Rapid development of object oriented semantic grammars

Ein Eclipse Editor für JSGF-basierte Grammatiken

Student: Marek Wester
Betreuer: Dipl.-Inform. Hartwig Holzapfel, Dr.-Ing. Thomas Schaaf

24.08.2005

Abstract

Für die Entwicklung von Grammatiken für Dialog-Management-Systeme im JSGF-Format gab es bisher keine Entwicklungsumgebung, die eine komfortable und schnelle Entwicklung unterstützt. Das JSGF-Grammatik-Format ist ein kontextfreies Grammatik-Format, welches die gleiche Syntax nutzt, wie das Grammtik-Format des Tapas-Dialog-Managers. Zwar gleichen sich diese Formate syntaktisch, allerdings ist das Tapas-Format durch den Einsatz objekt-orientierter Techniken ausdrucksmächtiger.

Für beide Grammatik-Formate wurde in der Studienarbeit ein Editor als eine Reihe von Eclipse-Plugins realisiert. Durch seinen modularen Aufbau ist der Editor leicht erweiterbar. Die Entwicklung der Grammatiken wird durch den Editor mit den neuesten Entwicklungstechnologien wie Syntax Highlighting, automatischer Fehlererkennung/-korrektur und weiteren mehr unterstützt.

Abstract english

Until now there was no development environment that supports the comfortable and fast development of JSGF grammars for dialog management systems. The JSGF-grammar format is a context free grammar format which uses the same syntax as the grammar format for the Tapas dialog manager. Indeed both formats are syntactically equivalent. However the Tapas format through the usage of object oriented technics is more expressive.

For this short term thesis, an editor was developed for both of the grammar formats as a set of Eclipse plugins. Its modular structure allows it to be easily extensible. The editor supports the grammar development with the newest development technologies like syntax hightlighting, automatic error detection and error correction and much more.

Danksagungen

Bedanken möchte ich mich bei Professor Waibel, für die Schaffung des interACT-Programmes, welches mir den Aufenthalt an der Carnegie Mellon University ermöglicht hat.

Bedanken möchte ich mich bei meinen beiden Betreuern Thomas Schaaf und Hartwig Holzapfel, die mich in allen meinen Belangen und Nöten in den USA unterstützt haben. Sie nahmen sich viel Zeit und waren immer sehr bemüht, alles für mich zu tun.

Weiterhin möchte ich mich bei meiner Beta-Testerin Patrycja Holzapfel bedanken, die mich immer mit den neuesten Fehlern versorgt hat.

Großer Dank geht auch an meine Freundin Svenja Albrecht für das Lesen und Korrigieren meiner Studienarbeit und für ihre Unterstützung während meines Aufenthaltes in den USA. Dafür auch Dank an meine Eltern.

Inhaltsverzeichnis

1 Einleitung 1
1.1 Motivation . . . 1
1.2 Die Wahl des Frameworks . . . 2
1.3 Die Gliederung der Studienarbeit . . . 4

2 Grundlagen 5
2.1 Grammatiken . . . 5
2.1.1 Java Speech Grammar Format . . . 6
2.1.2 Tapas . . . 7
2.2 Eclipse . . . 8
2.2.1 Das Framework . . . 9
2.2.2 Plugin . . . 9
2.2.3 Sichten, Perspektiven, Editoren . . . 10
2.2.4 Fazit . . . 11

3 Anforderungen und Umsetzung 12
3.1 Architektur . . . 13
3.2 Parser-Plugin . . . 14
3.3 Syntax-Highlighting . . . 17
3.3.1 Wege in Eclipse . . . 17
3.3.2 Konkrete Umsetzung . . . 18
3.4 Outline . . . 19
3.4.1 Wege in Eclipse . . . 20
3.4.2 Konkrete Umsetzung . . . 20
3.5 Sprung zur Definition . . . 20
3.5.1 Wege in Eclipse . . . 21
3.5.2 Konkrete Umsetzung . . . 21
3.6 Anzeigen der Definition . . . 22
3.6.1 Wege in Eclipse . . . 22
3.6.2 Konkrete Umsetzung . . . 22
3.7 Aufgabenliste . . . 22
3.7.1 Wege in Eclipse . . . 24
3.7.2 Konkrete Umsetzung . . . 24
3.8 Textvervollständigung . . . 25
3.8.1 Wege in Eclipse . . . 27

3.8.2 Konkrete Umsetzung 27
3.9 Automatische Fehlererkennung 28
3.9.1 Wege in Eclipse 28
3.9.2 Konkrete Umsetzung 30
3.10 Automatische Fehlerbereinigung 31
3.10.1 Wege in Eclipse 32
3.10.2 Konkrete Umsetzung 32

4 Zusammenfassung und Ausblick **35**
4.1 Zusammenfassung 35
4.2 Ausblick 36

A Abkürzungen **37**

B Handbuch **38**
B.1 Installation 38
B.2 Anlegen eines JSGF-Projektes 38
B.3 Anlegen eines Tapas-Projektes 39
B.4 Funktionen 40
B.5 Tastenbelegungsübersicht 42

Literaturverzeichnis **43**

Abbildungsverzeichnis

3.1 UML-Diagramm der Architektur der Plugins 14
3.2 UML-Diagramm Parser 15
3.3 Syntax-Highlighting im JSGF-Editor 17
3.4 Outline im JSGF-Editor 19
3.5 Anzeige von Definitionen 22
3.6 Aufgaben Liste im JSGF-Editor 23
3.7 Ein Beispiel für die Textvervollständigung 26
3.8 Erkannte Fehler 29
3.9 UML-Diagramm Fehler 31
3.10 UML-Diagramm der Fehlerbereinigungsarchitektur 33

