMethodBlockStatement -> GOTQueryExpressionStatement
14
15 GOTQueryLocalVariableDeclarationStatement ::= GOTQueryLocalVariableDeclaration ';'
16 /.$putCase consumeLocalVariableDeclarationStatement(); $break ./
17
18 GOTQueryLocalVariableDeclaration ::= GOTMetaType PushModifiers GOTMetaVariableDeclarators
19 /.$putCase consumeLocalVariableDeclaration(); $break ./
```

Listing 5.8: Das Listing zeigt den Body von Querymethoden. Er enthält lokale Metavariablen Deklarationen und genau einen Query.

Der Body von Querymethoden

Der Body von Querymethoden besteht aus einer beliebigen Anzahl von lokalen Metavariablen Definitionen und genau einem Query. Der Querymethoden Body ist für Querymethoden, `match` und `declare` Sprachelemente gleich.

```
1  ----
2  -- Queryexpression
3
4  GOTQueryExpressionStatement ::= GOTQueryExpression
5  /.$putCase consumeGOTQueryExpressionStatement(); $break ./
6
7  GOTQueryExpression -> GOTQueryAndExpression
8
9  GOTQueryAndExpression -> GOTQueryOrExpression
10 GOTQueryAndExpression ::= GOTQueryAndExpression '&&' GOTQueryOrExpression
11 /.$putCase consumeBinaryExpression(OperatorIds.AND_AND); $break ./
12
13 GOTQueryOrExpression -> GOTQueryNotExpression
14 GOTQueryOrExpression ::= GOTQueryOrExpression '||' GOTQueryNotExpression
15 /.$putCase consumeBinaryExpression(OperatorIds.OR_OR); $break ./
16
17 GOTQueryNotExpression -> GOTQueryPrimary
18 GOTQueryNotExpression ::= GOTPushNot GOTQueryPrimary
19 /.$putCase consumeUnaryExpression(OperatorIds.NOT); $break ./
20
21 GOTPushNot ::= '!'
22 /.$putCase consumePushPosition(); $break ./
23
24 GOTQueryPrimary -> BooleanLiteral
25 GOTQueryPrimary -> GOTQueryMethodInvocation
26 GOTQueryPrimary ::= GOTPushLPAREN GOTQueryExpression GOTPushRPAREN
27 /.$putCase consumePrimaryNoNewArray(); $break ./
28
29 GOTPushLPAREN ::= '('
30 /.$putCase consumeLeftParen(); $break ./
31
32 GOTPushRPAREN ::= ')'
33 /.$putCase consumeRightParen(); $break ./
34
35 ----
36 -- Querymethod Invocation
37
38 GOTQueryMethodInvocation ::= Name '(' GOTArgumentListopt ')'
39 /.$putCase consumeMethodInvocationName(); $break ./
```

Listing 5.9: Das Listing zeigt die Grammatik-Regeln für einen Query. Der Query ist mit einer eingeschränkten Java-Expression zu vergleichen. Er beinhaltet die folgenden Operatoren: `&&`, `||`, `!`. Als Primary sind `true`, `false` und der Aufruf von Querymethoden erlaubt.

Queries

Da eine Java-Expression umfangreicher als ein Query bzw. eine Queryexpression ist, ließe sich ein Query mit Hilfe der Java-Expression abbilden. Daraus würde allerdings folgen, ähnlich der Problematik bei einer Queryklasse, dass im Parser eine zusätzliche Überprüfung vorgenommen werden muss und Expression eingeschränkt werden muss. Es muss geprüft werden, ob die Expression ein gültiger Query ist und nur die zugelassenen Elemente enthält. So darf beispielsweise ein Query im Gegensatz zu einer Java-Expression keinen „?“-Operator enthalten. Um dem aus dem Weg zu gehen wurden Queries (Queryexpressions) von Grund auf neu definiert. Die für Queries notwendigen Grammatik-Regeln werden in Listing 5.9 gezeigt.

Die `match` und `declare` Sprachelemente

Die `match` und `declare` Elemente sind bis auf ihre unterschiedlichen Bezeichner syntaktisch äquivalent. Es wird daher repräsentativ nur auf das `match` Sprachelement eingegangen. Das `match` Statement soll optional an den Header eines Teams angehängt werden können. Daher muss der Header um dieses optionale Element erweitert werden. Dafür wird das `match` Element der

```
1
2  ----
3  -- match Integration
4
5  -- {Generic Object Teams: match
6  ClassHeader ::= ClassHeaderName
7    ClassHeaderExtendsopt ClassHeaderImplementsopt ClassHeaderPlayedByopt
8    Predicateopt GOTMatchOrDeclareopt
9  -- Jan Marc Hoffmann}
```

Listing 5.10: Das Listing zeigt eine Erweiterung der `TypeDeclaration` und damit eines Teams um die `match` und `declare` Sprachelemente. Verantwortlich dafür die Regel `GOTMatchOrDeclareopt`, die das eine oder andere Sprachelement zulässt.

```
1  ClassHeaderName1 ::= Modifiersopt 'class' 'Identifier'
2  /.$putCase consumeClassHeaderName1(); $break ./
3  -- {Generic Object Teams
4  ClassHeaderName1 ::= Modifiersopt 'class' 'MetaIdentifier'
5  /.$putCase consumeClassHeaderName1(); $break ./
6  ClassHeaderName1 ::= Modifiersopt 'class' 'UnBoundMetaIdentifier'
7  /.$putCase consumeClassHeaderName1(); $break ./
8  ClassHeaderName1 ::= Modifiersopt 'class' 'BoundMetaIdentifier'
9  /.$putCase consumeClassHeaderName1(); $break ./
10 -- Jan Marc Hoffmann}
```

Listing 5.11: Das Listing zeigt die Grammatikerweiterung um die Verwendung von Metavariablen in Klassennamen zu ermöglichen.

Produktion für einen `ClassHeader` hinzugefügt. Das `match` Element ähnelt von der Syntax her einer `MethodDeclaration`. Es fehlen jedoch Modifier und Rückgabewert. Den Body des `match` Konstruktes macht der in Abschnitt 5.1.2 beschriebene Querymethoden Body aus. Analog zu Queryklassen wird die Neudefinition der Verwendung und Einschränkung einer `MethodDeclaration` vorgezogen. In der Grammatik lässt sich der Einsatz des `match` Elementes nicht auf ein Team beschränken. Diese Einschränkung muss später im Parser vorgenommen werden. Listing 5.10 zeigt die Einbindung des `match` Sprachelementes in den Header eines Teams.

Metavariablen in Klassennamen

Generic Object Teams soll die Möglichkeit bieten auch für Klassennamen Metavariablen zu verwenden. Da in der OT-Grammatik Klassennamen nicht über `SimpleName` definiert werden, sondern direkt Identifier verwendet werden, muss die für die Klassennamen zuständige Regel `ClassHeaderName1` für Metanamen erweitert werden. In Listing 5.11 werden die verwendeten Regeln gezeigt. Diese Veränderung wird analog für Methodennamen und damit auch Callin- und Calloutbindungen vorgenommen.

Der `per` Block

Der `per` Block ist ein neues Konstrukt von Generic Object Teams, welches beliebige Elemente innerhalb von Klassenrümpfen und Methodenrümpfen umschließen soll. In Hinblick auf die AST-Realisierung wird hierfür auf grammatikalischer Ebene unterschieden, zwischen der Verwendung in Klassen und Methoden. Diese Unterscheidung ist notwendig, da im AST ein per-Element entweder

als Klasse oder als Block dargestellt werden soll. Wird in der Grammatik diese Unterscheidung schon getroffen, können die entsprechenden Methoden zur AST-Erstellung im Parser über die Makro-Definitionen an den Regeln, direkt aufgerufen werden.

Für die Definition von per Blöcken in der LPG Parsergenerator Grammatik muss eine andere Vorgehensweise als üblich verwendet werden. Analog zu bisherigen Umsetzungen sollte ein per-Element aus einem Header und einem Body bestehen. Den Inhalt des Bodies bilden dann alle Elemente, die ein per Block umschließen kann. Dieser Umstand führt jedoch beim Erzeugen des Parsers aus der Grammatik zu zahlreichen Reduce/Reduce und Shift/Reduce Fehlern. Die Grammatik wird als nicht LALR(1) konform eingestuft. Die Grammatik ist also nicht mehr eindeutig.

Um die Grammatik eindeutig zu gestalten, ohne eine umfassende Änderung bestehender Regeln vorzunehmen, wurde daher eine andere Variante entwickelt. Dafür wird der `per` Block als einfaches umschließendes Konstrukt gesehen (was er im Prinzip auch ist), dass nur aus einem Anfang und einem Ende besteht. Für den Anfang und das Ende werden jeweils eigene Regeln erstellt. Vor, bzw. nach einem jeden Element, welches innerhalb eines Classbodies oder Methodbodies stehen kann, wird nun die entsprechende Anfang- oder Ende-Regel gesetzt. Glücklicherweise haben alle relevanten Elemente eine gemeinsame übergeordnete Regel, die als Stellvertreter gewählt werden kann.

Listing 5.12 zeigt die Regeln für einen per-Block, der Elemente innerhalb einer Klasse umschließen soll. Die Integration dieses per-Blocks in bestehende OT-Regeln erfolgt in Zeile 44. Das Element `ClassBodyDeclaration` beschreibt alle Elemente, die eine Klassen enthalten darf. Folglich ist es sinnvoll die `per` Start- und Ende-Regeln um dieses Element zu legen.

Die Definition der Regeln für das per Element, welches innerhalb von Methodenrümpfen verwendet werden kann, läuft ähnlich. Das OT-Element, welches alle Elemente, die innerhalb von Methodenrümpfen vorkommen können, heißt `BlockStatement`. Es bietet daher die Grundlage für die Start- und Ende-Regeln. Auf eine Abbildung der zugehörigen Grammatik wird aufgrund ihrer Ähnlichkeit zur Klassen-Variante des per Blocks verzichtet.

Variablen

In Generic Object Teams sollen zusätzlich zu herkömmlichen Java Variablen-Deklarationen auch Metavariablen-Deklarationen zugelassen werden. Dafür muss die Regel, die für die Variablen-Deklaration zuständig ist, erweitert werden. Die notwendige Erweiterung der Regel `Variable DeclaratorId` zeigt Listing 5.13.

Grammatikanpassungen für diet-Parsing

Um *Diet-Parsing* (siehe Abschnitt 5.3.3) für Querymethoden zu ermöglichen, muss eine zusätzliche Regel eingeführt werden, welche direkt vom Startsymbol auf den Inhalt eines Querymethoden Body führt. Eine solche Regel, die einen alternativen Weg zum Startsymbol öffnet, heißt in der OT-Grammatik *Goal*. Goals werden verwendet, um Teil-Parsevorgänge durchzuführen. Im Fall von Diet-Parsing sollen Methodenrümpfe separat geparst werden. Um Mehrdeutigkeiten zwischen den verschiedenen Wegen zum Startsymbol zu vermeiden, ist das erste Symbol einer Goal-Regel immer

```
----
-- per-Class Definition

GOTPerClassOpen ::= GOTPerClassHeader GOTPerClassBodyStart

GOTPerClassClose ::= GOTPerClassDeclaration

GOTPerClassBodyStart ::= GOTPerClassOpenBlock

GOTPerClassBodyEnd ::= GOTPerClassCloseBlock

GOTPerClassDeclaration ::= GOTPerClassBodyEnd
/.$putCase consumeClassDeclaration(); $break ./

GOTPerClassHeader ::= GOTPerClassHeaderName GOTPerClassArgumentList GOTPerClassHeaderRightParen
/.$putCase consumeClassHeader(); $break ./

GOTPerClassHeaderName ::=  'per' GOTPerClassHeaderLeftParen
/.$putCase consumeGOTPerClassHeaderName(); $break ./

GOTPerClassHeaderLeftParen ::= '('

GOTPerClassHeaderRightParen ::= ')'
/.$putCase consumeGOTPerClassHeaderRightParen(); $break ./

GOTPerClassOpenBlock ::= '{'

GOTPerClassCloseBlock ::= '}'

GOTPerClassArgumentList ::= GOTPerClassArgument
GOTPerClassArgumentList ::= GOTPerClassArgumentList ',' GOTPerClassArgument
/.$putCase consumeArgumentList(); $break ./

GOTPerClassArgument ::= GOTName

----
-- per-Class Integration

ClassBodyDeclaration -> ConstructorDeclaration
--1.1 feature
ClassBodyDeclaration ::= Diet NestedMethod CreateInitializer Block
/.$putCase consumeClassBodyDeclaration(); $break ./
-- {Generic Object Teams
ClassBodyDeclaration ::= GOTPerClassOpen ClassBodyDeclarationsopt GOTPerClassClose
-- Jan Marc Hoffmann}
```

Listing 5.12: Das Listing zeigt die Grammatik-Regeln für den per-Block, der innerhalb von Klassenrümpfen verwendet werden kann. Im Gegensatz zu anderen Realisierungen ähnlicher Elemente wird der per-Block mit Hilfe von Start und Ende Elementen gebildet. Dies erfolgt durch die Regeln `GOTPerClassOpen` und `GOTPerClassClose`, die um alle Elemente, die innerhalb von Klassenrümpfen auftreten können, gelegt werden müssen.

```
VariableDeclaratorId ::= 'Identifier' Dimsopt
-- {Generic Object Teams
VariableDeclaratorId ::= 'MetaIdentifier' Dimsopt
VariableDeclaratorId ::= 'UnBoundMetaIdentifier' Dimsopt
VariableDeclaratorId ::= 'BoundMetaIdentifier' Dimsopt
-- Jan Marc Hoffmann}
```

Listing 5.13: Abgebildet ist die Erweiterung der Grammatikregel, die für die Variablen-Deklaration in Java zuständig ist. Die erweiterte GOT-Variante lässt neben normalen Identifiern auch Metaidentifier zu.

ein Terminal. Der Parser kann dann später das besagte Symbol für dieses Goal auf den Stack legen und die Parserlogik dahingehend beeinflussen, das Goal abzuarbeiten.

Für Querymethoden wird ein neues Goal angelegt, welches mit dem Terminal „...“ beginnt und auf die Regel zum Parsen von Querymethoden Bodies (siehe 5.1.2) führt. Die Grammatik-Regel für das Goal wird in Listing 5.14 gezeigt.