B.1 Syntax-Highlighting, Outline, Aufgabenliste, automatische Fehlererkennung / -beseitigung, Fehlerübersicht 39
B.2 Konfiguration der Schlüsselwörter für die Aufgabenübersicht und der Farben für das Syntax-Highlighting 41

Tabellenverzeichnis

3.1 Fehlerarten/-lösungen 34

B.1 Tastenbelegungsübersicht 42

Kapitel 1

Einleitung

In dieser Studienarbeit wurden ein Editor für die Entwicklungsumgebung Eclipse als eine Reihe von Plugins entwickelt, welcher die Entwicklung von JSGF- und Tapas-Grammatiken erleichtern und beschleunigen soll.

1.1 Motivation

Die Entwicklung von Grammatiken für Dialog-Management-Systeme verläuft in der heutigen Zeit oft mit Texteditoren ohne Funktionalitäten, welche die Entwicklung unterstützen. Dies bedeutet, dass eine Grammatik von einem Benutzer korrekt in einen Editor eingeben werden muss. Denn Fehler werden erst bei der Verarbeitung durch das Dialog-Management-System gemeldet. Weiterhin ist die Pflege bestehender Grammatiken sehr schwer, da herkömmliche Texteditoren keine Möglichkeiten bieten, Grammatiken übersichtlich darzustellen. Die Gewinnung der Übersicht kann sehr schwer fallen, da jeder Buchstabe gleich dargestellt wird und beispielsweise Schlüsselwörter nicht markiert werden.

Es ist möglich, dass bei der Grammatikentwicklung vergessen wird, welche Nichtterminale definiert wurden und dadurch Mehrfachdefinitionen in der Grammatik auftauchen. Grammatiken mit einer großen Anzahl von Nichtterminaldefinitionen können das Behalten der Nichtterminale durch den Entwickler erschweren. Dies wird weiterhin verschärft, wenn der Entwickler eine ihm unbekannte Grammatik bearbeitet.

Eine einfache Suche nach der Definition bietet keine Lösung für dieses Problem, da ein Nichtterminal zwar nur einmal definiert wird, aber mehrfach verwendet werden kann.

Die Entwicklung von Grammatiken ist ein iterativer Prozess, der sich über mehrere Tage erstreckt. Grammatiken werden zur Strukturierung über mehrere Dateien verteilt. Innerhalb des Entwicklungsprozesses verbleiben Teile der Grammatik bis zum nächsten Iterationsschritt in einem unvollständigen Zustand. Es ist nicht leicht die Übersicht über die Vervollständigung

der unfertigen Teile zu behalten. Die Rückkehr an die unfertigen Stellen wird nicht durch die vorhandenen Werkzeuge unterstützt. Die Suche nach diesen Stellen kosten den Entwickler Zeit, in der er nicht produktiv arbeiten kann.

Die vollständige Eingabe langer Nichtterminale stellt einen Tippaufwand dar, der dem Entwickler abgenommen werden kann. Selbst moderne Textverarbeitungssysteme bieten die Möglichkeit der Vervollständigung von Wörtern. Diese Vervollständigung kann im Kontext der Grammatikentwicklung zielführender geschehen. Die Kontexte, in denen sich der Cursor befindet, könnten zur Erzeugung von Vervollständigungsvorschlägen mit einbezogen werden.

Rapid Application Development ist zum Schlagwort im Bereich der Softwareentwicklung geworden. Heute wird weniger Zeit gebraucht, um komplexere Software zu entwickeln, als dies noch am Ende des letzten Jahrhunderts der Fall war. Moderne Entwicklungsumgebungen und große Bibliotheken leisten einen großen Teil der Entwicklungsarbeit. Da Entwickler von JSGF- und Tapas-Grammatiken noch nicht von den heutigen Möglichkeiten profitieren können, soll in dieser Studienarbeit ein Editor entstehen, der die aufgezählten Schwächen der von Grammatikentwicklern genutzten Werkzeuge für diese Grammatiken beseitigen soll.

1.2 Die Wahl des Frameworks

Ein *Framework* ist ein Klassengerüst, welches eine Anwendungsarchitektur vorgibt. Dieses Gerüst kann durch eigene Klassen erweitern werden. Zur Entwicklung eines Programms müssen diese Klassen an dem Gerüst registriert werden. Das Framework kümmert sich um die Instantiierung der Klassen und das Aufrufen der Methoden, welche die Klassen definieren. Das Vorgehen des Frameworks wird als Hollywood-Prinzip bezeichnet: "Don't call us - we'll call you".

Durch die gegebene Anwendungsarchitektur des Frameworks konnte die Entwicklungsarbeit vollkommen auf die gestellte Aufgabe fokussiert werden und wurde nicht in redundante Arbeit investiert. Die Erweiterbarkeit eines Frameworks und der darauf aufbauenden Komponenten sollte eine wichtige Eigenschaft bei der Entwicklung werden.