```
1 Goal ::= '...' GOTQueryMethodBlockStatementsopt
```

Listing 5.14: Das Listing zeigt ein zusätzliches Goal zum Diet-Parsing von Querymethoden. Der Parser kann dieses Goal verwenden um Methodenrümpfe zu parsen.

5.1.3 Zusätzliche Constraints

Leider lassen sich nicht alle syntaktischen und semantischen Anforderungen (Constraints) in der Grammatik abbilden. Diese, bisher nicht abgedeckten Constraints, werden im Folgenden beschrieben.

1. `match` und `declare` sind nur innerhalb von Teams erlaubt.
2. `per`-Blöcke dürfen nicht ohne umschließendes `match` oder `declare` existieren.
3. Jede Metavariable muss von einem `per`-Block umschlossen sein.
4. Metavariablen dürfen nur in Teams mit `match` oder `declare` Element verwendet werden.
5. Ein Queryimport darf nur für Queryklassen verwendet werden.

5.2 Scanner

Der Scanner, oder auch Lexer genannt ist für die Erkennung der Mikrosyntax von OT zuständig. Die Mikrosyntax beschreibt die Syntax der einzelnen Wörter der Sprache. Der Scanner liest die einzelnen Zeichen der Eingabe und bildet daraus Wörter. Kann eine Zeichenfolge nicht als Wort der Sprache erkannt werden, meldet er einen Fehler. Der OT Scanner ist von Hand geschrieben und erkennt alle Wörter der Sprachen OT und Java. Dabei wird ein Wort / Terminal in Verbindung mit einer Terminal ID gebracht. Diese ID dient dem Parser als Grundlage, um die Syntax der Wörter zueinander, die Makrosyntax der Sprache zu erkennen. Für Generic Object Teams muss der Scanner erweitert werden, um Wörter von GOT zu erkennen und Terminal IDs (`TerminalTokens`) zuzuweisen. Im Unterschied zu anderen Variationen von Scannern, liest der OT Scanner nicht eine gesamte CompilationUnit ein und erstellt daraus einen Tokenstrom, sondern wird vom Parser gesteuert und liest Token (Wort) für Token, nach Bedarf ein. Die dabei zentral verwendete Methode, ist die `getNextToken` Methode, welche das nächste Token an der aktuellen Position des Scanner zurück gibt.

Der Scanner kommt sowohl bei der Erstellung der Typhierarchie (Phase 1.1), als auch beim Resolving von Statements in Methodenrümpfen (Phase 2.1) zum Einsatz. Nähere Details zum

Compilierungsprozess zeigt Abbildung 5.1. Da der Scanner eng verzahnt mit dem Parser ist und nicht als eigene Phase betrachtet werden kann, wird er im Folgenden Abschnitt 5.3 parallel zum Parser erläutert. Dabei wird auch auf die notwendigen Adaptionen für GOT eingegangen.

5.3 Parser

Der Parser ist im Compilierungsprozess (siehe Abbildung 5.1) für die Umwandlung von Quellcode in einen AST zuständig ist. Da der Generic Object Teams Parser als Erweiterung des OT Parsers realisiert werden soll, ist es notwendig den Aufbau und die Funktionsweise des OT Parsers näher zu untersuchen.

Der Object Teams Parser ist ein LALR(1) Shift-Reduce Parser. LALR(1) bedeutet, dass der Parser die Eingabe von links nach rechts liest und Rechtsableitungen bildet. Das heißt, dass Ableitungen vom speziellen zum allgemeineren Element gebildet werden. Dabei wird immer *ein* Token im Voraus gelesen, um die nächste Regel zu bestimmen.

Shift-Reduce Parser funktionieren mit vier verschiedenen Aktionen (`Shift`, `Reduce`, `Accept`, `Error`) und einem Stack. Dabei wird der Tokenstrom von links nach rechts eingelesen, aber die Produktionen von rechts nach links angewendet. Es wird vom Speziellen zum Allgemeinen vereinfacht. Bei der `Shift` Aktion wird ein Token eingelesen und auf den Stack gelegt. `Reduce` wiederum versucht in der Menge der Produktionen eine Regel zu finden, deren rechte Seite mit den Tokens / Non-Terminals auf dem Stack übereinstimmt. Wird eine solche Regel gefunden, wird die linke Seite auf den Stack gelegt und die rechte Seite vom Stack entfernt. `Accept` beschreibt den Endzustand des Parsers. Es wurde das Startsymbol gelesen. `Error` beschreibt den Fehlerzustand des Parsers. Es wurde eine Tokensequenz gelesen, die sich nicht mit der zugrunde liegenden Grammatik bilden lässt.

Der Parsierungsprozess lässt sich mit folgendem Algorithmus beschreiben.

1. Verschiebe ein Token vom Eingabestrom auf den Stack.
2. Reduziere die obersten Elemente auf dem Stack so lange, bis die obersten Elemente auf dem Stack nicht mehr der rechten Seite einer Grammatikregel entsprechen.
3. Es sind noch weitere Token im Eingabestrom vorhanden. Fange wieder bei Schritt 1 an.
4. Es sind keine Token mehr im Eingabestrom vorhanden. Ende.

Das Grundgerüst dieses Algorithmus ist einfach. Jedoch ist die zugrunde liegende Grammatik oft sehr komplex. Die Problematik liegt in der Umsetzung dieser Grammatik in eine von der Maschine verwendbare Form. Für LALR(1) Grammatiken bietet es sich an, die Logik und somit alle Produktionsregeln in einem deterministischen, endlichen Automat zu kodieren. Dafür werden Goto- und Aktionstabellen erstellt, die die Zustandsübergänge und Stacktransformationen des Parsers beschreiben. In diesen Tabellen wird festgehalten, welcher Zustand dem Aktuellen folgt und ob eine der Aktionen `Shift`, `Reduce`, `Accept` und `Error` ausgeführt werden soll.

Aufgrund der Größe dieser Tabellen, schon bei kleineren Grammatiken, bietet es sich an, um den Aufwand und die Anzahl der Fehler möglichst gering zu halten, einen Parsergenerator zu verwenden. Solch ein Parsergenerator erzeugt aus einer LALR(1) [1] Grammatik die notwendigen Tabellen. Außerdem stellt er optional die notwendige Logik zur Verwendung der Tabellen in einer Programmiersprache zur Verfügung. Oftmals ist diese Logik für mehrere Programmiersprachen generierbar. Manche Generatoren erzeugen zusätzlich einen Scanner.

5.3.1 LPG Parser

Das OTDT verwendet den LALR Parser Generator (LPG) Parsergenerator, um die Logik des Parsers zu generieren. Im folgenden Abschnitt sollen daher die Funktionsweise und Eigenheiten des LPG Parsergenerators beschrieben werden.

Der LPG Parser, ehemals JikesPG, ist ein Parsergenerator, welcher aus einer LALR(1) Grammatik, Action- und Gototabellen erzeugt. Außerdem erstellt er für die Verwendung der Tabellen die notwendige Logik in Form von Stubs für Java.

Generierung und Generat

Nachdem man eine LALR(1) konforme Grammatik in der von LPG lesbaren Syntax erstellt hat, ruft man den Parsergenerator mit der Grammatik als Argument auf. Nach kurzer Rechenzeit gibt der Generator eine Vielzahl von Informationen im Terminal aus. Die davon wichtigsten Informationen lassen sich auf folgende Abschnitte zusammenfassen.

1. *** The following Terminals are useless:
2. *** The following Non-Terminals are useless:
3. This grammar is LALR(1).

Die beiden Meldungen 1 und 2, sind selbsterklärend. Es existieren (Non)-Terminals, die nicht vom Startsymbol erreichbar sind. Diese können aus der Grammatik entfernt werden.

Der wichtigste Abschnitt ist hierbei der letzte Satz. Sofern er besagt, dass die Grammatik LALR(1) konform ist, war die Generierung erfolgreich. Ist sie das nicht, befindet sich ein semantischer Fehler in der Grammatik. Es wird zwar dennoch ein Generat erzeugt, die spätere Verwendung kann jedoch zu unvorhersehbaren Problemen führen. Eine der häufigsten Ursachen hierfür ist das Vorhandensein von zwei Produktionen, die den gleichen Pfad zum Startsymbol besitzen. Wir erinnern uns an die Vorgehensweise des Parsers aus Abschnitt 5.3. Der Parser liest von links nach rechts, produziert aber von rechts nach links. Findet er ein Token, welches über zwei verschiedene Wege bis zum Startsymbol reduziert werden kann, ist ein undefinierter Zustand erreicht. Der generierte Parser mag oberflächlich funktionieren, aber der Entwickler wird sich wundern, warum ein bestimmter Zustand niemals erreicht wird. Zur näheren Bestimmung des Fehlers wird eine Regelnummer angegeben,

[1] LALR Parser sind auf Performance optimierte LR Parser. Es werden Zustände mit gleichen Followmengen zusammengeführt. Dadurch wird eine geringere Tabellengröße erreicht, welche in schnellerem Parsen resultiert.

die man im ausführlichen Log `grammatik.l` nachschlagen kann. Der Name *grammatik* steht hier für den Dateinamen der übergebenen Grammatik (siehe Eclipse Foundation (2011b)).

Das fertige Generat des LPG Parsergenerators besteht aus 7 Dateien.

- `grammatik.l` – Die ausführliche Logdatei der Generierung. Sie beinhaltet detaillierte Informationen über die Erzeugung der Tabellen.
- `javasym.java` – Ein Interface mit allen im Parser verwendeten Tokens.
- `javadef.java` – Ein Interface mit Parser Konstanten, wie z. B. Anzahl der Terminals, Startzustand, Errorsymbol und Endzustand.
- `JavaAction.java` – Stub für die Verbindung von generiertem und nicht-generierten Code. Hier werden die `consume`-Methoden aufgerufen, die für die Erstellung von AST-Knoten verantwortlich sind (siehe Abschnitt 5.3.1.
- `javahdr.java` – Bezeichner für die Produktionsregeln.
- `javadcl.java` – Action- und Gototabellen in Arrays kodiert.
- `javaprs.java` – Stub für die Verwendung der Action und Gototabellen des LPG Parsers.

Integration in das Eclipse JDT

Die mit dem Generator erstellten 7 Dateien müssen in das JDT (OTDT) integriert werden. Der Kern des Parsers befindet sich im `org.eclipse.jdt.internal.compiler.parser` Package. Die generierten Dateien müssen, bevor sie im OT Parser verwendet werden können, umgewandelt werden. Dafür ist die Methode `Parser.buildFilesFromLPG` zuständig, die aus der `javahdr.java` und der `javadcl.java` binäre Dateien mit den Werten der darin enthaltenen Arrays erzeugt. Diese Dateien heißen `parser[1-24].rsc` und `readableNames.properties`. Beim Start von Eclipse werden die Dateien gelesen und in entsprechende Arrays in der `Parser` Klasse geladen.

Das in der `javasym.java` enthaltene Interface mit Terminal Tokens Konstanten wird 1:1 in das Interface `TerminalTokens` übernommen. Das Interface `javadef.java` mit Parser Konstanten wird in das Interface `ParserBasicInformation` übernommen. Die in der `JavaAction.java` enthaltene `void consumeRule(int act)` Methode wird in die `Parser` Klasse eingefügt.

Funktionsweise des OT Parsers

Die Funktionsweise des OT Parser soll im folgenden Abschnitt dargestellt werden. Dabei wird auf die wichtigsten Details eingegangen, die für die Entwicklung des GOTDT relevant sind.

Der Kern des Parsers befindet sich in der `org.eclipse.jdt.internal.compiler.parser.Parser` Klasse. Die darin befindliche `void parse()` Methode ist die Hauptschleife des Parsers. Der in der Methode enthaltene Algorithmus, lässt sich wie folgt, grob beschreiben:

1. Lese ein Token vom Scanner. (`int scanner.getNextToken()`)

2. Schreibe das Token auf den Stack.
3. Prüfe ob die oberen Elemente des Stacks reduziert werden können. Wenn ja, reduziere und rufe die `void consumeRule(int act)` Methode auf. Wenn nein, gehe zu Schritt 1.
4. Wenn ein Fehler aufgetreten ist, starte den Diagnose Parser. Wenn der Ende-Zustand erreicht wurde, beende.

Um folgende Designentscheidungen nachvollziehen zu können, muss hier allerdings noch etwas tiefer ins Detail vorgegangen werden.

Da der LPG Parsergenerator keinen Scanner erzeugt, ist die Klasse `Scanner` von Hand geschrieben worden. Der Scanner liest den Eingabestrom und erzeugt aus gelesenen Wörtern Token. Die Id eines erkannten Tokens wird an den Parser zurück geliefert. Neben der Id, wird auch die aktuelle Position und die Startposition des aktuellen Tokens gespeichert, damit der Parser Position und Wert des aktuellen Tokens auslesen kann. Im Beispiel eines Identifier kann der Parser mit Hilfe von dieser Position den Wert des Identifier auf einen speziellen Identifierstack schreiben. Der Scanner liest immer ein Token auf Anfrage des Parsers ein. Alle verfügbaren Token und deren Ids sind in dem vom Parser und Scanner implementierten Interface `TerminalTokens` definiert.

Die Überprüfung in Schritt 3, ob die aktuellen Elemente auf dem Stack reduziert werden können, erfolgt mit Hilfe von zwei statischen Methoden `static int tAction(int state, int sym)` und `static int ntAction(int state, int sym)`. Hierbei wird die Id des aktuell eingelesenen Terminals (Tokens), oder im zweiten Fall die Id des auf dem Stack befindlichen Non-Terminals, zusammen mit dem aktuellen Zustand an die entsprechende Methode übergeben. Das Ergebnis ist der neue Zustand. Abhängig von diesem Zustand kann die Prozedur dann wiederholt, oder in den Error oder Accept Zustand übergegangen werden.

Die Entscheidung, ob die eingelesene Zeichenkette mit der Grammatik abbildbar ist, wird vom generierten Parser Automaten übernommen. Der Kern dieses Automaten sind große Tabellen mit Zustandsübergängen und Token Ids. Als Beispiel wird das Token mit der Id 10 eingelesen und zusammen mit dem aktuellen Zustand 100 an die `tAction` Methode übergeben. Als Ergebnis liefert diese Methode dann je nach Grammatik den neuen Zustand 101, oder 10000 für Accept, oder 20000 für Error.

Dank der Verwendung eines solchen Automaten, der mit einfachen Operationen auf Integer Werten arbeiten kann, sind Parsing Entscheidungen sehr performant zu treffen. Der Nachteil dieses Systems ist allerdings die nahezu unmögliche Manipulation des Automaten. Denn nur der Parsergenerator kennt die Verknüpfung von Zuständen, Terminals und Produktionen der Grammatik. Veränderungen der Logik oder Erkennung von speziellen Zuständen, ohne eine neue Generierung der Logik Tabellen kann also bestenfalls mit empirischen Versuchen anhand von bekannten syntaktischen Mustern vorgenommen werden. Dieser Nachteil der Veränderung prägt, wie in Abschnitt 5.3.2 beschrieben, die Entscheidung über die gewählte Technik der Adaption des OTDT Parsers für GOT.

Sofern die Logik des Parsers entschieden hat, dass eine ReduceAktion ausgeführt werden soll, wird die Methode `consumeRule(int act)` aufgerufen. In dieser Methode befindet sich ein an die tausend Fälle großes Switch Statement, welches die Nummern der Produktionsregeln, die intern verwendet werden, auf Methoden abbildet. Diese Funktion wurde aus der Grammatik generiert.

Jede Regel wird einer Methode mit dem Präfix `consume` zugeordnet. Die `consume` Methoden sind von Hand implementiert. Die `consumeRule(int act)` Methode bildet also die Schnittstelle zwischen generiertem und nicht generiertem Parser. In den `consume` Methoden wird der AST aufgebaut.

Exemplarisch soll in folgendem Beispiel erläutert werden, wie die unten stehende Methode durch den Parser geparst wird. Der Ablauf der Parsierung ist leicht vereinfacht. Es werden nicht alle Reduce Operationen aufgelistet.

```
1 public void method() {
2 }
```

1. Shift `public` auf den Stack
2. Shift `void` auf den Stack
3. Shift `Identifier` auf den Stack
4. Shift `'('` auf den Stack
5. Shift `')'` auf den Stack
6. Reduce `consumeMethodHeaderName`
7. Reduce `consumeFormalParameterListopt`
8. Shift `'{'`
9. Reduce `consumeMethodHeaderRightParen`
10. Reduce `consumeMethodHeader`
11. Shift `'}'`
12. Reduce `consumeBlockStatementsopt`
13. Reduce `consumeMethodBody`
14. Reduce `consumeMethodDeclaration`

An diesem Beispiel kann man gut erkennen, wie der Parser den AST vom speziellen ins allgemeine Element aufbaut.

Besonderheiten des OT Parsers

Eine Besonderheit des OT Parser ist seine Fähigkeit, neben kompletten Quelldateien auch einzelne Bereiche zu parsen. Mit Hilfe von dieser Möglichkeit ermöglicht der Parser eine Parsierung einer kompletten CompilationUnit ohne, die darin befindlichen Methodenrümpfe zu parsen. In einem späteren Schritt werden dann, bei Bedarf, die Methodenrümpfe geparst. Diese spezielle Behandlung von Methodenrümpfen wird im OT Parser *diet-Parsing* genannt. Ziel von diet-Parsing ist eine schnellere Parsierung, da ein Großteil einer Java Datei aus Statements besteht, die damit übersprungen werden können. Der aus dem diet-Parsing entstehende AST ohne Statements kann

z. B. für die Outline verwendet werden, die nicht auf Informationen innerhalb von Methodenrümpfen angewiesen ist. Neben dem separaten Parsen von Methodenrümpfen sind noch weitere Teil-Parsierungsvorgänge möglich. Die Grundlage für diese Funktionalität muss in der Grammatik des Parsers in Form von Goals geschaffen werden (siehe Abschnitt 5.1.2).

5.3.2 Adaption für Generic Object Teams

Die Adaption des OT Parsers kann über verschiedene Varianten vollzogen werden. Ein Ziel dieser Arbeit ist es die Adaption, wenn möglich, mit *OT-Equinox* durchzuführen. Im Folgenden werden die vier möglichen Varianten den OT/J Parser für Generic Object Teams zu erweitern, erläutert und gegeneinander abgewogen.

Zum besseren Verständnis der folgenden Varianten soll hier nochmal auf die Funktionsweise des OT/J Parsers hingewiesen werden. Die Logik des Parsers wird durch einen Parsergenerator erzeugt. Sie ist als Automat realisiert und arbeitet mit numerischen Zuständen und Ids. Die generierten Zustandstabellen müssen beim Start des Parsers aus Resourcendateien in Arrays eingelesen werden. Token und spezielle Zustände werden als *finale*, statische, numerische Konstanten in den beiden Interfaces `TerminalTokens` und `BasicParserInformation` generiert. Bei der Veränderung der Grammatik mit anschließender Re-Generierung, ändern sich die Ids dieser Konstanten (siehe Abschnitt 5.3.1).

Variante 1: Adaption durch Manipulation der Kommunikation zwischen Scanner und Parser

In dieser Variante wird der Umstand verwendet, dass alle Informationen, die der Parser über den Quelltext bekommt, zuerst durch den Scanner gehen. Die Idee ist daher den Scanner zu adaptieren, um neue, bisher unbekannte Elemente einzulesen. Da der Parser diese Elemente allerdings nicht kennt und mit ihnen nichts anfangen kann, sofern sie nicht in der OT Grammatik vorhanden sind, muss man sich einem Trick behelfen. Dieser Trick besteht darin die neuen, durch den Scanner gelesenen Elemente, in bestehenden Tokens zu verpacken. Die so verpackten Elemente werden dann im Parser entpackt und gesondert behandelt. Die vom Scanner gesammelten Informationen kann der Parser dann verwenden, um den AST entsprechend anzupassen.

Die Adaption des Scanners kann mit der in Listing 5.15 gezeigten Rolle erreicht werden. Dabei wird von der aktuellen Position des Scanners an geprüft, ob die folgende Zeichenkette ein für GOT interessantes Token ist. Wenn dies der Fall ist, wird der Parseradapter aktiviert (siehe 5.16) und die durch den Scanner festgestellten Informationen wie Position und Wert übergeben. Dem OT Parser wird danach ein leeres Token in Form des `TokenNamenull` Tokens weitergeleitet. Dieses Token ist für den Parser im Body einer Klasse oder Methode ein neutrales Element, bei dem der AST nicht verändert wird. Der Parseradapter wartet auf das `TokenNamenull` Token und manipuliert mit Hilfe der dem Adapter übergebenen Informationen den AST. Eine detailliertere Auseinandersetzung mit dieser Art von Parseradaption bietet das Object Teams Blog von Stephan Herrmann (siehe Stephan Herrmann (2010a) und Stephan Herrmann (2010b)).

```
public team class ScannerAdaptor {
  protected class GOTScanner playedBy Scanner {
      // intercept execution of getNextToken
      int getNextToken() <- replace int getNextToken();

      callin int getNextToken() throws InvalidInputException {
        // activate ParserAdaptor and feed it with the informations we got
        new ParserAdaptor(token, start-2, end+1).activate();
        // pretend we read nothing important
        return TerminalTokens.TokenNamenull;
      }
  }
}
```

Listing 5.15: Das Listing zeigt ein Team für die Adaption des OT Scanners für GOT. Hierbei wird mit Hilfe eines Callins jeglicher Aufruf der Methode `getNextToken` abgefangen. Das aktuelle Token wird in der Callin-Methode gelesen. Daraufhin wird ein weiterer Aspekt aktiviert, der als Argumente Positionen und Wert des aktuellen Tokens übergeben bekommt. Dieser ParserAdaptor Aspekt hat zur Aufgabe die ihm übergebenen Informationen auszuwerten und den AST entsprechend anzupassen. Der Rückgabewert des Callins wird das neutrale `TokenNamenull`, um dem aufrufenden Parser vorzugaukeln, dass das aktuelle Token nicht von Bedeutung ist.

```
public team class ParserAdaptor {
  public ParserAdaptor(int token, int start, int end) {}

    protected class GOTParser playedBy Parser {
        // intercept execution of consumeToken
        void consumeToken(int type) <- replace void consumeToken(int type);

        callin void consumeToken(int type) {
            if (type == TerminalTokens.TokenNamenull) {
                // adjust \ac{AST}
                return;
            }
            base.consumeToken(type);
        }
    }
}
```

Listing 5.16: Das Listing zeigt ein Team, welches den Parser adaptiert und vom Scanner aufgerufen wird, sofern im Scanner ein interessantes Sprachelement gelesen wurde. Dieses Team hat zur Aufgabe den AST entsprechend der vom Scanner Team gesammelten Informationen anzupassen. Hierfür wird mit Hilfe eine Callins die Ausführung der `consumeToken` Methode abgefangen und im Falle eines gelesenen `Tokennamenull` Tokens, die Information aus dem Scanner Team verwendet, um den AST anzupassen.

Für bestimmte Adaptionen im kleineren Rahmen ist diese Variante sinnvoll. Unter diese Adaptionen fällt das Hinzufügen von eingebetteten Sprachen, bei denen ein Start und ein Ende erkannt werden kann. Diese Sprachen können dann geparst und in OT/J AST kompatible Elemente umgewandelt und dem AST hinzugefügt werden. Außerdem ist natürlich das Umbenennen von Tokens möglich. Ein Beispiel für eine Umbenennung wäre es natürliche Zahlen bis 10 auch in textueller Form zuzulassen (eins, zwei, drei, ...).

Wird die Komplexität der eingebetteten Sprache zu groß, bietet sich die Verwendung eines zusätzlichen Parsergenerators an, um die Makrosyntax der Eingabe zu prüfen (siehe 5.3.2).

Für die Realisierung des GOT Parsers ist diese Variante nur bedingt sinnvoll. Das Erweitern der Grammatik des Parsers um zusätzliche Tokens lässt sich zwar gut umsetzen, aber Verändern und Ersetzen von bestehenden Elementen der Grammatik ist nur zufriedenstellend möglich, sofern man alle Vorkommen unabhängig von deren syntaktischer Position in der Grammatik ersetzen will. Denn sobald man ein Element nur an bestimmten Stellen ersetzen möchte, benötigt man einen Einblick in die Logik des Parsers. Da die Makrosyntax Überprüfung des Parsers wie in Abschnitt 5.3.1 beschrieben, durch den Parser Automaten vollzogen wird, gestaltet sich eine Manipulation der Logik als schwierig.

Variante 2: Adaption mit Hilfe eines zusätzlichen Parsergenerators

In dieser Variante wird ein zusätzlicher Parser verwendet, um die Syntax von Generic Object Teams Sprachelementen zu erkennen. Hierbei wird versucht den bestehenden OT Parser zwar beizubehalten, aber beim Auftreten von GOT Elementen einen zweiten, externen Parser einzuschalten.

Um dies zu ermöglichen, wird ein `ParserAdapter` Team erstellt, dass die `Parser.parse()` Methode des OT Parsers mit Hilfe eines replace Callins adaptiert. Da der bestehende Parser nur erweitert werden soll und nicht komplett ersetzt werden soll, wird der Quellcode der Basismethode in die Callin Methode kopiert. Innerhalb dieser Kopie werden dann alle Stellen angepasst an denen ein Token vom Scanner eingelesen wird. Sofern es sich bei einem gelesenen Token um ein GOT spezifisches Token handelt, wird davon ausgegangen, dass die folgenden Tokens zu einem GOT Sprachelement gehören. Der GOT Parser wird ab diesem Punkt aktiviert. Er parsiert den folgenden Quelltext, sofern er für ihn erkennbare GOT Elemente enthält. Diese Elemente werden dann in den bestehenden OT AST und den GOT AST eingepflegt. Nach dem Einsatz des GOT Parsers wird der OT Parser wie gewohnt bis zum nächsten Einsatz des GOT Parsers ausgeführt.

Bei dem externen Parser handelt es sich um einen, mit Hilfe von ANTLR, generierten Parser. ANTLR ist ein Parsergenerator, der mit einem ausgereiften Tooling ausgestattet ist. Dieses ermöglicht das Generieren eines Scanners, Parsers und das Debuggen und Visualisieren eines ASTs. Der mit Hilfe von ANTLR erzeugte AST muss am Ende des Parse-Vorgangs in einen OTDT kompatiblen AST umgewandelt werden. Dies erfolgt in dem der komplette AST abgelaufen wird und für jeden ANTLR Knoten ein passender OTDT Knoten erzeugt wird.

Der Vorzug dieser Variante ist der Zugriff auf das Tooling des ANTLR Parsergenerators. Somit können leicht Scanner, Parser und AST für GOT Elemente erzeugt werden. Außerdem kann mit Hilfe des Tooling, schnell und effektiv eine Grammatik für GOT Sprachelemente erstellt werden.

Nachteil dieser Variante ist die ungenügende Kenntnis über den aktuellen Zustand des OT Parsers. Um den Zusatz-Parser zu aktivieren, muss sichergestellt sein, dass sich im nachfolgenden Sourcecode ein GOT Sprachelement befindet. Weitergehend muss geprüft werden, ob dieses Element im aktuellen Kontext verwendet werden darf. Diese Überprüfungen sind jedoch nahezu unmöglich durchzuführen, da der OT Parser bekanntlich alle Zustände in numerische Ids kodiert, die sich bei jeder kleinen Veränderung der Grammatik grundlegend verändern. Somit kann keine genaue, zuverlässige Makrosyntax Überprüfung für die Einbettung der GOT Elemente verwendet werden.

Ein weiteres Problem ist die Aufteilung der Grammatik Definition in mehrere Dateien und Formate. Die gesamte GOT Grammatik würde aus einer OT Grammatik im LPG Format mit einer zusätzlichen GOT Erweiterung im ANTLR Format bestehen. Da die GOT Grammatik stark verkettet mit der OT Grammatik ist, würde diese Trennung die Lesbarkeit und das Verständnis der Grammatik einschränken.

Variante 2 entspricht einer erweiterten Variante 1. Der Hauptunterschied besteht in der Parser gestützten Erkennung von eingebetteten Sprachelementen. Ähnlich wie Variante 1 leidet sie unter den gleichen Unzulänglichkeiten. Sie ist eher für eingebettete Sprachen geeignet, die durch ein Anfang- und Ende-Token ausgezeichnet werden können.

Variante 3: Adaption durch Erweiterung der LPG Grammatik

Die in den vorherigen Varianten ausstehenden Problemstellungen sollen in dieser Variante gelöst werden. Die OT Grammatik des LPG Parsers soll direkt um Generic Object Teams Elemente erweitert werden. Das Ergebnis wäre eine übersichtliche, an einer Stelle definierte, GOT Grammatik.

Dafür wird die OT Grammatik um die in Kapitel 5.1 beschriebenen Ergänzungen erweitert. Diese Veränderungen finden in der OT Grammatik Definition des OT Parsers statt. Mit Hilfe der veränderten OT Parser Grammatik wird ein neuer Parser generiert. Die daraus entstehenden Resourcen-Dateien müssen im nächsten Schritt in den OT Parser integriert werden. Mit Hilfe von OT Equinox wird dabei ein Adapter erstellt, der die für GOT veränderten Resourcen Dateien einliest und beim Parsen einer GOT/J Datei die Goto- und Actiontabellen des original OT/J Parsers durch die neu generierten des GOT/J Parsers ersetzt.

Zu Problemen kommt es erst im nächsten Schritt. Es müssen neben den beiden Tabellen auch die beiden Interfaces mit statischen Konstanten ausgetauscht werden. Also `TerminalTokens` und `BasicParserInformation`. Da diese Interfaces statische, final Konstanten beinhalten, die vom Java Compiler *Inline* kompiliert werden, ist eine Adaption durch Object Teams oder Aspect/J unmöglich.

Wenn der Zugriff auf ein Feld oder eine Methode nicht durch einen Aspekt abgefangen werden kann, bleibt nur eine Möglichkeit. Es müssen alle Methoden, die die besagten Felder verwenden, adaptiert und die Zugriffe darin auf andere Felder umgeleitet werden. In Object Teams bedeutet dies alle Klassen, deren Methoden die besagten Konstanten verwenden, mit einer Rolle zu adaptieren. Alle in den Klassen befindlichen Methoden, die die Konstanten verwenden, müssen mit einem replace Callin kopiert und verändert werden.

Im Falle des `TerminalTokens` Interfaces lässt sich dieses Vorgehen umgehen. Der Parsergenerator hat zwar die Ids der einzelnen Konstanten geändert, allerdings nicht ihre Namen. Wichtig ist für die meisten Aufrufer lediglich das die gleiche Token Konstante immer die gleiche Id hat. Welche Id ist in der Regel egal. Es sind nur solche Stellen problematisch, an denen das Token nicht über die Konstante im Interface, sondern direkt über den Wert / Id verwendet wird. Die einzigen Stellen die direkt die Werte der Konstanten verwenden sind die beiden Methoden `Parser.tAction` und `Parser.ntAction`. Denn nur dort kommen alte TerminalToken Ids mit neuen Actiontabellen zusammen. Denn Actiontabellen beinhalten in ihren Zustandsübergängen die Werte der TokenIds. Ein Mapping an dieser Stelle von alten zu neuen Ids würde die Problematik der im Interface `TerminalTokens` definierten Konstanten lösen.

Leider ist dieses Verfahren nicht auf die Konstanten in dem Interface `BasicParserInformation` anwendbar. Für diese Konstanten muss eine Ersetzung von mehr als 20 Methoden vorgenommen werden, welche mehr als 2000 Zeilen kopierten Code ausmachen.

Diese Quellcode Kopien sind der klare Nachteil dieser Variante.

Variante 4: Ersetzung des Core Plugins

Alternativ zu der reinen OT Equinox Implementierung in Variante 3 ist es möglich das gesamte `org.eclipse.jdt.core` Plugin zu kopieren, um die problematischen Interfaces `TerminalTokens` und `BasicParserInformation` austauschen zu können. Außerdem können dann die Resourcendateien direkt ersetzt werden. An allen anderen Stellen wird weiterhin OT Equinox verwendet, um die Logik des GOT Parsers in einem extra Plugin zu bündeln. Bei dieser Variante spart man sich die Adaption der 20+ Methoden, die die alten Ids des `BasicParserInformation` Interfaces verwenden.

Der Austausch eines Eclipse Plugins auf diese Weise ist jedoch unüblich und mit weiteren Einschränkungen verbunden. So ist der Entwicklungszyklus des GOT `org.eclipse.jdt.core` Plugins im GOTDT stets an die zugehörige OT Variante gebunden. Eine neue Version des OT `org.eclipse.jdt.core` Plugins erfordert eine neue GOT Version des Plugins. Außerdem können die üblichen Plugin Lademechanismen (Dropins, Updatesite) nur mit Hilfe von zusätzlichen Hacks verwendet werden. Zusätzlich muss der adaptierte Parser zu jeder Zeit aktiv sein, da der Basisparser keine Kenntnis von den neuen Token Ids hat.

Varianten 1-4: Auswertung

Alle vorgestellten Varianten haben ihr Einsatzgebiet. So ist Variante 1 sehr gut zu verwenden für einfache, eingebettete Sprachen und Variante 2 für komplexe eingebettete Sprachen. Für Generic Object Teams allerdings eignen sich die letzten beiden Varianten besser. Ideal, da das `org.eclipse.jdt.core` Plugin nicht verändert werden muss, scheint Variante 3. Man muss sich allerdings Fragen ob der Vorteil, dass man mit Hilfe von OT/Equinox das Kopieren des Plugins sparen kann, wirklich ein Vorteil ist, wenn dafür der Code von Dutzenden Methoden kopiert werden muss. In Variante 4 wird zwar das ganze Plugin kopiert und es werden einige Dateien komplett

ersetzt, aber bei dem kopierten Code handelt es sich um Code, der automatisch vom Parsergenerator generiert wurde. Er enthält nur Konstanten und keine Methode. Es ist sehr unwahrscheinlich, dass sich an diesem Code etwas grundlegendes ändert, was nicht Ergebnis einer Generierung des Parsers ist.

Grade wenn man die Evolutionsfähigkeit der beiden Varianten 3 und 4 miteinander vergleicht, fällt auf, dass durch das Kopieren des `org.eclipse.jdt.core` Plugins in Variante 4, weniger Logik gewartet werden muss als in Variante 3. Da nicht die Methoden, die die Konstanten in dem Interfaces `BasicParserInformation` verwenden, angepasst werden, sondern die Konstanten selbst.

Aus diesen Gründen wird für die Adaption des OT/J Parsers die Variante 4 verwendet. Hier sei nochmals erwähnt, dass alle anderen für GOT relevanten Änderungen mit Hilfe von Aspekten realisiert werden.

5.3.3 Generic Object Teams Parser

Im Folgenden soll die für GOT verwendete Variante 4 der Adaption des OT Parsers näher erläutert werden.

Der erste Schritt der Adaption besteht darin, eine Kopie der OT Version des `org.eclipse.jdt.core` Plugins anzulegen. Neben der Ersetzung der Konstanten Interfaces `TerminalTokens`, `BasicParserInformation` und den Resourcendateien wird darauf Wert gelegt, keine weiteren Änderung im `org.eclipse.jdt.core` Plugin vorzunehmen. Es soll möglich sein mit Hilfe eines einfachen Helper Programms die Umwandlung der OT Version des `org.eclipse.jdt.core` in die GOT Version zu automatisieren.

Automatisierte Generierung

Um Änderungen an der Grammatik schnellstmöglich in den GOT/J Parser zu integrieren, wurde ein kleines Java Programm geschrieben, welches die Generierung des Parsers aus der Grammatik automatisiert. Der erste Schritt der Generierung ist der Aufruf des LPG Parsers. Das daraus entstehende Generat wird dann in die Resourcendateien `parser[1-24].rsc` und `readableNames.properties` mit Hilfe der `Parser.buildFilesFromLPG` Methode umgewandelt. Der nächste Schritt ist das Ersetzen der vorhanden Resourcen im `org.eclipse.jdt.core` Plugin durch die neu Generierten. Danach müssen noch die generierten Konstanten in die Interfaces `TerminalTokens` und `BasicParserInformation` geschrieben werden. Dafür wurden vorab Start- und Endmarker in den original Interfaces hinzugefügt. Anhand dieser Marker kann dann eine Ersetzung der Konstanten erfolgen. Vorteil der Marker ist, dass folgende Generierungen vollständig automatisch ablaufen können, da die Marker einmalig gesetzt werden müssen. Der letzte Schritt ist das Kopieren der generierten `consumeRule` Methode in den mit OT/Equinox realisierten Parser Adapter.

Adaption des Scanners

Der Scanner wird mit Hilfe des Teams `GOTScannerAdaptor` adaptiert. Er hat zur Aufgabe das Verhalten des OT Scanners zum Scannen von GOT Elementen anzupassen. Das Team `GOTScannerAdaptor` enthält den `GOTScanner`. Der `GOTScanner` ist eine Rolle auf den Scanner im `org.eclipse.jdt.core` Plugin.

Für die Erkennung von Tokens im Zeichenstrom ist die Scannermethode `getNextToken` zuständig. Sie sucht den Zeichenstrom von der aktuellen Position das nächste Token und gibt es an den Aufrufer zurück. Um neben OT Elementen auch GOT Elemente verarbeiten zu können, wird die Methode `getNextToken` mit Hilfe eines replace Callins adaptiert. Die Callinmethode sorgt dafür, dass zusätzlich zu allen bekannten OT Tokens auch die im folgenden aufgelisteten GOT Token im Quelltext erkannt werden.

- `TokenNamematch`
- `TokenNamedeclare`
- `TokenNameotquery`
- `TokenNameper`
- `TokenNameMetaIdentifier`
- `TokenNameUnBoundMetaIdentifier`
- `TokenNameBoundMetaIdentifier`

Besonderheiten ergeben beim Erkennen der MetaIdentfier Token. Da die für die MetaIdentifier einleitenden Zeichen '?', '-', '+' auch in anderen Kontexten vorkommen können, müssen erweiterte Erkennungen durchgeführt werden, um einer fehlerhaften Token Erzeugung vorzubeugen.

Außerdem gilt es die `scanIdentifier` Methode des Scanners zu adaptieren. Diese Methode erlaubt die Erkennung eines einzelnen Identifiers ab der aktuellen Position des Scanners im Quelltext. Die Methode muss in ihrer Funktionalität erweitert werden, um auch Metaidentifier zu erkennen. Dafür wird ein replace Callin verwendet.

Nach der Adaption der Methode kam es im Tooling zu einem Ausfall der `SelectionEngine`, welche für die Referenzierung des vom Nutzer markierten Quelltextes mit AST Knoten verantwortlich ist. Dadurch konnten alle Features, die auf einem markierten Text angewendet werden, wie z. B. „Open Declaration“ und „Open TypeHierarchie“ nicht mehr verwendet werden. Ursache dafür war der zusätzliche Aufruf der Scannermethode `getCurrentIdentifierSource` im Adapter. Der Grund des Fehlverhaltens der `SelectionEngine` beim mehrfachen Aufruf der `getCurrentIdentifierSource` Methode konnte erst in einer vom Scanner abgeleiteten Klasse gefunden werden. Dort führte der mehrfache Aufruf der Methode zu einer wiederholten Initialisierung eines Suchstrings. Im Normalfall würde dies kein Problem bereiten, jedoch werden aus Effizienzgründen im OT Compiler an vielen Stellen Strings (char-Arrays) direkt per Referenz verglichen. Und dabei führt folglicherweise eine erneute Initialisierung eines Strings zu einer neuen Referenz und damit falschen Ergebnissen im Vergleich.

Adaption des Parsers

Die Adaption des Parsers erfolgt mit Hilfe des `GOTParserAdaptor` Teams. Der adaptierte Parser verwendet die durch den Parsergenerator neu erzeugte Logik zur Überprüfung der Makrosyntax der Sprache GOT. Dabei werden die neu hinzugekommenen Token des GOT Scanners verwendet. Im Folgenden werden die wichtigsten Veränderungen zum OT Parser erläutert.

Verbindung der generierten Parserlogik mit händisch erstelltem Code

Wie im Abschnitt 5.3.2 in den Varianten 3 und 4 beschrieben, wurde die Logik des OT Parsers über eine spezielle GOT Grammatik erweitert und neu erzeugt. Daraus folgt auch, dass die Methode `consumeRule(int rule)`, die das Mapping zwischen Produktionen und `consume` Methoden enthält, neu generiert wurde. Die `consumeRule` Methode des OT Parsers wird damit ungültig, da die Ids der Produktionen in der Methode nicht mehr zu der neu generierten Logik des Parsers passen. Die consumeRule Methode muss also durch eine neue Version ersetzt werden. Dabei sind verschiedene Arten der Ersetzung denkbar. Dafür sei vorerst ein Ausschnitt aus der Methode in Listing 5.17 betrachtet. Die Methode besteht aus einem komplexen Switch Statement, welches die durch die Parserlogik verwendeten Ids von Produktionen in Methodenaufrufe zur Erzeugung von AST Knoten übersetzt.

Die Ersetzung der bestehenden `consumeRule` Methode erfolgt entweder über eine Textersetzung im OT Parsercode des `org.eclipse.jdt.core` Plugins, oder über ein replace Callin im `GOTParser` `org.objectteams.gotdt` Plugin. Beide Varianten haben ihre Vorzüge. Wird die Methode direkt im `org.eclipse.jdt.core` Plugin ausgetauscht, müssen die Aufrufe von `consume` Methoden, die für die Basis des GOT Parsers gedacht sind, nicht über den Umweg von Callouts weitergeleitet werden. Diese Einsparung spart, bei einigen hundert Methoden, ein gutes Stück Rechenzeit. Außerdem müsste zur Compilierung von Java und OT Quellcode der GOT Parser nicht aktiviert werden. Ein Nachteil wäre jedoch der Anteil an `consume` Methoden, der durch den Parsergenerator in die `consumeRule` Methode generiert wurde, der spezifisch für GOT ist. Diese Methoden müssten als leere Stubs auf ihre Adaptierung durch den GOT Parser warten. Die Methoden könnten durch die Verwendung einer parametrisierten Auswahl Methode auf eine Einzige reduziert werden.

Eine Adaption mit einem replace Callin würde den OT Parser unverändert lassen und eine leichtere Wartung und Evolution ermöglichen, da keine Änderung in der Parser Klasse vorgenommen werden müssen.

Beide Varianten haben ihre Vorzüge. Im Prototyp wurde der Einsatz eines replace Callins gewählt, um die Evolutionsfähigkeit möglichst hoch zu gestalten. Mit erhöhtem Reifegrad des Prototyps, bei dem der Fokus mehr auf die Performanz gelegt wird, sollte die `consumeRule` Methode direkt in den Parser eingebaut werden. Dabei könnten die in Abschnitt 5.3.3 erläuterten Automatisierungen zur Anwendung kommen.

Da der OT Parser ohne die auf die Parser Logik angepasste `consumeRule` Methode nicht einsatzfähig ist, muss die `GOTParser` Rolle und damit ihr umschließendes Team `GOTParser Adaptor`, zu jeder Zeit aktiv sein. Sowohl für Java als auch OT und GOT Code.