Bei der Wahl des Frameworks kam es also darauf an, dass dieses Funktionen bietet, welche den Editor für die schnelle Entwicklung von Grammatiken aufwerten können. Da sowohl JSGF als auch Tapas vor allem im Java-Umfeld genutzt werden, wäre es wünschenswert, wenn mit Hilfe des Frameworks auch eine Java-Entwicklungsumgebung implementiert wäre. Die Quelloffenheit des Frameworks kann die Entwicklung erleichtern, weil so Einblick in die Umsetzung der vom Framework gebotenen Konzepte und Funktionen möglich ist. Weiterhin bieten die durch das Framework exemplarisch implementierten Entwicklungsumgebungen eine reiche Quelle an Beispielen. Da diese Studienarbeit im universitären Umfeld angesiedelt

ist, spielen auch Anschaffungskosten eine große Rolle. Weiterhin sollte das Ergebnis der Studienarbeit nach seiner Fertigstellung weiterentwickelt und genutzt werden können. Dies zu gewährleisten, kann durch den Einsatz eines Frameworks versucht werden, welches von einer großen Zahl von Entwicklern genutzt wird.

Bei Berücksichtigung dieser Kriterien bleiben zwei Kandidaten: Eclipse und Netbeans.

Netbeans[ ] ist ein Framework für die Entwicklung von Entwicklungsumgebungen. Es ist quelloffen, kostenlos, bietet eine Java-Entwicklungsumgebung und ist in Java implementiert. Der Editor dieses Frameworks bietet als herausragende Funktionen Syntax Highlighting, Textvervollständigung, Darstellung von Fehlern während der Entwicklung und automatische Fehlerkorrektur. Leider ist die Benutzeroberfläche mit der Swing-Bibliothek implementiert, weswegen die ganze Entwicklungsumgebung sich in ihrem Aussehen von den Plattformen, auf denen sie läuft, äußerlich unterscheidet. Auch die Bedienung grafischer Standard-Bedienelemente und -Dialoge ist nicht immer konform zu der Bedienung, wie sie sonst von der jeweiligen Plattform gegeben ist. Die subjektiv empfundene Geschwindigkeit beim Entwickeln mit Netbeans macht ein Arbeiten schwierig, da während der Eingabe von Programmtexten oft auf die Entwicklungsumgebung gewartet werden muss.

Eclipse[ ] ist als Framework für die Entwicklung von Entwicklungsumgebungen entwickelt worden. Die gebotene Funktionalität des Editormoduls ist mit der des Editormoduls von Netbeans vergleichbar.

Da in Eclipse nicht die langsamere Swing-Bibliothek für die Anzeige der Bedienelemente benutzt wird, sondern die SWT-Bibliothek, ist diese Umgebung trotz Java-Unterbaus subjektiv so schnell wie eine C++ Anwendung.

Die erste Entwicklungsumgebung, welche mit Hilfe des Eclipse-Frameworks entwickelt worden ist, war die Entwicklungsumgebung für Java (JDT - Java Development Tools).

Mit der Entwicklung des JDT ist eine erfolgreiche Entwicklungsumgebung entstanden, welche sich durch messbar und spürbar hohe Geschwindigkeit auszeichnet. Sie bietet Funktionen, welche in dieser Weise von keiner anderen Entwicklungsumgebung bereitgestellt werden. Besonders beschleunigen die Funktionen im Bereich des Refactoring und der Fehlererkennung und automatischen Fehlerbeseitigung die Entwicklung. Die Implementierung des JDT zeigt, welche Möglichkeiten das Eclipse-Framework bezüglich der Entwicklung eines Editors bietet. Es kann als Referenzimplementierung angesehen werden.

Aus den genannten Gründen und aufgrund der größeren Beliebtheit von Eclipse in der Gemeinschaft der Entwickler (gemessen an der Verfügbarkeit von Erweiterungen) ist Eclipse für diese Studienarbeit gewählt worden.

1.3 Die Gliederung der Studienarbeit

Zuerst wird in Kapitel 2 ein Überblick über die zu editierenden Grammatiken gegeben. Diesem folgt eine Darstellung der für das Verständnis der Arbeit wichtigen Komponenten des Eclipse-Frameworks. Kapitel 3 beschreibt die einzelnen implementierten Funktionen und ihre Umsetzung.

Die Beschreibung der einzelnen Funktionen ist jeweils unterteilt in drei Teile. Zuerst wird die Motivation für eine Funktion und die Funktion selber beschrieben. Im zweiten Teil soll gezeigt werden, welche Möglichkeiten es für die Umsetzung der Funktion im Eclipse-Framework gibt. Der dritte Teil enthält schließlich die konkrete Umsetzung und begründet die getroffenen Entwurfsentscheidungen.

In Kapitel 4 findet sich eine Zusammenfassung der Studienarbeit, der ein Ausblick auf Arbeiten folgt, die den Editor weiter verbessern könnten. Im Anhang findet sich eine Übersicht über die benutzten Abkürzungen und ein Anwenderhandbuch für den Editor.

Kapitel 2

Grundlagen

In diesem Kapitel wird eine Übersicht über die Grammatiken gegeben, für welche die Editoren entwickelt worden sind. Weiterhin wird der für die Studienarbeit wichtige Teil des Eclipse-Framework erläutert.

2.1 Grammatiken

Zur Entwicklung von Dialog-Management-Systemen werden Grammatiken mit der Mächtigkeit von kontextfreien Grammatiken verwendet.

Kontextfreie Grammatiken sind folgendermaßen definiert[ ]:

$G = (N, \sigma, P, S)$, wobei N Menge der Nichtterminalsymbole, σ Menge der Terminalsymbole, P Menge der Produktionsregeln und S Startsymbol ist. Für alle Regeln P gilt: $A \rightarrow \xi$ wobei $A \in N, \xi \in V^*$ und $V \in \sigma \cup P$

In Worten bedeutet dies, dass auf der linken Seite ein Nichtterminal steht und auf der rechten Seite eine Folge von Terminalen oder Nichtterminalen in beliebiger Reihenfolge stehen kann oder dass die rechte Seite leer (ε) ist.