```
1  public team class GOTParserAdaptor {
2      protected class GOTParser playedBy Parser {
3        protected void consumeRule(int act) {
4          // some OT Parser methods
5          case 56 : if (DEBUG) { System.out.println("ClassOrInterface ::= Name
               "); } //$NON-NLS-1$
6            consumeClassOrInterfaceName();
7          break;
8
9          case 854 : if (DEBUG) { System.out.println("RecoveryMethodHeader ::=
                RecoveryMethodHeaderName..."); } //$NON-NLS-1$
10           consumeMethodHeader();
11         break;
12
13         // some GOT-Parser methods
14         case 966 : if (DEBUG) { System.out.println("GOTMatchDeclaration ::=
               GOTMatchHeader GOTMatchBody"); } //$NON-NLS-1$
15           consumeGOTMatchDeclaration();
16         break;
17
18       case 1003 : if (DEBUG) { System.out.println("
             GOTPerMethodBlockHeaderName ::= per..."); } //$NON-NLS-1$
19           consumeGOTPerMethodBlockHeaderName();
20         break;
21       }
22     }
23 }
```

Listing 5.17: Ein Ausschnitt der `consumeRule` Methode. Sie ist die Verbindung zwischen generiertem und händisch erzeugtem Parsercode. Sie bietet ein Mapping zwischen in der Parserlogik verwendeten Ids von Produktionen und für die Erzeugung von AST Knoten zuständigen, `consume` Methoden. Die GOT Version dieser Methode enthält sowohl Regeln für Java, OT als auch für GOT.

Einen großen Teil des Parsers machen die für die Erzeugung von AST Knoten verantwortlichen `consume` Methoden aus. Im Regelfall sind diese Methoden sehr einfach gestrickt und erstellen die entsprechenden Java / OT Knoten im OT AST. Oft können die bestehenden Methoden für GOT Elemente wiederverwendet werden. Zusätzlich müssen im GOT Parser einige `consume` Methoden eingeführt werden, die für GOT Elemente, GOT Knoten im GOT AST erstellen.

Parsierung der `match` und `declare` Sprachelemente

Die `match` und `declare` Sprachelemente, können wie in Abschnitt 2.2.2 erläutert, leicht mit Hilfe von Java Methoden dargestellt werden. Um die Elemente auch als solches erkennen zu können, werden für sie im GOT AST entsprechende Knoten erstellt. Um einen leichteren Zugriff zu ermöglichen werden sie im zugehörigen `GOTTeamDeclaration` GOT AST Knoten referenziert.

Um die Auflösung der Namen in den Elementen zu erleichtern, wird sich eines Tricks bedient. Die Elemente werden als reguläre Methoden direkt in das zugehörige Team kopiert. Da sie nun ein durch den OT Compiler erfassbarer Teil des Teams sind, werden sie als reguläre Methoden aufgelöst und allen in ihnen enthaltenen Namen werden Bindings zugeordnet. Um auch die deklarierten Metavariablen auflösen zu lassen und im ganzen Team bekannt zu machen, werden diese in Felder

umgewandelt und ebenfalls dem Team hinzugefügt. Durch diese Maßnahme verhalten sich die im `match` oder `declare` deklarierten Metavariablen wie reguläre Felder. Sie sind innerhalb des Teams verfügbar und werden im OT Compiler auflösbar. Etwaige Typfehler werden automatisch vom OT Compiler erkannt und gemeldet.

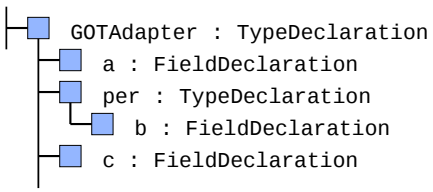
Der `per` Block

Der `per` Block besteht im AST, je nach Kontext, aus zwei verschiedenen Elementen. Innerhalb von Klassen wird der durch eine Klasse und innerhalb von Methoden durch einen Block dargestellt. Dieser Unterschied muss im Parser berücksichtigt werden. Im Folgenden wird zur Vereinfachung jedoch nur von dem `per` Element als Klasse ausgegangen, da die Vorgehensweisen für beide Varianten ähnlich sind.

Ähnlich dem `match` und `declare` Element, können auch beim `per` Element einige Transformationen angewendet werden, um die spätere Verarbeitung im Compiler zu erleichtern. In Listing 5.2a ist eine einfache Klasse mit einem `per` Element dargestellt. Abbildung 5.2b zeigt die dargestellte Klasse direkt transformiert in einen OT AST. Beim weiteren Durchlaufen des Compilers wird der hier generierte AST zu einem Typfehler führen. Denn die Variable `b` wurde innerhalb der Klasse `per` deklariert. Sie befindet sich in einem von der Variable `c`, die in der Klasse `GOTAdapter` deklariert wurde, unerreichbarem Namensraum. Dieser Umstand führt zu einem Fehler bei der Initialisierung von Variable `c`. Wenn das per Element mit Hilfe einer Klasse oder Blocks realisiert wird, muss ebenso die Namensauflösung dementsprechend angepasst werden. Sie muss den Namensraum des per Elements ignorieren. Da diese Veränderung relativ aufwendig ist, wird sich eines Tricks bedient.

```
1  public team class GOTAdapter
2  {
3    int a = 0;
4
5    per(?c) {
6      int b = 0;
7    }
8
9    int c = b;
10 }
```

(a) Es wird ein einfaches GOT Team mit einem per Element gezeigt. Hervorzuheben ist hierbei die Initialisierung des Feldes `b` innerhalb des per Blocks und die Verwendung dieses Feldes `b` als Initialisierung für das Feld `c`. Das Feld `b` befindet sich in einem für das Feld `c` unzugänglichen Namensraum.



(b) Dargestellt ist die OT AST Repräsentation des in Listing 5.2a dargestellten Quellcodes.

Abbildung 5.2: Dargestellt ist die Transformation vom links dargestellten Quellcode in den rechts dargestellten OT AST. Dabei wird durch die Darstellung des per Elements als Klasse, ein unerwünschter, neuer Namensraum eröffnet.

Sobald das per Element vom Parser erstellt wurde und seinem übergeordneten Knoten als Kindknoten hinzugefügt werden soll, schaltet sich ein Callin dazwischen. Dieses Callin hat folgende Aufgaben:

- Alle im per Element enthaltenen Features in das dem per Element übergeordnete Element kopieren. Hierbei muss die Reihenfolge im Sourcecode eingehalten werden.
- Alle Referenzen der im per Element enthaltenen Features auf ihr übergeordnetes Element auf das dem per übergeordnete Element leiten.
- Eine `GOTClassDeclaration` oder `GOTMethodDeclaration` im GOT AST anlegen und diesen das per Element als Member übergeben.
- Das per Element vom Stack löschen und somit ein Hinzufügen zum übergeordneten Knoten verhindern.

Diese Aufgaben müssen sowohl für einen per Block im Klassenkontext als auch im Methodenkontext durchgeführt werden.

Der aus dieser Transformation resultierende OT AST und GOT AST werden in Abbildung 5.3 gezeigt. Der so entstandene OT AST durchläuft ohne weitere Anpassungen den Compiler, da alle Elemente aus dem Namensraum des per Elementes entfernt wurden. Dennoch sind alle Informationen mit dem Umweg über den GOT AST abrufbar. In einem weiteren Schritt, am Ende des Compilerprozesses, wird aus diesen Informationen ein gültiger und aufgelöster GOT DOM AST erzeugt.

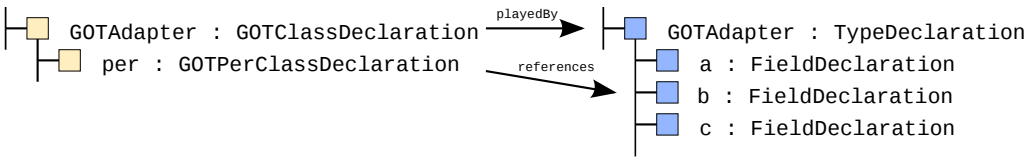


Abbildung 5.3: Hier dargestellt ist die OT und GOT AST Repräsentation des in Listing 5.2a dargestellten Codebeispiels. Der für den GOT Compiler relevante AST ist die Kombination aus diesen beiden ASTs. Das per Element wird ausschließlich im GOT AST gehalten, um ein fehlerfreies Resolving der im per Element befindlichen Elemente zu gewährleisten. Im Gegensatz zur Repräsentation in Abbildung 5.2b, wo die Kenntnis über das per Element einen Resolving Fehler bei der Initialisierung des Feldes `c` hervorruft.

Parsen ohne Methodenrümpfe: diet Parsing

Um diet-Parsing für GOT zu unterstützen, müssen einige Anpassungen vorgenommen werden. Neben dem Überspringen während der ersten Phase des Parsens, bei dem die Typhierarchie aufgebaut wird, muss außerdem eine Möglichkeit geschaffen in der zweiten Phase die einzelnen Methodenrümpfe explizit nachzuparsen. Die dafür notwendigen Änderungen werden im Folgenden erläutert.

Querymethoden, `match` und `declare` haben syntaktisch den gleichen Body. Er entspricht dem in Abschnitt 2.1.2 beschriebenen Querymethoden Body. Um diesen einzeln zu parsen, muss der Grammatik ein neues Goal hinzugefügt werden (siehe Abschnitt 5.1.2).

Das Parsen der Methodenrümpfe wird mit dem Aufruf der `parse(MethodDeclaration, CompilationUnit)` Methode gestartet. Die Methodenausführung wird mit Hilfe eines replace Callins im GOT Parser abgefangen. Sofern es sich bei der übergebenen CompilationUnit um eine GOT CompilationUnit handelt, wird das spezielle Goal für das Parsen von Querymethoden aktiviert. Ein Goal wird aktiviert, in dem das ihm zugeordnete Startsymbol als erstes Token der Parsierung auf den Stack gelegt wird. Somit kann eine Reduzierung zum Accept Symbol nur erfolgreich durchgeführt werden, sofern der Weg über das zugeordnete Startsymbol läuft. Nach dem erfolgreichen Parsen der Rümpfe, werden die gewonnen Informationen in den AST Knoten der zugehörigen Methode eingetragen.

Aktivierung

Der GOT Parser muss immer aktiv sein, da er die abgeänderte `consumeRule(int ruleId)` Methode enthält. Dies bedeutet jedoch nicht das alle Callins, zu jeder Zeit aktiv sein müssen. Ein Großteil der Callins wird nur für die Generierung von GOT AST Knoten benötigt. Die Aktivierung dieser Callins wird mit Hilfe von Guards an den Callins gesteuert. Diese Guards greifen dabei auf eine globale Variable zu, die nur gesetzt wird, sofern es sich bei der zu parsenden CompilationUnit um eine GOT CompilationUnit handelt. Das Setzen der Variable (`ast`), die gleichzeitig auch eine Instanz des GOT AST ist, wird nach dem Parse Beginn der CompilationUnit durchgeführt. Hierfür wird ein Callin auf die `consumeEnterCompilationUnit()` Methode verwendet. Diese Methode wird immer dann aufgerufen, wenn das Parsen einer neuen CompilationUnit gestartet wurde.

5.4 Resolving und Typechecking

An diesem Punkt im Compilierungsprozess sind die CompilationUnits von ihrer Quellcode Darstellung in ASTs umgewandelt worden. Es gilt nun eine Typhierarchie zu erstellen und alle Namen in den ASTs eindeutig aufzulösen. Wir befinden uns also in den in Abbildung 5.1 dargestellten Phasen 1.2 und 2.2 des Compilierungsprozesses.

Die in den Phasen 1.1 und 2.1 erzeugten ASTs sind zu diesem Zeitpunkt noch nicht vollständig. Sie enthalten eine Vielzahl von Namen, die entweder für Typen, Variablen oder Features stehen. Aus den ASTs ist zu erkennen, ob es sich bei diesen Namen um einen Typ oder eine Variable handelt, es ist jedoch noch nicht eindeutig bestimmt um *welchen* Typ, oder *welche* Variable es sich handelt. Da Object Teams Namensräume unterstützt wäre es denkbar, dass zwei Namen zwar den gleichen Identifier besitzen, aber für unterschiedliche Typen stehen, da ihr Namensraum unterschiedlich ist. So kann `List` für die Klasse `List` aus dem Package `java.util` oder dem Package `java.awt` stehen. Um Eindeutigkeit zu schaffen, ist es also notwendig, dass für einen Namen eindeutig bestimmt wird für was er steht. Es muss eine Referenz zu dem ihm zugehörigen Typ oder Feature hergestellt werden. Nach dem Parsen sind alle Referenzen unaufgelöst. Erst im weiteren Schritt des Resolvings werden diese Referenzen aufgelöst und damit ihre Ziele bestimmt. Sobald eine Referenz aufgelöst wurde, kann mit Hilfe der für die Sprache festgelegten Typechecking Regeln überprüft werden, ob der der Referenz zugeordnete Name korrekt verwendet wurde.

Es müssen zwei Aufgaben erledigt werden. Zum einen müssen alle Referenzen aufgelöst werden. Zum anderen muss überprüft werden, ob die Typisierungsregeln der Sprache eingehalten wurden. Für Java und OT ist sind diese Aufgaben schon vollständig erledigt.

Für GOT wurden jedoch zusätzlich zu den bekannten Referenzen noch Metavariablen eingeführt, die bestehende Java Referenzen ersetzen können und das Typsystem erweitern. Metavariablen sollen ähnlich wie herkömmliche Variablen auch einem Typsystem unterliegen. Dieses Typsystem fordert, dass alle Metavariablen vor ihrer Verwendung mit einem Metatyp deklariert werden. Außerdem unterliegen Metavariablen wie herkömmliche Variablen, Typisierungsregeln, die für GOT aufgestellt wurden. Das Typsystem von GOT und die Erweiterung des Resolvings für GOT, werden in diesem Abschnitt erläutert.

5.4.1 Das Generic Object Teams Metavariablen Typsystem

Jede Metavariable muss mit einem Metatyp vor ihrer Verwendung deklariert werden. Die dafür zur Verfügung stehenden Metatypen werden im folgenden aufgelistet.

- **?Class** konform zu java.lang.Class
- **?Type** konform zu java.lang.reflect.Type
- **?Method** konform zu java.lang.reflect.Method
- **?Field** konform zu java.lang.reflect.Field
- **?Modifier** konform zu java.lang.reflect.Modifier
- **?String** konform zu java.lang.String
- **?int** konform zu int / java.lang.Integer
- **?boolean** konform zu boolean / java.lang.Boolean

Wie aus der Auflistung zu erkennen ist, wurde zu jedem Metatyp ein konformer Javatyp ausgewählt. Diese Zuordnung hat den Vorteil, dass Metatypen in der Basis des GOT Compiler, dem OT Compiler, als ihr konformer Typ auftreten können und somit einige Funktionalität des Compilers wiederverwendet werden kann. Dazu später mehr. Außerdem ermöglicht die Typisierung als Reflection Typen, den Zugriff auf das Java Reflection API.

Implementierung des Typsystems

Um die Konformität zwischen Metatyp und Java Repräsentant herzustellen, muss ein Abbildung zwischen ihnen hergestellt werden. Dafür wurde die Klasse `GOTMetaTypeMapping` erstellt, die es als Aufgabe hat, alle Konformität bezogenen Aufgaben zu regeln. Die Klasse bietet Methoden, die für einen Metatyp den entsprechenden Javatyp liefern und umgekehrt. Neben einer solchen Abbildung wird auch die Kompatibilität zwischen zwei Metatypen über die Klasse `GOTMetaTypeMapping` festgelegt. Es sind alle Metatypen zu sich selbst kompatibel. Zusätzlich ist `?Class` zu `?Type` kompatibel. Die Kompatibilität von `?String`, `?int` und `?boolean` zu `?Class` ist bereits über Java gegeben.

Mit Hilfe des Aspekts `GOTLookupAdaptor` wird die Ausführung der Methode `LookupEnvironment.findType` abgefangen. Diese Methode ist für das Nachschlagen von Typen im Compiler zuständig. Es wird die Klasse `GOTMetaTypeMapping` bemüht, um dem übergebenen Metatyp einen Javatyp zuzuordnen. Mit Hilfe des Javatyps kann der Compiler die Metavariable auflösen und auf etwaige Typfehler prüfen. Dieses Verfahren ist jedoch nicht universell für alle Stellen, an denen Metavariablen eingesetzt werden, verwendbar. Die Unterschiede und Spezialfälle werden im Laufe einer detaillierten Betrachtung des Resolvings von Metavariablen erläutert.

5.4.2 Resolving von Metavariablen

Um Metavariablen vollständig aufzulösen, werden zusätzlich zum AST zwei weitere Datenstrukturen benötigt. Diese sind Scopes und Bindings. Wie in Abschnitt 4.5 und 4.4 angerissen, sind diese Strukturen essentiell für die Auflösung von Referenzen und das Typechecking. Ein Scope bietet die Möglichkeit innerhalb von Namensräumen nach Features und Typen zu suchen und ist für die Erstellung der Typ-Hierarchie zuständig. Bindings stellen die Verknüpfung zwischen Namen und Typ oder Feature dar. Ein Name ist eindeutig, sofern er ein aufgelöstes, gültiges Binding besitzt. Die wichtigsten Bindings für GOT sind das `TypeBinding`, `VariableBinding` und `FieldBinding`. Ein `TypeBinding` beschreibt eine Referenz von einem Namen zu einem Typ. Ein `VariableBinding` bildet die Verbindung zwischen einer Variable und ihrer Deklaration. Und ein `FieldBinding` ist ein spezielles `VariableBinding`, bei dem die Variablen Deklaration ein Feld ist.

Doppeldeutigkeit von Metavariablen

Wird eine Metavariable an einer Stelle eingesetzt, wo sie eine Javavariable ersetzt, wie in Listing 5.18, Zeile 4 gezeigt, kann die Metavariable ohne weitere Anpassungen am Compiler, ihrem Metatyp zugeordnet und überprüft werden. Diese Auflösung ist möglich, da Metavariablen zu Javavariablen bezüglich ihrer Bindings identisch sind. Für beide Variablen müssen zwei Bindings erstellt werden, um sie eindeutig aufzulösen. Zum einen muss ein `VariableBinding` zu der zugehörigen Variablen Deklaration erstellt werden und zum anderen muss ein `TypeBinding` zu ihrem Typ erstellt werden.

Im Fall der Ersetzung einer Javavariable durch eine Metavariable, kommt die Repräsentation einer Metavariable als Rolle, die an eine herkömmliche Javavariable gebunden ist, zu gute. Es wird dank der Rollen-basierten Umsetzung von GOT Knoten, die der Metavariable zugrunde liegende Basis (Javavariable) verwendet, um deren Deklaration zu bestimmen. Da die Deklaration von Metavariablen als Javafelder statt findet, die in das dem OT Compiler bekannte Team kopiert wurden, ist die gesuchte Deklaration analog zu einem regulären Feld vom Compiler aufzufinden. Der Typ der Variable wird dann mit der in Abschnitt 5.4.1 adaptierten `findType` Methode bestimmt. Daraufhin kann die Verwendung der Variable bezüglich auf ihre Typisierung überprüft werden. Die somit aufgelöste Metavariable besitzt das notwendige `FieldBinding` (`VariableBinding`) und `TypeBinding`.

```
1 ...
2 public void saveAll() {
3   per(?someFields) {
4     save(obj.?someFields);
5   }
6 }
7 ...
```

Listing 5.18: Das Listing zeigt einen Ausschnitt aus einem GOT Aspekt. Dabei wird für jedes Feld, welches die Metavariable `?someFields` im Objekt `obj` enthält, die `save` Methode aufgerufen. Interessant an diesem Beispiel ist das die in Zeile 4 gezeigte Metavariable an einer Stelle verwendet wird wo vorher ebenso eine Javavariable stehen könnte und daher das Resolving „Out of the box“ funktioniert.

```
1 ...
2 per(?dataClasses) {
3   protected class GenericRole playedBy ?dataClasses {
4   }
5 }
6 ...
```

Listing 5.19: Es wird eine generische Rolle gezeigt. Diese Rolle wird für jede Klasse in der Menge `?dataClasses` erzeugt. Die Metavariable `?dataClasses` wird nach dem `playedBy` Keyword verwendet beschreibt damit eine Referenz zu einem Typ, der Basis der Rolle.

Metavariablen werden jedoch nicht nur als Ersatz für Javavariablen verwendet. Sie werden auch an Stellen verwendet, an denen ein Name einen Typ referenziert (siehe Listing 5.19). Typpreferenzen unterscheiden sich bezüglich ihrer Bindings von Metavariablen. Sie besitzen nur ein `TypeBinding`, aber kein `VariableBinding`, da sie direkt auf einen Typ zeigen und nicht der Umweg über eine Variablen Deklaration gegangen werden muss. Da für Typpreferenzen kein `VariableBinding` vorgesehen ist, wird der Compiler auch nicht probieren ein solches zu erstellen. Für Metavariablen ist dieses Binding jedoch essentiell. Denn ohne `VariableBinding` ist keine Referenz zur Deklaration vorhanden, die wiederum verantwortlich für den Typ der Metavariable ist und das `TypeBinding` liefert. Für Metavariablen gilt also, dass ohne `VariableBinding` kein `TypeBinding` erstellt werden kann.

Die Verwendung von Metavariablen für Typpreferenzen erfordert eine Adaptierung der Resolving Mechanismen des Compilers. Der Compiler muss für solche Metavariablen ein `VariableBinding` erstellen, welches genutzt werden kann, um das notwendige `TypeBinding` zu erzeugen.

Um die Veränderungen am Compiler möglichst gering zu halten, wird sich abermals eines Tricks bedient. Da die Auflösung von Metavariablen als Ersatz von Javavariablen schon funktioniert, wird der Typpreferenz eine unsichtbare Javavariable angehängt. Dafür wird dem zugehörigen GOT Knoten der Typpreferenz (`GOTSingleTypeReference`) ein Member hinzugefügt. Dieses Member ist vom Typ `GOTSingleNameReference` und repräsentiert eine Metavariable auf Basis einer Javavariable. Diese Variable bekommt den gleichen Identifier, wie die Typpreferenz und wird mit Hilfe der bereits für Variablen funktionierenden Mechanismen aufgelöst. Das daraus

entstehende `TypeBinding` wird in die Typpreferenz eingetragen. Das `VariableBinding` kann bei Bedarf aus der `GOTSingleNameReference` gelesen werden. Außerdem muss der Typ der Metavariable überprüft werden. Es dürfen keine nicht zu `?Type` kompatiblen Typen an Stelle von Typpreferenzen verwendet werden. Abbildung 5.4 zeigt den daraus entstehenden GOT AST für eine `GOTSingleTypeReference`, welche für Metavariablen auf Typpreferenzen verwendet wird.

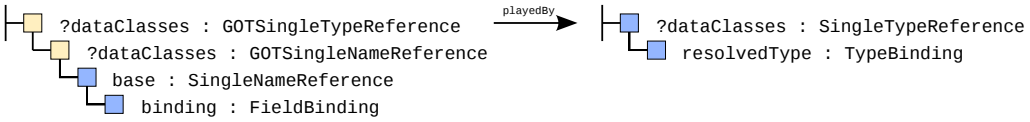


Abbildung 5.4: Abgebildet ist der GOT AST einer Metavariable im Einsatz als Typpreferenz. Die im OT AST vorhandene `SingleTypeReference` wird durch eine Rolle um eine herkömmliche Javavariable (`SingleNameReference`) erweitert. Diese Javavariable kann mit existierender Compiler Funktionalität aufgelöst werden. Dadurch entstehen `TypeBinding` und `FieldBinding` (`VariableBinding`).

Neben Typpreferenzen und Javavariablen existieren noch weitere OT Elemente, die durch Metavariablen ersetzt werden können. Dazu zählen beispielsweise Feldreferenzen und Methodenreferenzen für die Verwendung in Callout- und Callinbindungen. Deren Adaption greift ebenfalls auf die Einführung einer zusätzlichen `SingleNameReference` für Metavariablen zurück. Um näher auf die Details der verwendeten Adaptionen eingehen zu können, muss jedoch zu erst der Ablauf des Resolving und Typechecking im Compiler etwas näher erläutert werden.

Ablauf des Resolving und Typechecking

Vorerst soll der grobe Ablauf des Resolvings und Typecheckings im Eclipse JDT (OTDT) beschrieben werden.

An erster Stelle, steht die Erstellung einer Typhierarchie (siehe Phase 1.2 in Abbildung 5.1). Der Parser parst dafür die CompilationUnits und erstellt für jede einen eigenen Compiler AST. Dabei werden alle Methodenrümpfe ausgelassen (diet-Parsing). In nächsten Schritten werden `LookupEnvironment` und `CompilationUnitScopes` für die geparsten `CompilationUnit Declarations` erstellt und die gesamten ASTs runter bis auf Methodenebene rekursiv durchlaufen. In diesem Durchlauf werden entsprechende Scopes für Klassen, Teams, Rollen, Methoden, Callins und Callouts erstellt. Die Verknüpfungen zwischen den Typen und Features werden mit Hilfe von Bindings hergestellt. Bindings werden in die AST Knoten eingetragen. Dabei benötigte, neu entdeckte CompilationUnits werden in den Prozess integriert und durchlaufen ebenfalls die beschriebenen Schritte.

Der Aufbau der Typhierarchie erfolgt mit Hilfe von Methoden in den Scopes, die die Präfixe `build` und `create` tragen. Diese Methoden erstellen die notwendigen Bindings und Sub Scopes. Diese Scopes rufen wiederum deren Sub Scopes auf. Die Scope Hierarchie wird rekursiv abgelaufen.

Nachdem die Typhierarchie erstellt wurde, werden die ausgelassenen Methodenrümpfe vom Parser geparst und in die bestehenden ASTs eingepflegt. Die ASTs verfügen jetzt über eine große Menge von Variablen und Referenzen, die aus den Statements der Methodenrümpfe resultieren. Hier

beginnt das eigentliche Resolving (siehe Phase 2.2 in Abbildung 5.1). Es werden rekursiv von oben nach unten alle AST Knoten durchlaufen. Dies erfolgt in dem auf den CompilationUnits die `resolve` Methoden aufgerufen werden. Diese Methoden rufen rekursiv auf den Kinder Knoten äquivalente Methoden mit dem Präfix `resolve` auf, welche für die Erstellung der ausstehenden Bindings zuständig sind.

Um doppeltes Durchlaufen zu vermeiden, wird nach dem Auflösen eines Typs oder einer Referenz eine entsprechende Typüberprüfung durchgeführt und die dabei gefunden Typfehler gesammelt und dem Tooling gemeldet.

Metavariablen als Referenz auf eine Basis

Für Rollen, die eine Basis zugeordnet bekommen, ist es wichtig, dass die Referenz auf ihre Basis aufgelöst wird. Der Basistyp bestimmt den Typ der Rolle. Für Features wie Callins und Callouts und die Vererbung ist der aufgelöste Basistyp eine Grundlage, um weitere Typüberprüfungen durchführen zu können.

Hinter dem `playedBy` Keyword wird der Basistyp angegeben. Der dort aufgeführte Name wird im Compiler AST als `SingleTypeReference` dargestellt. Enthält diese Referenz eine Metavariable, wird der GOT Knoten `GOTSingleTypeReference` aktiv. Dieser Knoten löst bei Bedarf die Metavariable, wie in Abschnitt 5.4.2 beschrieben, mit Hilfe einer `GOTSingleNameReference` auf. Dafür bietet der Knoten `GOTSingleTypeReference` eine `buildType(Scope)` Methode. Diese Methode wird vom zugehörigen Scope aufgerufen, sofern die Basis der Rolle bestimmt werden soll. Die Methode `findBaseclass` des Scopes muss dafür mit einem replace Callin adaptiert werden, um den von der `GOTSingleTypeReference` angebotenen Mechanismus (`buildType`) zum Auflösen der Metavariable aufzurufen. Nach dem Auflösen der Metavariable muss deren Typ überprüft werden, da nur Metatypen, die kompatibel zu `?Class` sind, als Basisreferenz zugelassen werden können.

Das Team `GOTLookupAdaptor` ist für die Adaptierung aller Scopes zuständig, die für die Typhierarchie Phase angepasst werden müssen. So auch für den `ClassScope`, der mit einer `GOTClassScope` Rolle adaptiert wurde, um das Auffinden einer Basis in Form von Metavariablen zu ermöglichen.

Da der Typ einer Basisreferenz, die durch eine Metavariable ersetzt wurde immer der konforme Typ der Metavariable und damit `java.lang.Class` ist, entstehen einige Fehlermeldungen. Diese Fehlermeldungen sind für das GOTDT irrelevant, da die vorliegende generische Rolle nicht direkt in Bytecode umgesetzt wird. Folgende Fehlermeldungen treten auf.

- Dekapselung einer finalen Klasse (`decapsulationOfFinal`)
- Weben in Systemklasse (`tryingToWeaveIntoSystemClass`)

Diese Fehlermeldungen werden für generische Rollen vom `GOTProblemReporter` abgefangen und entfernt (siehe Abschnitt 5.4.3).

Metavariablen in Callin- und Calloutbindungen

Callin- und Calloutbindungen sind im technischen Sinne Mappings zwischen Rollen- und Basisfeatures. Ein solches Mapping besteht immer aus zwei Seiten. Die linke Seite steht für Rollenfeatures und die rechte Seite für Basisfeatures. Innerhalb der auf diesen Seiten spezifizierten Features ist es möglich Metavariablen an Stelle von Referenzen auf Methoden und Feldern zu verwenden. Da die linke Seite dieser Mappings, auf der Rollen Seite, immer auf eine Methode zeigen muss, wird diese vom dem Compiler AST Knoten `MethodSpec` gebildet. Die rechte Seite, also die der Basis, wird bei Callins immer von einem oder mehreren `MethodSpec` Knoten gebildet. Bei Callouts ist es entweder ein `MethodSpec` Knoten oder im Fall eines Callout-To-Field ein `FieldAccessSpec` Knoten, welcher einen `MethodSpec` Knoten spezialisiert.

Die Verwendung einer Metavariable in einer Callin- oder Calloutbindung kennzeichnet sich durch den entsprechenden Metaidentifier in den `MethodSpec` oder `FieldAccessSpec` Knoten.

Die Auflösung von Callins und Callouts kann in Object Teams erst nach dem Aufbau der Typhierarchie erfolgen, da vorher keine Binding auf eine Basisklasse existiert und somit auch keine Basisfeatures aufgelöst werden können. Daher wird das Resolving innerhalb der Compiler AST Knoten vorgenommen. Dafür dienen, die im Abschnitt 5.4.2 vorgestellten `resolve` Methoden der Knoten. Die in diesem Fall zuständige Methode ist `MethodSpec.resolveFeature`. Mit Hilfe des GOT Knotens `GOTMethodSpec` wird diese Methode adaptiert und ähnlich den vorherigen Adaptionen ein `GOTSingleNameReference` Knoten für die Metavariable erzeugt und aufgelöst. Die daraus gewonnen Informationen werden in den `GOTMethodSpec` Knoten eingetragen. Da der Knoten `FieldAccessSpec` vom `MethodSpec` Knoten erbt ist hinsichtlich dieser Auflösung keine weitere Anpassung notwendig. In einem weiteren Schritt jedoch muss der aufgelöste Typ der Metavariable geprüft werden. So darf in einer `MethodSpec` nur eine Metavariable vom Typ `?Method` verwendet und in der Klasse `FieldAccessSpec` nur eine Metavariable vom Typ `?Field` verwendet werden. Dafür wird eine `checkType` Methode in dem Knoten `GOTMethodSpec` eingeführt, die von der erbenden `GOTFieldAccessSpec` Rolle auf die entsprechende Typüberprüfung überschrieben wird.

Zur Verdeutlichung des Resolvings von Metavariablen mit Hilfe des `GOTSingleNameReference` Knotens, wird der für das Resolving zuständige Teil des `GOTMethodSpec` Knotens in Listing 5.20 gezeigt. Da der Compiler bereits Metavariablen im Knoten `SingleNameReference` auflösen kann, wird zusätzlich zum eigentlichen `GOTMethodSpec` Knoten ein `GOTSingleNameReference` Knoten erstellt, der die gewünschten Bindings erstellt.

Bei der Verwendung von Metavariablen für die Referenzierung von Features in Callin- und Calloutbindungen, treten einige Fehlermeldungen auf, die für das GOTDT keine Relevanz haben.

- Methode oder Feld konnte nicht gefunden werden (unresolvedMethodSpec + boundMethod-Problem)

Diese für GOT irrelevanten Fehlermeldungen werden vom `GOTProblemReporter` entfernt.