Es gibt verschiedene Arten der Darstellung von kontextfreien Grammatiken. Eine Art der Darstellung ist die, welche bei der oben dargestellte Definition verwendet wurde. Die Darstellungselemente sind hier nur Nichtterminale, Terminale und Pfeile. Andere Darstellungsarten sind die Backus-Naur-Form (BNF) und die erweiterte BNF (EBNF)[ ]. Bei der BNF werden Produktionsregeln zur Darstellung verwendet, allerdings unterscheidet sich die Notation im Detail. Bei der EBNF ist die Verwendung von regulären Ausdrücken innerhalb der Produktionen möglich. Eine weitere Art der Notation, ohne Mächtigkeit hinzu zu fügen, bieten JSGF-Grammatiken und Tapas-Grammatiken.

2.1.1 Java Speech Grammar Format

Das JSGF[ ] ist ein Format für die textuelle Repräsentation von Grammatiken, welches für Dialog-Management-Systeme verwendet wird und vom W3C[1] standardisiert wurde. Die Mächtigkeit dieser Grammatiken entspricht in der Chomsky-Hierarchie den kontextfreien Grammatiken.

Nichtterminale werden in eckigen Klammern geschrieben und dürfen alle Zeichen enthalten, welche in Java Variablenbezeichner sein können[ ]. Außerdem sind noch folgende Zeichen erlaubt:

```
+ - : ; , = | / \ ( ) [ ] @ # % ! ^ & ~
```

Eine Datei im JSGF-Format besteht aus zwei Teile, nämlich dem Kopf der Datei und dem Rumpf.

Der Kopf der Grammatik beginnt mit dem *self- identifying header*, welcher die Version des JSGF-Formates, das verwendete Encoding sowie die Sprache, für welche die Grammatik abgefasst ist, bekannt gibt. In der nächsten Zeile wird der Name der Grammatik definiert. Darunter können Produktionen, welche im JSGF-Kontext Regeln genannt werden, in den Namensraum der jeweiligen Datei importiert werden.

Im Rumpf werden die Regeln definiert, welche die Grammatik definieren. Eine Regel kann dabei sowohl privat, also nicht von anderen Grammatiken importierbar, als auch öffentlich sein, so dass sie in andere Grammatiken importiert werden kann.

Die Regeln können auf der rechten Seite Terminale und Nichtterminale enthalten. Dabei sind Gruppierungen von diesen durch Klammerung möglich. Auch die Verwendung des Kleenestar sowie das Verodern von einzelnen Terminalen, Nichtterminalen oder Gruppen ist möglich. Weiterhin können veroderte Alternativen mit Gewichtungen annotiert werden, welche die Wahrscheinlichkeit für das Vorkommen dieser Alternative darstellen.

Zudem ist es möglich, so genannte *Tags* an Terminale, Nichtterminale oder Gruppen anzuhängen. Innerhalb von geschweiften Klammern dürfen anwendungsspezifische Anweisungen stehen, die aus Sicht der JSGF-Grammatik ignoriert werden. Damit kann der Hersteller eines Dialog-Management-Systems eines konstruieren, welches gültige JSGF-Grammatiken akzeptiert und diese mit eigenen Befehlen anreichert. Diese Grammatiken würden die standardkonforme Grammatik faktisch inkompatibel zu anderen Dialog-Management-Systemen machen. Allerdings wird es durch die Nutzung eigener Befehle möglich, die Grammatiken sowohl im Sinne der Chomsky-Hierarchie mächtiger zu machen, als auch ihre Ausdrucksmächtigkeit zu erhöhen.

[1] www.w3c.org

2.1.2 Tapas

Tapas[ ] ist ein Dialog-Management-System, welches an der Universität Karlsruhe entwickelt wird. Es akzeptiert Grammatiken, welche syntaktisch mit den JSGF-Grammatiken übereinstimmen. Semantisch wertet es diese Grammatiken jedoch auf. Hier sollen nur die für die Erweiterung des JSGF-Editors relevanten Features und Unterschiede besprochen werden.

Der größte Unterschied besteht darin, dass neben der Grammatik noch eine Ontologie definiert wird, mit deren Hilfe neue Regeln generiert werden können. Dies führt zu einem Definitionsfehler in JSGF-Grammatiken, da dort eine Regel, welche durch ein Nichtterminal referenziert wird, entweder in der Grammatik definiert oder importiert werden muss.

Regeln im Tapas-Format werden in einer Vektor-Notation definiert. Sie sehen folgendermaßen aus:

```
<semantische Definition,
  syntaktische Definition,
  2. syntaktische Definition>
```

Im semantischen Teil der Definition wird ein Typ referenziert, welcher in der Ontologie definiert wurde. Im syntaktischen Teil kann die grammatikalische Klasse der Regel definiert werden. Im ersten syntaktischen Teil sind folgende Klassen möglich: VP, V, NP, N, AP, A. Der zweite Teil ist nicht näher bestimmt, hier könnte beispielsweise *inf* für Infinitiv stehen.