```
1  ...
2  // Callins
3  void resolveFeature (ReferenceBinding receiverType, BlockScope scope, boolean callinExpected,
4      boolean isBaseSide, boolean allowEnclosing)
5    <- replace
6      void resolveFeature (ReferenceBinding receiverType, BlockScope scope, boolean
7        callinExpected, boolean isBaseSide, boolean allowEnclosing);
8  //
9
10 @SuppressWarnings("basecall")
11 callin void resolveFeature(ReferenceBinding receiverType, BlockScope scope,
12   boolean callinExpected, boolean isBaseSide, boolean allowEnclosing) {
13
14   metavar = new GOTSingleNameReference(this.getSelector(), this.getSourceStart(), this.
         getSourceEnd());
15   Binding binding = metavar.resolveMetavar(scope);
16
17   checkType(scope, binding);
18 }
19 ...
```

Listing 5.20: Das Listing zeigt den für das Resolving zuständigen Teil des `GOTMethodSpec`-Knotens. Der Knoten ist für die Auflösung von Metavariablen auf Feldern und Methoden Spezifikationen in Callins und Callouts zuständig. Das Auflösen der Metavariable erfolgt über einen zusätzlichen `GOTSingleNameReference` Knoten.

Resolving qualifizierter Referenzen mit Metavariablen Anteilen

Neben einfachen Referenzen wie `?someClass`, sollen Metavariablen auch in qualifizierten Referenzen wie `somepackage.another.one.""?someClass` möglich sein. Um solche Referenzen aufzulösen ist es wichtig, das letzte Glied der Qualifizierung aufzulösen, um den Typ der qualifizierten Referenz zu bestimmen. Dafür wird die `resolveType` Methode von dem GOT Knoten `GOTQualifiedNameReference` adaptiert und das letzte Element aus der Qualifizierung ausgewählt. Für die darin enthaltene Metavariable wird dann wie üblich eine `GOTSingleNameReference` erstellt und aufgelöst. Aus dem daraus resultierenden Binding lässt sich der Typ der qualifizierten Referenz festlegen. Eine Einschränkung von Metatypen ist hierbei nicht notwendig.

Es wurden bisher nur qualifizierte Metareferenzen betrachtet, die nur als letztes Glied eine Metavariable enthalten. Metavariablen im Package Anteil der Qualifizierung wären ebenso denkbar, wurden jedoch im Rahmen dieser Arbeit nicht umgesetzt.

Resolving von per Blöcken

Per Blöcke wurden im GOT Parser vorsätzlich vor den weiteren Phasen im OT Compiler versteckt. Dies hat zum Vorteil, dass die Elemente innerhalb der per Blöcke regulär vom OT Compiler aufgelöst und überprüft werden können.

Andernfalls hätte die durch die Blöcke veränderte Hierarchie der Elemente im Compiler AST zu Problemen im Resolving geführt. Wären die Blöcke nicht versteckt worden, hätte der Compiler für die für per Blöcke verwendeten Compiler AST Elemente, `TypeDeclaration` und `Block` in der Typhierarchie Erstellungs Phase, eigene Namensräume (Scopes) erstellt. Diese hätten beim Resolving erhebliche Code Anpassung gefordert, um die gewünschte Funktionalität zu erreichen.

Da die per Blöcke nicht Teil des Compiler AST sind, werden sie ohne Anpassungen nicht vom Resolving erfasst. Daher werden GOT AST Knoten, die Referenzen auf per Blöcke halten können, `GOTClassDeclaration` und `GOTMethodDeclaration`, derart erweitert, dass ein Aufruf auf die `resolve` Methoden ihrer Basen im OT Compiler AST auch an die enthaltenen per Blöcke weitergeleitet wird. Dafür bietet die gemeinsame Oberklasse der per Blöcke im Compiler AST `GOTPerBlock` eine `resolve` Methode an, die das Resolving eines per Blocks startet.

Die `resolve` Methode der per Blöcke hat jedoch relativ wenig zu tun, da alle Elemente innerhalb eines per Blocks schon von dem per Block übergeordneten Element aufgelöst werden. Dies ist der Fall, da alle Elemente für diesen Zweck vom GOT Parser dorthin verschoben wurden. Es bleibt für den per Block nur die Auflösung der per-spezifischen Elemente. Spezifisch enthält ein per Block eine Argumentliste von Metavariablen oder einen weiteren per Block. Die Argumente werden als Array von Knoten des Typs `GOTSingleNameReference` gespeichert. Dank dieser Darstellung genügt ein einfacher Aufruf der `resolve` Methoden der Argumente, um diese aufzulösen und auf Typfehler zu prüfen.

5.4.3 Constraints

Im Zuge der Einführung von Metavariablen entstehen zusätzliche Constraints in Bezug auf da die für herkömmliches Object Teams keine Relevanz haben. Dabei kann man zwei Arten von Constraints unterscheiden. Zum einen solche, die durch die Einführung von neuen Strukturen von Generic Object Teams entstehen und zum anderen solche, die im Zuge des Einsatzes von Metavariablen und Queries die Integrität zwischen Metavariablen und Queries wahren müssen. Diese Constraints, die sich auf Queries beziehen, werden im Folgenden implizite Constraints genannt.

Constraints durch neue Sprachelemente

Durch die Typisierung von Metavariablen und die Einführung von neuen Elementen wie dem `per` Block entstehen einige zusätzliche Constraints, die durch das Typsystem erkannt werden müssen. Diese werden im Folgenden beschrieben.

1. Hinter dem Keyword `playedBy` steht der Basistyp der Rolle. Dieser Typ muss eine Klasse sein. Eine Metavariable benötigt an dieser Stelle also zwingend den Typ `?Class`.
2. Eine Metavariable, die für das Basis Feld eines Callout-to-Field eingesetzt wird, muss vom Typ `?Field` sein.
3. Metavariablen im Basis Teil eines Callout-to-Method oder Callins müssen vom Typ `?Method` sein.
4. Jede Metavariable außerhalb eines Queries muss zwingend von einem per Block umschlossen sein, der als Argument diese Metavariable enthält.

Implementierung des per Constraints

Für die Generierung ist es zwingend notwendig, dass alle Metavariablen per Blocks zugeordnet werden können. Es ist daher sinnvoll schon im Tooling nur Quellcode für die Generierung zuzulassen, bei dem alle Metavariablen von per Blöcken umgeben sind. Dafür muss eine Überprüfung auf dieses Constraint in den Compiler integriert werden.

Für die Überprüfung bietet sich der `GOTSingleNameReference` Knoten an, da jede Metavariable vom Typ `GOTSingleNameReference` ist. Es ist sinnvoll direkt nach dem Resolving einer Metavariable zu prüfen, ob diese auch von einem per Block umgeben ist, der sie quantifiziert. Dafür wird am Ende der `resolve` Methode des Knotens die `checkForSurroundingPer(Scope)` Methode aufgerufen. Diese Methode ermittelt mit Hilfe des vom Resolving bekannten Scope die übergeordnete Methode oder Klasse. Diese Methode oder Klasse wird dann auf ihre enthaltenen per Elemente untersucht. Befindet sich ein per Element in dem übergeordneten Element, wird überprüft ob die Metavariable in der Argumentliste des per Blocks enthalten ist und die Position des per Blocks im Quelltext die aktuelle Metavariable umschließt. Ist dies nicht der Fall wird die nächst höhere Methode oder Klasse überprüft. Wird kein passender per Block gefunden, wird eine Fehlermeldung mit Hilfe des `GOTProblemReporter` erzeugt. Nicht alle Metavariablen müssen auf ein per Element überprüft werden. Ausnahmen sind Metavariablen, die in einer Argumentliste eines per Blocks stehen, Metatypen und Metavariablen innerhalb von match, declare und `Query` Elementen.

Implizite Constraints durch Metavariablen und Queries

Metavariablen werden mit Hilfe von Queries belegt. Dabei erhält jede Metavariable, abhängig vom Query, eine Menge an Sprachelementen der Basis zugeordnet. So würde eine Metavariable `?myClasses`, die mit dem Query `beginsWith(?myClasses, "My")` belegt wird, nur Klassen enthalten, die mit „My“ beginnen. In diesem Beispiel wird davon ausgegangen, dass sich nur Klassen in der Metavariable befinden. Diese Einschränkung lässt sich über die Typisierung von Metavariablen festlegen. Wird die Metavariable mit dem Typ `?Class` deklariert, darf Sie nur Klassen enthalten. Auf der Ebene von Prolog ist die Typisierung der Metavariable nicht mehr bekannt. Prolog kennt nur Queries. Um zu einer korrekten Ausführung in Prolog zu gelangen impliziert das GOTDT den Query `isClass(?myClasses)` für diese Metavariable. Die Information, die sich über die Typisierung im GOTDT befindet, wird in einen Query übertragen.

Neben impliziten Constraints, die durch die Typisierung entstehen, gibt es noch Weitere. So werden zusätzliche Constraints häufig über die Position der Metavariable im GOT Code und der umgebenen Struktur impliziert. Als Beispiel sei die in Listing 5.21 gezeigte Callinbindung gegeben. In diesem Beispiel werden eine Reihe von Constraints impliziert. Über die Typisierung ist festgelegt, dass die Metavariable nur Methoden enthalten darf (`isMethod(?baseMethod)`). Außerdem müssen die Basismethoden eine zur Rollenmethode kompatible Signatur haben. Der Rückgabetyp muss allgemeiner oder gleich der Rollenmethode sein. Die Basismethode muss genügend Argumente für die Rollenmethode liefern. Die Argumente der Basismethode müssen gleich oder spezieller zu denen der Rollenmethode sein. Der für dieses Beispiel komplett implizierte Query wird in Listing

```
protected class GenericRole playedBy ?someBase
  match(?Class ?someBase, ?Method ?baseMethod) {
  ...
  }
{
  String roleMethod(int i) <- replace ?baseMethod;
  ...
}
```

Listing 5.21: Es wird eine Callinbindung gezeigt, die in der hier dargestellten Form mehrere Constraints impliziert. Diese müssen im zugehörigen Query abgebildet werden.

```
protected class GenericRole playedBy ?someBase
  match(?Class ?someBase, ?Method ?baseMethod) {
    isMethod(?baseMethod) && returnCompatible(?baseMethod, "String")
    && argumentCompatible(?baseMethod, 1, "int")
  }
{
  String roleMethod(int i) <- replace ?baseMethod;
  ...
}
```

Listing 5.22: Das Listing zeigt eine Callinbindung mit einem vollständigen Query. Es wurden bereits alle implizierten Constraints in den Query eingepflegt.

5.22 gezeigt. Die dabei verwendeten Aufrufe von Querymethoden in den Queries, werden in einer Bibliothek noch weiter verfeinert.

Im Folgenden ist eine Auflistung der gängigsten impliziten Constraints gegeben.

1. Constraints die durch die Typisierung entstehen. Zum Beispiel für den Metatyp `?Class` wird das Constraint `isClass(..)` impliziert.
2. Metavariablen auf Basis Referenzen müssen kovariant zum Basistyp, der Rolle von der sie erben, sein.
3. Metavariablen auf Methoden auf der Basis Seite einer Callin Bindung müssen einen kontravarianten Rückgabetyp und einen kovarianten Argumenttyp besitzen. Die Basismethode muss für die Rollenmethode entsprechende Argumente bieten.
4. Metavariablen auf Methoden auf der Basis Seite von Methoden Callouts müssen einen kovarianten Rückgabetyp haben. Die Argumente müssen kontravariant sein.

Implementierung der Erkennung von impliziten Constraints

Implizite Constraints können durch das Tooling erkannt und dem Nutzer mitgeteilt werden. Dieser sollte dann entscheiden können, ob er seine Queries manuell anpasst, oder sich vom Tooling eine Erweiterung für seinen Query erstellen und einpflegen lässt.

Grundlage dafür bildet die Erkennung des implizites Constraints. Dafür muss im Laufe der Code-Analyse Phase (siehe Phase 3.1 in Abbildung 5.1) für die entsprechenden Constraints eine Überprü-

fung durchgeführt werden. Die Code-Analyse besteht, ähnlich dem Resolving, aus einem rekursiven Aufruf von `analyse` Methoden von der `CompilationUnitDeclaration` an über alle Knoten des Compiler AST. Die einzelnen Knoten können, indem sie die `analyse` Methode überschreiben, den AST auf die Einhaltung von Constraints überprüfen. Eine bekannte Untersuchung in der Analyse Phase ist das Finden von unbenutzten Elementen.

Sofern ein impliziter Constraint erkannt wurde, müssen zwei weitere Aufgaben erledigt werden. Es muss eine Query Darstellung des Constraints erstellt werden und es muss überprüft werden, ob der für die Metavariablen zuständige Query das Constraint bereits abbildet. Im Idealfall würde man dafür den Query in eine Normalform überführen und prüfen, ob der für das Constraint erzeugte Query, teil dieses in die Normalform überführten Queries ist. Die Überführung in eine Normalform gestaltet sich für GOT Queries jedoch äußerst schwierig. Zum einen weil die Grundmenge von Termen noch nicht festgelegt ist und zum anderen weil eine Auswahl von Basis Elementen über verschiedenste Wege getroffen werden kann. Dabei erschwert die mögliche Verwendung von Zeichenketten in Querymethoden die Normalisierung zusätzlich. Eine dahin ausgerichtet gewählte, eingeschränkte Menge von Grundtermen könnte dieses Problem beseitigen. Hierbei steht die Möglichkeit zur maschinellen Auswertung gegen die Ausdrucksstärke von Queries.

Für den Übergang wird ein vereinfachtes Verfahren zur Erkennung von Sub Queries in Queries verwendet. Die Queryexpression wird von rechts nach links durchwandert bis entweder ein von dem Operator „&&“ (und) verschiedener Operator oder der gesuchte Query gefunden wurde. Abbildung 5.5 zeigt beispielhaft einen vollständig geklammerten Query in textueller und AST Darstellung. In diesem Query soll nach dem erstellten Constraint `compatible?sub, ?baseclass` gesucht werden. Dafür wird der oben beschriebe Algorithmus wie folgt angewendet:

1. Ist der aktuelle Knoten ein „&&“ Knoten (1, 3, 5), prüfe ob es sich bei dem rechten Zweig um den gesuchten Sub Query handelt (2, 4, 6).
2. Falls kein Ergebnis im rechten Zweig gefunden werden konnte, wandere den linken Zweig hinab.
3. Handelt es sich bei dem aktuellen Knoten um einen „&&“ Operator Knoten (3, 5), gehe zu Schritt 1. Andernfalls, beende die Suche ergebnislos.

Implementierung von Constraint Nummer 2 im Detail

Beispielhaft soll eine Erkennung für das implizite Constraint Nummer 2 in das Generic Object Teams Development Tooling eingebaut werden. Dafür muss zum einen das Constraint erkannt und in eine Compiler Warnung umgesetzt werden und zum anderen dem Nutzer eine Möglichkeit zur Verfügung gestellt werden, das Constraint durch das Tooling in seinen Query einpflegen zu lassen.

Das Constraint Nummer 2 besagt, dass die Basisreferenz einer Rolle, die von einer anderen Rolle erbt, gleich oder spezieller als die Basisreferenz der Super Rolle sein muss.

Rollen mit einer Metavariable als Basisreferenz werden in GOT mit dem `GOTClassDeclaration` Knoten dargestellt. Dieser GOT Knoten hat als Basis einen `TypeDeclaration` Knoten, von dem er die `analyseCode` Methode adaptiert. Ein Aufruf dieser Methode startet die eigentliche Analyse

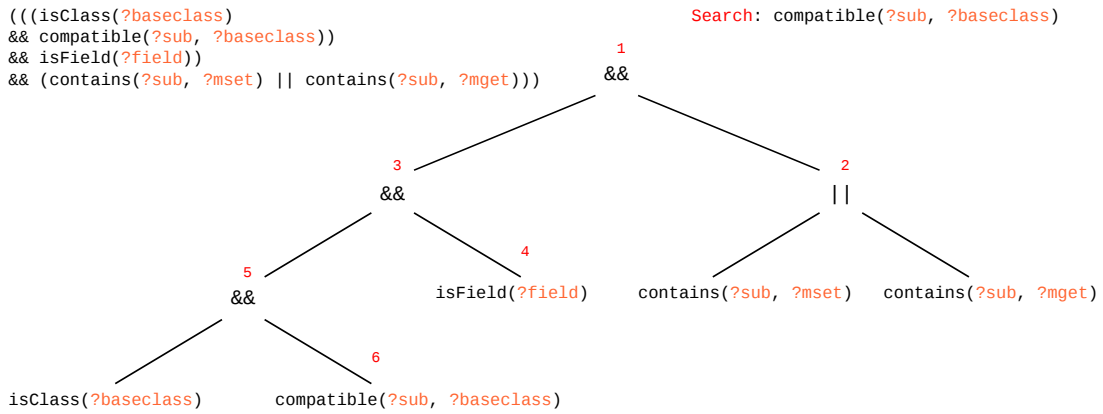


Abbildung 5.5: Die Abbildung zeigt einen Query in textueller und AST Repräsentation. In Rot werden die für die Suche nach dem Sub Query `compatible(?sub, ?baseclass)` notwendigen Schritte dargestellt. Der verwendete Algorithmus ist die Grundlage für die Erkennung von impliziten Constraints und deren Erkennung in vom Nutzer spezifizierten Queries durch das Tooling.

des Constraints in der `checkMetaBaseClassRedefinition` Methode. Für die Überprüfung werden die Basisreferenz der Rolle und die ihrer Superklasse bestimmt. Handelt es sich bei beiden um nicht identische Metavariablen, liegt das implizite Constraint vor. Eine Constraint Factory erzeugt daraufhin ein Objekt der Klasse `ImplicitCompilerConstraint`, welches eine Compiler AST Repräsentation des Constraints enthält. Mit Hilfe dieser Repräsentation kann über die `QueryEngine` geprüft werden, ob der zugehörige Query das gesuchte Constraint bereits enthält. Ist dies nicht der Fall, wird mit Hilfe des `GOTProblemReporter` eine Warnung erzeugt. Die Warnung bekommt eine String Serialisierung des Constraints als Argument zugewiesen, um in einem späteren Schritt das Constraint exakt bestimmen und dem Nutzer Lösungsvorschläge unterbreiten zu können.

Im Userinterface des GOTDT werden mit Hilfe des Quickfix Extension Points Lösungsvorschläge erstellt. Dafür wird für jede Warnung, die auf einen impliziten Constraint hinweist ein Lösungsvorschlag erstellt. Die durch den Compiler mitgegebene, serialisierte Darstellung des impliziten Constraint wird dabei verwendet um ein Objekt vom Typ `ImplicitDomConstraint` zu erzeugen. Dieses liefert eine DOM AST Darstellung des Constraints. Mit Hilfe des API zur Manipulation des DOM AST (ASTRewrite) wird das Constraint dem existierenden Query angehängt, sofern der Nutzer dies wünscht.

Fehlermeldungen

Eine Syntax- und Typüberprüfung ist sinnlos, wenn der Aufrufer nicht über die dabei aufgedeckten Fehler informiert wird. Für diesen Zweck wurde das `GOTProblemReporterAdaptor` Team erstellt, welches alle für die Fehlerbehandlung notwendigen Klassen adaptiert. Dabei sind zwei Aufgaben zu erfüllen. Es müssen für GOT ungültige Fehlermeldungen heraus gefiltert und zusätzliche GOT-spezifische erzeugt werden. Alle Fehlermeldungen haben in der Basis des GOTProblemReporter, `ProblemReporter` eine eigene Methode, um sie auszulösen. Das Filtern erfolgt daher in dem

die entsprechenden Methoden adaptiert und mit Hilfe des übergebenen Kontextes des Fehlers überprüft werden. Ist die Quelle des Fehlers dabei ein GOT Element wie z. B. `GOTMethodSpec`, eine Metavariable im Basisteil einer Bindung, wird die Ausführung der Fehlermethode abgebrochen. Zusätzliche GOT Fehler werden mit Hilfe von Rollenmethoden im `GOTProblemReporter` erstellt, die dann auf allgemeingültige Methoden des ProblemReporter zugreifen, um den Fehler dem Tooling zur Verfügung zu stellen und dem Nutzer anzuzeigen.

5.5 Code Generation und DOM AST Erzeugung

Die letzten Phasen im Compilierungsprozess machen die Analyse (3.1) der aufgelösten ASTs, die Erzeugung von Bytecode (3.2) und die Erstellung des DOM AST (3.3) aus. Abbildung 5.1 zeigt diese Phasen im Compilierungsprozess. Während der Analyse Phase werden Überprüfungen vorgenommen, die während des Resolvings nicht möglich waren. Ein bekanntes Beispiel für eine Analyse in dieser Phase ist die Suche nach nicht verwendetem Code. Für Generic Object Teams können in dieser Phase komplexe implizite Constraints überprüft werden (siehe Abschnitt 5.4.3).

Die Erzeugung von Bytecode erfolgt in der nächsten Phase. In dieser Phase wird aus dem aufgelösten Compiler AST Bytecode erzeugt. Da Metavariablen aber erst in einem weiteren Source-to-Source Konvertierungsschritt aufgelöst werden, kann die Codeerzeugung nur für Elemente durchgeführt werden, die keine Metavariablen enthalten. Die Codeerzeugung wird für jede `CompilationUnitDeclaration` gestartet, die rekursiv ihren AST in Bytecode umwandelt. Diese `CompilationUnitDeclaration` wird im GOT Compiler AST adaptiert und die für die Codeerzeugung zuständige Methode `generateCode` mit einem replace Callin abgefangen. Die ursprüngliche Basismethode wird nur ausgeführt, sofern sie keine `GOTTeamDeclaration`, die auf Metavariablen schließen lässt, enthält. CompilationUnits, die ausschließlich Queryklassen enthalten werden, obwohl sie Metavariablen enthalten in Bytecode übersetzt. Dies ist dank der Konformität von Metatypen zu Javatypen möglich. Außerdem müssen Metavariablen in Queries innerhalb von Queryklassen nicht belegt werden. Die Übersetzung von Queryklassen in Bytecode ist notwendig, da das Resolving die Bytecode Repräsentation für die Auswertung von Imports verwendet.

5.5.1 Umwandlung des Compiler AST in den DOM AST

Die letzte Phase im Compilerprozess (Phase 3.3 in Abbildung 4.1.1) bildet die Umwandlung des Compiler AST in den DOM AST. Dieser DOM AST bildet das Ergebnis des GOT Compilers. Es ist die Basis für die Auswertung von Metavariablen und Transformation in OT Code durch spätere Transformer.

Für die Umwandlung des Compiler AST in den DOM AST ist der `ASTConverter` zuständig. Der Converter bietet für alle Compiler AST Elemente Methoden, die sie in DOM AST Elemente umwandeln. Sie sind zu erkennen durch das Präfix `convert`. Die für GOT wichtigsten `convert` Methoden wandeln Typen, Methoden, Blöcke und Expressions um. Alle diese Methoden werden mit Callins adaptiert. Im Compiler AST wurden einige Transformationen vorgenommen, um das Resolving leichter zu gestalten. Da das Resolving an diesem Punkt abgeschlossen ist, können dahingehend

vorgenommene Änderungen rückgängig gemacht werden. Die für die dafür notwendigen Schritte werden im Folgenden erläutert.

Konvertierung von match und declare Sprachelementen

Für das `match` und `declare` Element wurden einige strukturelle Veränderungen vorgenommen. So sind die Elemente nicht gesondert von den übrigen Body Declarations gelistet, wie es ihre Grammatik festlegt, sondern wurden im Compiler AST als Methoden im zugehörigen Team eingetragen. Zusätzlich wurden die im `match` und `declare` deklarierten Parameter in Felder umgewandelt und dem Team hinzugefügt.

Daher müssen vor dem Umwandeln einer `TypeDeclaration`, welche die Grundlage für eine `GOTTeamDeclaration` und damit den Container für `match` oder `declare` bietet, einige zusätzliche Operationen ausgeführt werden. Es müssen alle `match` oder `declare` Methoden aus der Liste der Body Declarations entfernt werden. Außerdem müssen alle Felder, die Metatyp Definitionen aus den Parameterlisten widerspiegeln, entfernt werden. Anschließend wird eine `GOTTeamDeclaration` im GOT DOM AST erstellt, die das entfernte `match` oder `declare` Element enthält und dieses mit Hilfe des generischen Property Mechanismus (siehe Abschnitt 4.2) im OT DOM AST speichert.

Konvertierung von per Blöcken

Für per Blöcke wurde ebenfalls eine geeignete Umstrukturierung vorgenommen, um die darin enthaltenen Elemente, möglichst ohne große Eingriffe in den für das Resolving verantwortlichen Code, aufzulösen. Alle Elemente innerhalb eines `per` Blocks wurden in das übergeordnete Element kopiert und dort aufgelöst. Diese Transformation muss sowohl für Knoten vom Typ `GOTPerClassBlock` als auch für Knoten des Typs `GOTPerMethodBlock` rückgängig gemacht werden. Es müssen folgende Schritte vorgenommen werden.

1. Alle Elemente, die vom per Block umschlossen werden, müssen aus dem umschließenden Element entfernt und in den per Block eingefügt werden.
2. Der per Block muss in die Liste der Bodydeclarations des umschließenden Elements eingefügt werden.

Diese Schritte müssen sowohl für per Blöcke innerhalb von Methoden als auch Klassen durchgeführt werden.

Konvertierung von markierten Knoten

Neben Knoten, die über Informationen aus der Basis direkt als Basen von GOT Knoten erkannt werden können, gibt es noch solche die explizit im GOT AST erstellt werden müssen, da ihre Basis keine Rückschlüsse erlaubt. Solche Knoten, wie zum Beispiel die `GOTQueryClass` müssen auch im GOT DOM AST erstellt und damit deren Basis markiert werden. Dafür werden entsprechende

OT Basisknoten bei der Konvertierung auf ihre eventuell vorhandenen Rollen im GOT Compiler AST überprüft und beim Vorhandensein solcher Rollen, entsprechende Knoten im GOT DOM AST erstellt.

6 Userinterface und Integration

Das Userinterface des GOTDT soll in seiner minimalen Form die Möglichkeit bieten GOT Projekte zu erstellen, Code zu erstellen, Fehler aufzuzeigen und bei Bedarf Lösungsvorschläge für Typfehler zu unterbreiten. Diese Funktionalitäten werden, dank des modularen Designs der Eclipse Plattform in ein eigenes Plugin namens `org.objectteams.gotdt.ui` eingebettet.

6.1 Projekte

Die Erstellung von GOT Projekten soll analog zu OT Projekten über einen Wizard erfolgen. Die Registrierung eines neuen Wizards mit dem Framework erfolgt über den Extension Point `org.eclipse.ui.newWizard`. Dieser bietet neben deklarativen Definitionen wie Name und Kategorie auch eine Klasse die vom Framework geladen wird, sofern der Wizard aufgerufen wird. Diese Klasse enthält die eigentliche Funktionalität des Wizards. Da der Wizard größtenteils dem des OTDT entspricht, um OT Projekte zu erstellen, wird als Superklasse für den Wizard die Klasse des OT Projekt Wizards verwendet.

Um GOT Projekte identifizieren zu können, wird eine GOT Nature erstellt. Diese Nature kann der Projektbeschreibung hinzugefügt werden. Die Nature gibt an, dass es sich bei dem Projekt um ein GOT Projekt handelt, in dem GOT Dateien enthalten sein können. Der Nature wird ein GOT Builder zugeordnet, der für die Übersetzung der GOT Dateien zuständig ist. Der Builder spezialisiert den OT Builder und wird ebenfalls über einen Extension Point, dem Framework bekannt gemacht. Beim Starten des Buildprozesses über die Eclipse Menus, wird der dem Projekt und der aktuellen Datei zugeordnete Builder aufgerufen. Der Builder sorgt dann für den Aufruf des Compilers mit allen vom User für die Compilierung ausgewählten CompilationUnits. Der GOT Projektwizard erstellt ein Projekt mit einer GOT Nature.

6.2 Editor mit Syntax Highlightning

Neben dem Compiler soll das GOTDT weitergehend Funktionalitäten besitzen, die den Nutzer beim Erstellen von Quelltext unterstützen. Das GOTDT bietet einen speziellen Editor zur Erstellung von Quelltext. Da GOT der Sprache OT sehr ähnlich ist, basiert der Editor auf dem OT Editor. Im Gegensatz zum Projektwizard wurde für das OTDT kein spezieller Editor über einen Extension Point registriert. Vielmehr wurde die Stärke von Object Teams verwendet, um den bisherigen Javaeditor mit einem Aspekt zu adaptieren und auf die Bedürfnisse von OT anzupassen.

An dieser Stelle entsteht ein neues Szenario. Im Gegensatz zu bisherigen Adaptionen liegt hier keine Javaklasse als Basis vor, sondern ein Team. Und dieses muss über Plugingrenzen adaptiert werden.

Das Team `JavaEditorAdaptor`, welches den Java Editor für OT adaptiert, wird mit Hilfe von Teamvererbung zum `GOTEditorAdaptor` spezialisiert. Dabei erbt das Team die Rollen seines Superteams. Je nach Bedarf können diese dann überschrieben werden. Beim Start des OTDT wird eine Instanz des `JavaEditorAdaptor` Teams erstellt. Wird nun beim Start des GOTDT eine Instanz des `GOTEditorAdaptor` erstellt, existieren zwei Teams im Tooling, welche die gleiche Funktionalität bieten. Es ist daher sinnvoll das `JavaEditorAdaptor` Team zu deaktivieren, sofern das GOTDT aktiviert wurde. Dafür wird über die globale Funktion `TransformerPlugin.getTeamInstances`, welche alle global aktivierten Teams in der Eclipse Plattform enthält, die Instanz des `JavaEditorAdaptor` Teams bestimmt. Mit Hilfe von dieser Instanz kann dann das überflüssige Team deaktiviert werden.

Der `JavaEditor` verwendet ein regelbasiertes System, um zu unterscheiden welche Zeichenkette im Editor hervorgehoben werden soll. Die für den Editor vorgesehenen Regeln werden beim Initialisieren des Editors in einer Liste zusammengetragen. Mit Hilfe einer darauf spezialisierten Rolle im `GOTJavaEditorAdaptor` wird diese Liste beim Initialisieren um GOT-spezifische Regeln ergänzt. Die somit erweiterte Regelmenge ermöglicht das Syntax Highlightning für GOT Keywords und Metavariablen.

6.3 Fehler und Lösungsvorschläge

Dank der Adaptierung des Javaeditors mit OT, bietet der resultierende GOT Editor sämtliche Funktionalitäten des Javaeditors. Darunter fallen auch Systeme zur Darstellungen von Fehlern und Warnungen. Alle Probleme, die durch den GOT Compiler festgestellt wurden, werden ohne notwendige Anpassungen im Editor angezeigt. Dazu zählen auch Probleme, die für GOT speziell sind; also auch Warnungen über die Verletzung von impliziten Constraints.

Um Lösungsvorschläge für Fehler und Warnungen bieten zu können, deklariert das JDT den Extension Point `org.eclipse.jdt.ui.quickFixProcessors`. Dieser ermöglicht die Definition von Quickfixes. Ein Quickfix ist ein Lösungsvorschlag für ein Problem, der eine (halb-)automatisierte Behebung dieses Problems beinhaltet. So kann in einem Quickfix der Quelltext des zugehörigen Editors über den AST verändert werden und somit der bestehende Fehler behoben werden, sofern dies der Wunsch des Benutzers ist. Für Generic Object Teams soll dieses System verwendet werden, um dem Nutzer die Möglichkeit zu bieten, implizite Constraints automatisch zu generieren und in bestehende Queries einpflegen zu lassen.

Dafür wird der beschriebene Extension Point verwendet, um einen speziellen Quickfix zu erstellen, der anhand der Problem Ids, für die einzelnen impliziten Constraints Lösungsvorschläge erstellt. Über die Problem Id wird das spezielle Constraint von der `ImplicitDomConstraintFactory` als DOM AST Expression erzeugt. Die so entstandene Expression wird dann, bei Bedarf, vom Quickfix in den AST eingetragen.

Die Struktur lässt ein einfaches Hinzufügen von weiteren impliziten Constraints zu.

6.4 Aktivierung

Das GOTDT hat prototypischen Charakter und ist daher weder performant, noch besonders robust. Es soll jedoch in der ausgereiften Entwicklungsumgebung Eclipse zusammen mit dem OTDT verwendet werden. Und dort nimmt das GOTDT einschneidende Änderungen am OT, bzw. Java Compiler vor. In Hinblick auf Performanz und Robustheit, kann es also nur von Interesse sein, die Veränderungen am OTDT durch das GOTDT gering und wenn möglich inaktiv zu halten, sofern diese nicht benötigt werden.

6.4.1 Teamaktivierung

Das GOTDT besteht aus einer Menge von Aspekten (Teams), die bestehende OTDT Klassen um Funktionalität für GOT erweitern. Eine Aktivierung bzw. Deaktivierung sollte also auf Basis dieser Teams stattfinden. Object Teams bietet die Möglichkeit Teams und damit alle darin enthaltenen Rollen und Callins zu (de-)aktivieren. Es werden dafür verschiedene Mechanismen zur Verfügung gestellt, von denen für diese Arbeit die explizite Teamaktivierung über die Methoden `Team.activate(Thread)` und `Team.deactivate(Thread)` am Wichtigsten ist. Diese Methoden erlauben eine Aktivierung auf Thread-Basis. Es kann also jedes Team für jeden Thread aktiviert oder deaktiviert werden.

Alternativ zur Verwendung der Teamaktivierung ist es möglich mit Hilfe von Guards an Rollen, oder Callins die Ausführung von Callins direkt am Ort des Geschehens einzuschränken. Dabei muss jedoch bei jedem Aufruf eines Callins eine zusätzliche Überprüfung durchgeführt werden, die Rechenzeit kostet. Außerdem ist es oft nicht möglich innerhalb einer Rolle oder eines Callins zu bestimmen, ob es sich hierbei um einen Teil einer Operation handelt, die innerhalb eines GOT Projekt ausgeführt wird.

Beide Mechanismen haben ihr Einsatzgebiet. Es wird daher eine Kombination beider verwendet. Sofern es möglich ist, sollten Guards an Callins oder Rollen eingesetzt werden, die die Ausführung der Callins auf GOT-relevante Operationen beschränken. Um jedoch unnötige Überprüfungen an teils hochfrequentierten Callins zu unterbinden, sollte wenn möglich, schon vorher erkannt werden, ob ein Aspekt (Team) in diesem Zustand des Programms aktiviert oder deaktiviert werden sollte.

Das GOTDT besteht aus einer Vielzahl von Teams, deren Aktivierung gesteuert werden muss. Tabelle 6.1 zeigt eine Auflistung dieser Teams. Dabei wird angegeben ob die Steuerung der Aktivierung für das jeweilige Team notwendig ist.

Es lassen sich zwei Kategorien von Teams unterscheiden. Solche Teams, die zu jedem Zeitpunkt aktiv sein müssen und solche, die erst aktiv werden müssen, sofern eine Operation, die sich auf ein Element innerhalb eines Projektes mit einer GOT Nature bezieht, ausgeführt wird. Das Parser Team muss zu jedem Zeitpunkt aktiv sein, da durch die Veränderung der Grammatik im

Team	Aktivierung
GOTCompiler	immer aktiv
GOTParserAdaptor	immer aktiv
GOTScannerAdaptor	bedingt
GOTLookupAdaptor	bedingt
GOTCompilerToDom	bedingt
GOTCodeassistAdaptor	bedingt
GOTProblemReporterAdaptor	bedingt
GOTCompilerASTRegistry	bedingt
GOTCompilerAST	bedingt
GOTUIActivator	immer aktiv
GOTJavaEditorAdaptor	bedingt

Tabelle 6.1: Die Tabelle zeigt die im GOTDT verwendeten Teams und ihren Aktivierungskontext. Ein Team muss entweder immer aktiv sein, oder während dem Compilieren einer CompilationUnit im Rahmen eines Projektes mit einer GOT Nature.

`org.eclipse.jdt.core` Plugin, selbst die Kompilierung von Java Dateien den Einsatz des GOT Teams erfordert.

Die Aktivierung der Teams wird von zwei Klassen gesteuert. Dazu zählen im `org.objectteams.gotdt` Plugin das Team `GOTCompiler`, welches die Aktivierung aller für den Compiler relevanten Teams steuert. Und im `org.objectteams.gotdt.ui` Plugin das `GOTUIActivator` Team, welches die Aktivierung von UI Teams *und* Compiler Teams steuert. Die Aktivierungsteams besitzen die Instanzen aller ihnen zugeordneten Teams. Über diese ist es ihnen möglich bei Bedarf die einzelnen Teams zu aktivieren oder deaktivieren. Die Entscheidung über die Aktivierung wird mit Hilfe von speziellen Rollen innerhalb der Aktivierungsteams getroffen. Diese Rollen werden im Folgenden Activationhooks genannt.

6.4.2 Activationhooks

Activationhooks werden benötigt, da eine Vielzahl verschiedener Eintrittsstellen in den Compilercode existieren, an denen entschieden werden muss, ob der GOT Compiler aktiviert werden muss. Außerdem werden im OTDT viele verschiedene Threads verwendet, um die Kompilierung durchzuführen. Es wäre also denkbar, dass in dem einen Thread ein Compilierungsprozess eines Java Projektes und in einem anderen die Kompilierung eines GOT Projektes stattfindet. Dabei soll der GOT Compiler nur für das GOT Projekt aktiv werden. Eine Aktivierung pro Thread ist also ebenfalls von Nöten. Activationhooks haben mindestens ein replace Callin auf eine Stelle im Tooling, an der ein Aufruf an den Compiler stattfindet und das zugehörige Projekt bestimmt werden kann, um zu entscheiden ob eine Aktivierung durchgeführt werden muss. Diese Stelle muss so im Kontrollfluss gewählt werden, dass nach dem Ausführen des Basecalls innerhalb des replace Callins, eine Deaktivierung des GOT Compilers erfolgen kann. Es ist möglich, dass Activationhooks überlappen, also der GOT Compiler bereits von einem höher im Kontrollfluss liegenden Hook aktiviert wurde. Um eine frühzeitige Deaktivierung des Compilers zu verhindern, werden Semaphore artige