Die Ontologie ist objekt-orientiert. Es können Klassen definiert werden, welche typisierte Eigenschaften haben. Diese Klassen können von anderen Klassen erben. Daher sind Regeln auf rechten Seiten referenzierbar, obwohl sie nicht definiert wurden. Dieses ist dann erlaubt, wenn der semantische Typ der betreffenden Regel in der Vererbungshierarchie eines Types, zu dem eine Regel definiert wurde, als Nachfolger steht und die gleichen syntaktischen Attribute mit dieser Regel teilt.

Weiterhin muss bei diesen Regeln beachtet werden, dass die auf der rechten Seite auftauchenden Nichtterminale entweder direkt in der Vererbungshierarchie als Eigenschaft des Nichtterminals auf der linken Seite stehen, oder, dass sie durch Angabe eines Tags auf dieses gemappt werden.

Neben diesen Regeln gibt es auch rein strukturelle Regeln, welche der Notation wie im JSGF folgen und keine weitere Bedeutung beisteuern.

Das Konzept von Tapas-Grammatiken soll durch folgende Grammatik und Ontologie verdeutlicht werden:

```
// semantische Regeln:
```

```
<act_open,VP,_> = <hlp_polite> <obj_openable,N,_> { OBJ
obj_openable } öffnen;

<obj_window,N,Sg>   = das Fenster;
<obj_door,N,Sg>     = die Tür;

// strukturelle Regel:
<hlp_polite> =  würdest du bitte | kannst du bitte;
```

Die Grammatik beginnt mit semantischen Regeln, gefolgt von einer strukturellen Regel. Zusätzlich soll folgende Ontologie angenommen werden:

```
class act_open {
  obj_openable  : OBJ;
};

class obj_openable inherits generic:object;

class obj_window inherits obj_openable;
class obj_door inherits obj_openable;
```

In der Ontologie werden durch das Schlüsselwort *class* semantische Typen definiert. Diese können Attribute haben. Der semantische Typ *act_open* hat das Attribut *obj_openable*. Dieses Attribut wird in der ersten Regel benutzt, um das semantische Konzept von *obj_openable* mit dem von *act_open* verbinden zu können. Näheres dazu findet sich in [ ].

Durch das Schlüsselwort *inherits* kann ein semantischer Typ von einem anderen semantischen Typ erben. So werden die semantischen Typen *obj_window* und *obj_door* auch *obj_openable*.

Durch die Vererbung sind folgende Ableitungen möglich, ohne, dass sie explizit in der Grammatik definiert wurden: würdest du bitte das Fenster öffnen, würdest du bitte die Tür öffnen, kannst du bitte das Fenster öffnen, kannst du bitte die Tür öffnen.

2.2 Eclipse

Eclipse ist ein Framework, um Entwicklungsumgebungen oder Rich-Client-Anwendungen zu entwickeln. Das Eclipse-Projekt wurde von IBM im Jahre 2001 veröffentlicht und für die Öffentlichkeit kostenlos freigegeben. Es steht unter der Common Public License[ ] und ist

damit auch in kommerziellen Projekten kostenlos nutzbar. Seitdem engagieren sich sowohl Open-Source-Entwickler, als auch Unternehmen in der Weiterentwicklung des Projektes. Da Eclipse einen großen Fundus an fertigen Modulen für die Entwicklung einer Entwicklungsumgebung und für die Erweiterung der bestehenden Entwicklungsumgebung bietet, ist es mit vergleichsweise geringem Aufwand möglich, Tools zu entwickeln, welche professionellen Ansprüchen genügen.

2.2.1 Das Framework

Eclipse kann als Umsetzung des Entwurfsmusters *Framework* angesehen werden. Komponenten werden bei einem System, dem Framework, registriert und später von diesem bei Bedarf aufgerufen. Das Framework bietet eine Anwendungsarchitektur, die feste Programmabläufe vorsieht und Punkte, an denen diese Programmabläufe erweitert werden können. An diesen Punkten kann der Entwickler eigenen Programm-Code einfügen, der den Programmablauf des Frameworks erweitert. Der Programm-Code wird durch die registrierten Komponenten bereitgestellt.

Das Framework-Konzept steht im Gegensatz zum Programmieren mit einer Programmbibliothek. Beim Programmieren mit einer Programmbibliothek wird ein Programm dadurch entwickelt, dass der Programmablauf durch den Entwickler festgelegt wird und die Funktionen der Bibliothek aufgerufen werden.

Die Komponenten heißen im Eclipse-Fall Plugins. Diese werden von einem Kern erkannt, in das Framework eingebunden und bei Bedarf aufgerufen. Dieses Konzept des späten Ladens ermöglicht kürzere Ladezeiten als beim direkten Laden und spart Speicher, wenn nicht alle Komponenten gleichzeitig genutzt werden.

Es werden dem Benutzer einer Eclispe-Software alle zur Verfügung stehenden Funktionen über Plugins angeboten. Die Erweiterung von Eclipse geschieht durch das Entwickeln von Plugins. Diese stellen eigene Funktionen bereit und können bestehende Plugins erweitern.

2.2.2 Plugin

Ein Plugin besteht aus einem Plugin-Descriptor und den eigentlichen kompilierten Java-Klassen. Beide Elemente werden in einem Unterordner des Plugin-Verzeichnisses in der Eclipse-Installation kopiert. Wenn das Plugin dann nach dem nächsten Start von Eclipse benötigt wird, so wird es automatisch geladen. Statt die Plugin-Dateien in ein Verzeichnis zu kopieren, ist es möglich, diese Elemente in ein Java-Archiv (JAR) zu verpacken und dieses in das Plugin-Verzeichnis zu kopieren. Dies spart Platz und bringt Ordnung in den Plugin-Ordner. Dieser Weg ist erst seit Eclipse 3.1 möglich und wurde in dieser Arbeit für

die Umsetzung genutzt.

Im Plugin-Descriptor wird deklarativ die Erweiterung (hier *extension*) des Frameworks, also anderer Plugins, an so genannten *extension points* in XML beschrieben[9]. Dazu sind Abhängigkeiten zu anderen Plugins und die zu erweiternde Stelle durch Angabe der Klasse, durch welche die Erweiterung implementiert ist, zu deklarieren. Das erweiterte Plugin kann bei seiner Ausführung über das Framework ermitteln, welche Klassen sich an diesem *extension point* registriert haben und diese instantiieren und den für diese Stelle im Framework vorgesehenen Programm-Code zur Ausführung bringen.

Weiterhin sind Stellen, an denen ein Plugin erweitert werden kann, an dieser Stelle zu deklarieren. Dies geschieht durch eine der XML-Definitionssprache XML-Schema ähnliche Sprache. In ihr kann deklariert werden, welche Optionen deklarativ über den *extension point* übergeben werden können. Die Reihenfolge und Anzahl der Optionen ist dort deklariert.

Durch dieses deklarative Vorgehen kann ohne jegliche Programmierung ein Software-Vertrag etabliert werden, welcher vor der Ausführung auf das Vorhandensein benötigter Komponenten hin überprüft werden kann. Versions-Konflikte sind durch das deklarative Beschreiben der Abhängigkeiten automatisch im Voraus erkennbar.

2.2.3 Sichten, Perspektiven, Editoren

Durch das Plugin-Konzept kann das Eclipse-Framework erweitert werden, ohne dass es der Benutzer bemerkt. Beispielsweise könnten mathematische Bibliotheken eingebunden werden oder wie im Falle des Editors, ein Plugin, welches den Parser beinhaltet. Die Benutzeroberfläche von Eclipse wird hingegen erweitert, indem neue *Sichten*, *Editoren* und *Perspektiven* über Plugins hinzugefügt werden.

Eine Eclipse-Oberfläche besteht aus vielen Fenstern, welche die für den Benutzer erreichbare Funktionalität zur Verfügung stellen. Dies können Editoren, Dateibäume oder auch Klassenhierarchieanzeigen sein. Die Organisation dieser Fenster, sowohl in Größe, als auch Position, wird *Perspektive* genannt. Die Fenster selber heißen *Sichten*.

Weiterhin gibt es noch den Begriff des *Editors*. Damit ist eine Sicht gemeint, welche mit einem Datenmodell, also beispielsweise einer Datei verknüpft ist. In einem Editor kann das Datenmodell bearbeitet werden. Eine Sicht hingegen stellt das Datenmodell nur dar.

Ein Sicht wird vom JSGF-Editor verwendet werden, um die Struktur der bearbeiteten Grammatik darzustellen. Weiterhin wird eine Sicht für die Übersicht über die Aufgaben und Fehler benötigt werden. Eine spezielle Perspektive ist für die JSGF-Entwicklung nicht notwendig, da die Standard-Perspektive, welche von Eclipse beim Start gewählt wird, alle notwendigen Sichten und Editoren anzeigt.

Sichten, Perspektiven und Editoren werden durch das Erweitern von bestehenden Klassen und das deklarative Erweitern von Plugins entwickelt.

2.2.4 Fazit

Die Entwicklung einer Eclipse-Anwendung geschieht immer deklarativ und durch das Erstellen von Programmcode. Die Deklarativität ermöglicht die Inspektion von Plugins schon vor der Benutzung ohne Kenntnis des Quelltextes. Dies schafft Robustheit, da Konflikte vor der Ausführung eines Programms erkannt werden.

Plugins werden zu einer Anwendung komponiert und stellen dann wieder ein eigenes Framework dar, welches wieder erweiterbar ist. So kann die Entwicklung im Eclipse-Kontext zu erweiterbaren Anwendungen führen.

Kapitel 3

Anforderungen und Umsetzung

Es war ein Editor zu entwickeln, welcher das Editieren sowohl von JSGF-Dateien, als auch von Tapas-Dateien ermöglichen sollte. Die Tapas-Funktion sollte ausschaltbar sein, da eine Integration des JSGF-Editors in das Eclipse-Subprojekt *Voicetools* angestrebt wird. Weiterhin sollte der Parser des Editors, welcher benutzt wird, um die Struktur der Grammatikdateien in ein Modell abbilden zu können, auch im Tapas-Dialog-Management-System genutzt werden. Das soll einerseits doppelte Arbeit bei der Entwicklung des Parsers verhindern. Andererseits soll so einer Inkompatibilität zwischen Dialog-Manager und Editor vorgebeugt werden.

Diesen nichtfunktionalen Anforderungen schlossen sich folgende funktionale Anforderungen an:

- *Syntax-Highlighting:* Besondere Elemente einer Grammatik sollten farblich hervorgehoben werden.
- *Outline:* Die Struktur der Grammatik sollte als Baum angezeigt werden. Wenn der Benutzer einen Knoten auswählt, dann sollte diese Stelle im Programm angesprungen werden.
- *Sprung zur Definition:* Wenn der Cursor über einem Nichtterminal ist, dann sollte nach Drücken der F3-Taste an die Stelle der Definition gesprungen werden.
- *Anzeigen der Definition:* Wenn der Benutzer mit dem Mauszeiger über einem Nichtterminal verharrt, dann sollte in einer Box, welche über dem Nichtterminal erscheint, die Definition angezeigt werden.
- *Aufgabenliste:* Wenn ein Benutzer frei wählbare Schlüsselwörter, wie beispielsweise *TODO* in einen Kommentar einfügt, dann sollte der Rest der Zeile in einer Liste gesammelt werden, damit projektübergreifend die Stellen gemerkt werden, welche später bearbeitet werden müssen.