```
protected class SomeActivationHook extends GOTActivationHook playedBy
    ASTParser
{
  void someHook(String source) <- replace void someMethod(String source);

  callin void someHook(String source) {
    if(isGOTProject()) {
      activate(Thread.currentThread());
      base.someHook(source);
      deactivate(Thread.currentThread());
    }
    else
      base.someHook(source);
  }
}
```

Listing 6.1: Das Listing zeigt ein Muster eines Activationhooks. Ein solcher Hook wird an Eintrittsstellen in den Compiler verwendet. Abhängig vom aktuellen Projekt dieses Aufrufes, werden die für den GOT Compiler verantwortlichen Teams aktiviert.

Zähler verwendet, die pro Thread und Team den Aktivierungzustand überwachen. Die Struktur eines Activationhooks zeigt Listing 6.1.

Um die Funktionsweise von Activationhooks zu verdeutlichen, wird ein Beispiel eines Hooks betrachtet. Die Methode `ASTParser.internalCreateAST(IProgressMonitor monitor)` wird verwendet, um programmatisch eine Quelldatei in einen DOM AST umzuwandeln. Für diese Umwandlung wird der komplette Compiler samt Parser, Resolving und Konverter verwendet. Der Aufruf der Methode `internalCreateAST` stellt den Anfang der Kompilierung dar. Vor dem Aufruf des Basecalls werden alle Compiler Teams, falls es sich bei dem zugehörigen Projekt um ein GOT Projekt handelt, aktiviert. Die Aktivierung erfolgt für den aktuellen Thread, aus dem heraus die Methode aufgerufen wurde. Nach dem Ausführen der Methode ist die Kompilierung beendet und der GOT Compiler kann für den aktuellen Thread deaktiviert werden.

Die Aktivierung von Teams mit Hilfe von Activationhooks hat einige Vorteile. So kann ein unnötiges Einschreiten von GOTDT Funktionen vermieden werden, sofern es nicht explizit in Form einer GOT Nature im zugehörigen Projekt erwünscht ist. Außerdem wird die Funktionalität des OT Compilers weitestgehend von Fehlern im prototypischen GOTDT befreit. Der Nachteil ist die aufwendige Bestimmung von geeigneten Stellen für einen Activationhook. Dies resultiert aus dem komplexen Aufbau des Toolings und der vielseitigen Eintrittspunkte in den Compiler.

6.4.3 UI Activationhooks

Einige Eintrittsstellen in den Compilercode lassen sich nur sicher im UI Plugin bestimmen. Dies ist darin begründet, dass sich erst dort im Kontrollfluss bestimmen lässt, zu welchem Projekt der Aufruf gehört. Daher werden für Strukturen wie den Reconciler, die im UI Plugin definiert sind, die Activationhooks im `GOTUIActivator` Team realisiert. Von dort wird dann das `GOTCompiler` Team über Aktivierungen benachrichtigt.

Die Aktivierung der UI Teams wurde im aktuellen Prototyp vernachlässigt, da weder Performanz der Anpassungen, noch die Gefahr von Fehlern ins Gewicht fallen.

7 Fazit und Ausblick

Die Sprache Generic Object Teams bietet eine nützliche Erweiterungen zu Object Teams. Bei der Entwicklung des GOTDT sind einige Einsatzgebiete für GOT aufgefallen. So wäre beispielsweise eine generische Definition von Activationhooks oder eine generisches Resolving der verschiedenen Darstellungen von Metavariablen denkbar.

Die Entwicklung der Werkzeugunterstützung führte mit Hilfe von Object Teams teils mit sehr geringen Anpassungen zu erstaunlichen Ergebnissen. Besonders hervorzuheben ist dabei die Adaption der ASTs. Dank der Möglichkeit, die OT ASTs in Takt zu lassen und notwendige Änderungen in Aspekte auszulagern, konnte ein Großteil der Compilierung bereits vom OTDT übernommen werden. Beispielsweise mussten die Kontrollflüsse innerhalb des sehr komplexen JDT (OTDT) nicht im Detail analysiert werden, um den Ort der Erzeugung eines AST Knotens für einen einfachen Namen (`SimpleName`) aufzufinden. Es konnte eine Rolle um den Knoten gelegt werden, die das Verhalten des Knotens in das eines GOT Knotens verändert.

Zu dem ermöglichte der Einsatz von Object Teams die Erweiterungen, die für Generic Object Teams vorgenommen werden mussten, in Module zu bündeln, die bei Bedarf aktiviert oder deaktiviert werden können. Dadurch konnte die Evolutionsfähigkeit verbessert werden, so dass Veränderungen am JDT und OTDT leichter in das GOTDT einfließen können. Zu dem kann dadurch sichergestellt werden, dass das GOTDT bei der Compilierung von reinem OT oder Java Code deaktiviert ist. So beeinflussen Fehler im GOTDT nicht das OTDT. Das Auffinden von Auslösern, um zu entscheiden ob das GOTDT aktiviert oder deaktiviert werden muss, war jedoch keine leichte Aufgabe. Die Komplexität und das stark vernetzte Design von Compiler und Userinterface erforderten ausgiebige empirische Versuche, um die wichtigsten Punkte zur Aktivierung des GOTDT zu finden.

Eine Beobachtung hat sich in der Entwicklung der Werkzeugunterstützung klar abgezeichnet. Programmteile, die auf eine Veränderung ausgelegt und anerkannte Design Patterns verwenden, sind mit Object Teams um vieles einfacherer zu adaptieren. Je weiter sich die Entwicklung vom Parser in Richtung Userinterface und Eclipse Framework verlegt hat, desto schneller konnten Erfolge erzielt werden. Dabei war der am schwersten zu adaptierende Teil der Parser. Der Parser wurde primär auf Performanz ausgelegt. Dafür wurden an einigen Stellen Methoden gegen schnell auszuwertende Arrays und Switchstatements eingetauscht. Für die Performanz zwar von Vorteil, führten diese zu einem für die Wiederverwendung fatalen Schritt. Es musste ein komplettes Plugin mit hunderten Dateien ausgetauscht werden, um ein einziges 20 Zeilen langes Interface auszuwechseln. Positiv im Hinblick auf die Wiederverwendung war dagegen der DOM AST, am anderen Ende des Compilerprozesses, der dank seines generischen Designs in seiner Struktur für Generic Object Teams Knoten angepasst werden konnte. Dadurch besteht selbst für Programme, die kein Wissen vom GOTDT haben, die Möglichkeit den GOT AST auszulesen und weiter zu verwenden.

Alle für die Arbeit angesetzten Ziele konnten erreicht werden. Es wurde eine Syntax gefunden, die sowohl die Definition von Generic Object Teams Queries als auch die Integration von Metavariablen in OT/J Code ermöglicht. Größere Probleme traten erst in der Integration der Sprache in das OTDT Tooling auf. Dabei war die Integration von neuen strukturellen Sprachelementen wie dem `per` Block besonders problematisch. So konnte kein äquivalentes OT Sprachelement gefunden werden, welches analog zu einem `per` Block sowohl innerhalb von Klassen als auch innerhalb von Methoden zum Bündeln von Elementen verwendet werden kann. Die Adaption des Parsers gestaltete sich weitaus schwieriger als Anfangs angenommen. Da die entscheidende Logik des Parsers mit Hilfe eines Generators erzeugt wurde und ohne diesen nicht nachvollziehbar und manipulierbar war musste die gesamte Logik des Parsers neu generiert werden. Dadurch folgten Anpassungen im Kern Modul des OTDT, welche eine ausgiebige Textkopie mit sich zogen. Die Anpassung der OT Grammatik für GOT, die dem Parsergenerator als Grundlage dient, erforderte eine Vielzahl von Workarounds, um die LALR(1) Eigenschaft der Grammatik einzuhalten. Erfreulicherweise konnten über die Repräsentation von GOT Sprachelementen als Java Elemente und einigen Transformationen im Parser, ein Großteil der Funktionalität des OTDT für Resolving und Typechecking wiederverwendet werden. Die Erkennung und Integration eines GOT spezifischen, impliziten Constraints konnte dank der umfangreichen Navigationsmöglichkeiten im AST und des ausgereiften GUI Frameworks, leicht in das Tooling integriert werden. Ebenso von Vorteil war die generische Struktur des DOM AST, der als Schnittstelle für aufbauende Arbeiten zur Generierung von OT Code aus GOT ASTs dient. Hierbei können alle relevanten Informationen des GOT AST über bekannte JDT Schnittstellen ausgelesen werden.

Obwohl Generic Object Teams als Erweiterung von Object Teams nur eine geringe Menge an zusätzlichen Elementen aufweist, war die Integration in das OTDT keineswegs trivial. Es wurden an die 10000 Zeilen Code benötigt, um die grundsätzlichen Sprachelemente in das Tooling zu integrieren. Dieser Code ist nahezu vollständig Aspektcode, der eine genaue Auseinandersetzung mit dem vorliegenden Basiscode und Konzepten wie Parallelität und Aktivierung erfordert. Der Basiscode war größtenteils undokumentierter, unkommentierter, auf Geschwindigkeit optimierter Code, der über die Jahre auf die Bedürfnisse des JDT und neuen Java und OT Versionen angepasst wurde. Er ist nicht für eine Adaption ausgelegt. Alleine das Verständnis der Struktur und Abläufe im OTDT hat wochenlanges Nachvollziehen des Kontrollflusses über den Debugger gefordert. Das OTDT ist ein komplexes Tooling, mit einem großen Funktionsumfang. Dieser Umfang und die fehlende Auslegung als Grundlage für Adaptionen, gestaltet die Entwicklung von aufbauender Software sehr schwer. Für die Entwicklung eines einfachen Prototyps eines Compilers sollte also eher auf einfachere Basissoftware zurück gegriffen werden, die nicht nur auf Geschwindigkeit, sondern auf Adaptionsfähigkeit und Verständlichkeit ausgelegt ist.

Die Verwendung von Object Teams zur Adaption des OTDT für GOT hat im Gegensatz zum Branchen und der direkten Veränderung der Quellen, den Programmieraufwand erheblich gesteigert. Entstanden ist dadurch jedoch ein Prototyp des Toolings für GOT, der der Evolution des JDT und OTDT ohne große Veränderungen am Code folgen kann. Um einen möglichst robusten Evolutionsprozess zu gewährleisten, sollte jedoch in aufbauenden Projekten eine umfangreiche Testsuite aufgebaut werden, die die einwandfreie Funktion nach einem Evolutionsschritt sicherstellen kann.

7.1 Ausblick

Das GOTDT in seiner aktuellen Form bietet noch viele Punkte, an denen Verbesserungen vorgenommen werden können. Im Vordergrund sollte dabei die Erweiterung des Recovery- und Diagnoseparsers stehen, um die Entwicklung von GOT Quellcode robuster zu machen. Es gibt noch eine Reihe von Quellcode Konstellationen, bei denen der Parser den Nutzer mit schwer nachvollziehbaren Fehlern konfrontiert. In weiteren Schritten, außerhalb des Parsers, sollten zusätzliche Quickfixes für implizite Constraints implementiert werden. Ein Endziel wäre die Erreichung der vollen Funktionalität des OTDT.

Bisher bietet das GOTDT noch keine Möglichkeit, um aus GOT Quellcode letztendlich OT Quellcode zu erzeugen. Daher muss in späteren Schritten die Integration mit den zu OT Code transformierenden Komponenten erfolgen. Denkbar wäre darauf folgend auch eine direkte Transformation im Compiler. Außerdem könnten Metavariablen schon während des Typecheckings über die Faktenbasis aufgelöst werden. Dies würde eine bessere Toolunterstützung und Früherkennung von Fehlern ermöglichen.

Der in dieser Arbeit verwendete Ansatz zur Adaption des OTDT für die Sprache Generic Object Teams, durch den Einsatz von Aspekten, könnte auch für das Bootstrapping des OTDT von Interesse werden. Hilfreich könnten dabei, die im Zuge der Entwicklung von Eclipse e4 angedachten Änderungen, am Parser und Compiler sein (siehe Eclipse Foundation (2011a)).

Literaturverzeichnis

[Eclipse Equinox 2011] ECLIPSE EQUINOX: *Eclipse Equinox Plugin System*. Webseite. 02 2011. – URL `http://www.eclipse.org/equinox/`. – Zuletzt besucht: 15.02.2011

[Eclipse Foundation 2011a] ECLIPSE FOUNDATION: *Bug 36939 - Improve support for Java-like source files*. Webseite. 02 2011. – URL `https://bugs.eclipse.org/bugs/show_bug.cgi?id=36939`. – Zuletzt besucht: 19.02.2011

[Eclipse Foundation 2011b] ECLIPSE FOUNDATION: *How to: Generate the Parser*. Webseite. 02 2011. – URL `http://www.eclipse.org/jdt/core/howto/generate%20parser/generateParser.html`. – Zuletzt besucht: 21.02.2011

[Eclipse JDT 2011] ECLIPSE JDT: *Eclipse Java development tools*. Webseite. 02 2011. – URL `http://www.eclipse.org/jdt/`. – Zuletzt besucht: 19.02.2011

[Eclipse RCP 2011] ECLIPSE RCP: *Rich Client Platform*. Webseite. 02 2011. – URL `http://www.eclipse.org/home/categories/rcp.php`. – Zuletzt besucht: 19.02.2011

[Havinga 2005] HAVINGA, Wilke: Designating join points in Composestar - a predicate-based superimposition selector language for Compose*. (2005)

[Herrmann 2003] HERRMANN, Stephan: Object Teams: Improving Modularity for Crosscutting Collaborations. In: *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. London, UK : Springer-Verlag, 2003, S. 248–264. – ISBN 3-540-00737-7

[Herrmann und Mosconi 2007] HERRMANN, Stephan ; MOSCONI, Marco: Integrating Object Teams and OSGi: Joint Efforts for Superior Modularity. In: *Journal of Object Technology* 6 (2007), Nr. 9, S. 105–125

[Kniesel und Rho 2006] KNIESEL, Günter ; RHO, Tobias: A Definition, Overview and Taxonomy of Generic Aspect Languages. In: *L'Objet* 11 (2006), Nr. 3. – URL `http://roots.iai.uni-bonn.de/research/logicaj/downloads/papers/knieselRho-lObjet2006.pdf`. – Zuletzt besucht: 31.03.2011

[LogicAj 2011] LOGICAJ: *LogicAj - Logic Aspects for Java*. Webseite. 02 2011. – URL `http://roots.iai.uni-bonn.de/research/logicaj/`. – Zuletzt besucht: 18.02.2011

[Mertgen 2007] MERTGEN, Andreas: *Modellbasierte Integration von Joinpoint Queries für die aspektorientierte Sprache ObjectTeams/Java*, Technische Unversität Berlin, Diplomarbeit, 2007

[Mertgen 2010] MERTGEN, Andreas: *Generic Object Teams - DRAFT*. 09 2010

[Object Teams 2011a] OBJECT TEAMS: *Object Teams - Programming with Roles and beyond*. Webseite. 02 2011. – URL http://www.eclipse.org/objectteams/. – Zuletzt besucht: 18.02.2011

[Object Teams 2011b] OBJECT TEAMS: *Object Teams Language Definition - Declared lifting*. Webseite. 02 2011. – URL http://www.objectteams.org/def/1.3/s2.html#s2.3.2. – Zuletzt besucht: 19.02.2011

[Object Teams 2011c] OBJECT TEAMS: *Object Teams Language Definition - Explicit role creation*. Webseite. 02 2011. – URL http://www.objectteams.org/def/1.3/s2.html#s2.4. – Zuletzt besucht: 19.02.2011

[OSGi Alliance 2011] OSGI ALLIANCE: *OSGi*. Webseite. 02 2011. – URL http://www.osgi.org. – Zuletzt besucht: 19.02.2011

[Philippe Charles 2011] PHILIPPE CHARLES: *LPG - LALR Parser Generator*. Webseite. 02 2011. – URL http://sourceforge.net/projects/lpg/. – Zuletzt besucht: 20.02.2011

[Rho u. a. 2006] RHO, Tobias ; KNIESEL, Günter ; APPELTAUER, Malte: Fine-grained Generic Aspects, Workshop on Foundations of Aspect-Oriented Languages (FOAL'06), AOSD 2006. Workshop on Foundations of Aspect-Oriented Languages (FOAL'06), in conjunction with Fifth International Conference on Aspect-Oriented Software Development (AOSD.06), March 20-24, 2006, Bonn, Germany, Mar 2006. – URL http://roots.iai.uni-bonn.de/research/logicaj/downloads/papers/RhoKnieselAppeltauer-FOAL06.pdf. – Zuletzt besucht: 31.03.2011

[Stephan Herrmann 2010a] STEPHAN HERRMANN: *The Object Teams Blog - IDE for your own language embedded in Java? (part 1)*. Webseite. 02 2010. – URL http://blog.objectteams.org/2010/02/ide-for-your-own-language-embedded-in-java-part-1/. – Zuletzt besucht: 19.02.2011

[Stephan Herrmann 2010b] STEPHAN HERRMANN: *The Object Teams Blog - IDE for your own language embedded in Java? (part 2)*. Webseite. 02 2010. – URL http://blog.objectteams.org/2010/02/ide-for-your-own-language-embedded-in-java-part-2/. – Zuletzt besucht: 19.02.2011

[Witte 2003] WITTE, Markus: *Portierung, Erweiterung und Integration des ObjectTeams/Java Compilers für die Entwicklungsumgebung Eclipse*, Technische Unversität Berlin, Diplomarbeit, 2003

Abkürzungsverzeichnis

OOP objektorientierte Programmierung
SWT Standard Widget Toolkit
RCP Rich Client Platform
LOC Lines of Code
JVM Java Virtual Machine
OT Object Teams/Java
OTDT Object Teams Development Tooling
GOT Generic Object Teams
GOTDT Generic Object Teams Tooling
JDT Java Development Tooling
AST Abstract Syntax Tree
DAST Dom Abstract Syntax Tree
CAST Compiler Abstract Syntax Tree
DOM Document Object Model
API Application Interface
UI User Interface
LPG LALR Parser Generator