- *Eingabevervollständigung:* Wenn der Benutzer anfängt, ein Nichtterminal zu schreiben, dann sollte der Editor ihm Vorschläge aus bereits in der Grammatik definierten Nichtterminalen machen, so dass er nach dem Tippen weniger Buchstaben diese zum vollständigen Nichtterminal erweitern lassen kann. Diese Funktion sollte auch an ausgewählten anderen Stellen, wie beispielsweise dem Grammatikkopf, verfügbar sein.

- *Automatische Fehlererkennung:* Wenn ein Benutzer einen Fehler (syntaktisch / semantisch) macht, so sollte dieser rot unterstrichen werden, wie dies in modernen Textverarbeitungen geschieht.

- *Automatische Fehlerbereinigung:* Fehler sollen nicht nur erkannt werden, sondern es sollen Lösungsvorschläge erzeugt werden, welche der Benutzer mit einem Tastendruck aktivieren kann.

3.1 Architektur

Diese Anforderungen führten zu der Unterteilung des Editors in drei Plugins.

Das erste Plugin ist der JSGF-Editor, welcher die funktionalen Anforderungen für JSGF-Grammatiken erfüllt. Dieser benutzt ein Plugin, welches den Parser und ein Modell für JSGF-Grammatiken liefert. Diese Abspaltung des Parsers war nötig, da dieser auch für den Tapas-Dialog-Manager genutzt werden sollte. Darum wird im Parser-Plugin keine eclipse-spezifische Funktion genutzt. Lediglich der deklarative Deskriptor macht die Funktionen des Parsers anderen Plugins zugänglich, macht den Parser damit zum Plugin.

Das dritte Plugin ist die Erweiterung des Parser-Plugins und des Editors um die Tapas-Funktionalität. Dieses Plugin wird irgendwann Teil einer größeren Entwicklungsumgebung werden, die dem Plugin die nötige Anbindung an Tapas liefert. Für die Studienarbeit wurde angenommen, dass dieses Plugin existiere und die nötige Funktionalität wurde durch ein Plugin verfügbar gemacht, welches Funktionen des Tapas-Dialog-Managers exportiert.

Abbildung 3.1 verdeutlich die gewählte Architektur. Die Plugins sind in diesem UML-Diagramm als Teilsysteme modelliert. Der Tapas-Dialog-Manager (DM) stellt hier, wie Eclipse, kein Plugin dar, sondern ein Teilsystem.

Die Aufteilung der Pakete innerhalb des JSGF-Editors ist nach den Vorschlägen für die Benennung von Paketen in Eclipse vorgenommen worden [ ]. Das bedeutet, dass Klassen Paketen zugeordnet werden, welche ihrer Funktionsklasse entsprechen. Beispielsweise sind alle Klassen, welche das Syntax-Highlighting betreffen, in dem Paket *highlighting* untergebracht werden. Das Prefix für den vollständigen Paketnamen ist *org.eclipse.vtp.editor.jsgf*. Als Infix wird das Wort *internal* eingefügt, wenn die Klasse im Namensraum außerhalb des Plug-

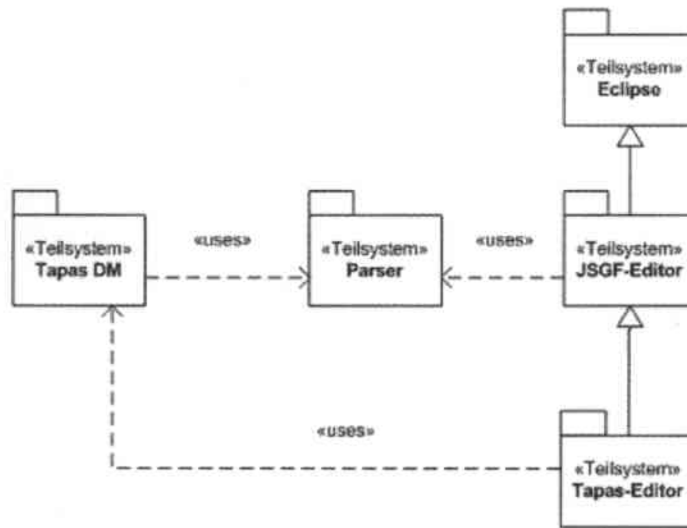

Abbildung 3.1: UML-Diagramm der Architektur der Plugins

ins nicht sichtbar sein soll. So ist der vollständige Name für die Klasse *JSGFCodeScanner*: *org.eclipse.vtp.editor.jsgf.internal.JSGFCodeScanner*. Wäre diese Klasse für die Benutzung außerhalb des Plugins gedacht, so hieße sie: *org.eclipse.vtp.editor.jsgf.JSGFCodeScanner*.

Im Folgenden werden nun die einzelnen Komponenten des Editors beschrieben.

3.2 Parser-Plugin

Ein rudimentärer Parser, der JSGF-Grammatiken parsen konnte, wurde zu Anfang der Studienarbeit zur Verfügung gestellt. Er musste um kleinere Fehler bereinigt werden. Es fehlten desweiteren Funktionen, die für den Einsatz des Parsers in einem Editor unerlässlich waren. Dies war die Möglichkeit, die Position einzelner Grammatikbestandteile innerhalb der Grammatikdatei zu bestimmen. Es fehlte ein zentrales Fehlermodell und es fehlten die Möglichkeit, die Kommentare der Grammatik erweiterbar zu analysieren, wie auch die Möglichkeit, das Verhalten des Parsers bei Nichtterminalen und den so genannten Tags zu beeinflussen.

Der Symbolentschlüsseler (*Lexer*) des Parsers wird von dem Lexer-Generator JFlex[ ] generiert. Für diesen Lexer wurde ein Fehlermodell entwickelt, welches im Editor zum Anzeigen von Fehler genutzt wird.

Das Standard-Verhalten des Lexers war es, eine Ausnahme zu werfen, wenn ein Fehler auftrat, was den Vorgang der Symbolentschlüsselung beim ersten Fehler abbrach. Dieses Verhalten ist bei einem Editor nicht wünschenswert, da während eine Grammatik erstellt wird und nur eine kleine Pause beim Tippen gemacht wird, die Grammatik in einem nicht validen Zustand zurück bleibt. Dies hängt damit zusammen, dass bei kleinen Pausen der

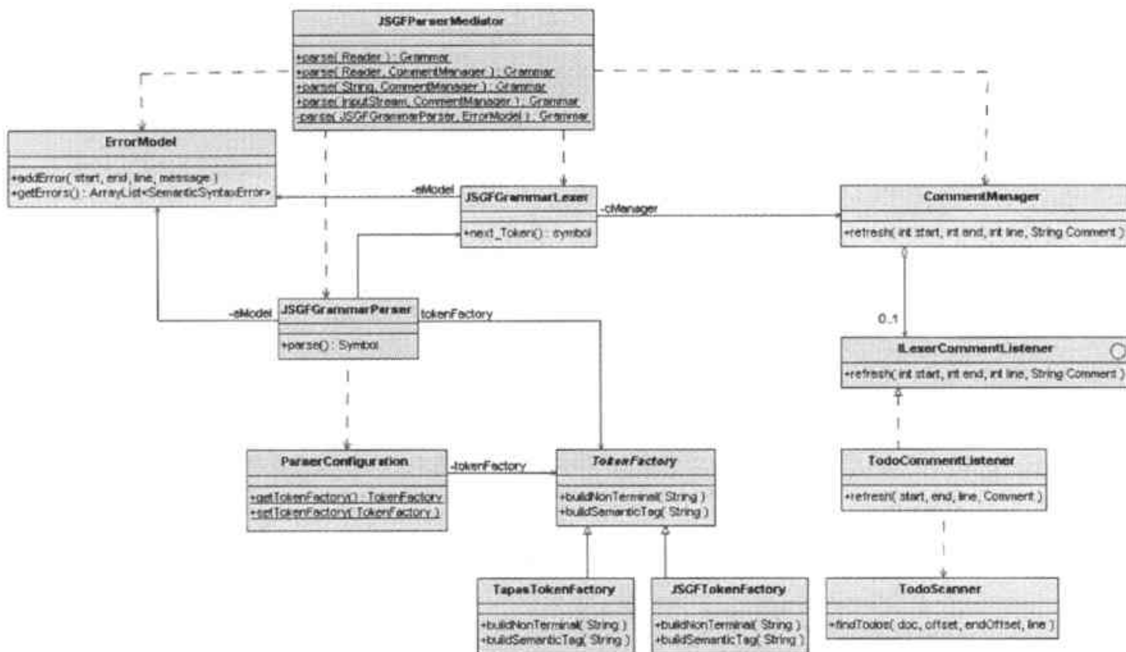

Abbildung 3.2: UML-Diagramm Parser

Parser gestartet wird, um die Grammatik zu analysieren. Dies hätte zur Folge, dass ein sich mitten in der Grammatik befindlicher Fehler das Weiterlesen der Grammatik ab dieser Stelle stoppen würde. Der Aufbau des Grammatik-Modells würde verhindert werden und dadurch wären viele der Funktionen des Editors nicht mehr verfügbar. Darum wurde das Verhalten des Lexers so verändert, dass dieser beim Auftreten eines Fehlers diesen in einem Fehlermodell-Behälter abgelegt, um dem Benutzer diesen später zu präsentieren. Der Lexer überliest das Zeichen, an dem er sich gerade befindet und bleibt in dem Zustand, in dem er sich vor Auftreten des Fehlers befunden hat.

Ähnliches gilt nun auch für den Zerteiler (*Parser*). Auch hier musste eine Möglichkeit gefunden werden, diesen vom Abbruch des Zerteilens abzuhalten, wenn dieser einen Fehler fand. Der verwendete *Look-ahead LR-Parser* [ ] mit Namen Cup[ ] sieht für diesen Fall eine Möglichkeit vor. Zur Fehlerverarbeitungskontrolle kann ein so genanntes Synchronisations-Token definiert werden. Der Parser versucht im Fehlerfall solange weitere Token zu lesen, bis das Synchronisations-Token gefunden wird. Die bis dahin gelesenen Token werden als Fehler-Token erkannt. Wird schließlich ein Synchronisations-Token gefunden, so wird die in Bearbeitung befindliche Produktion erfolgreich beendet. An diese Funktion wurde ein Fehlermodell angehangen, so dass nicht nur die Produktion erfolgreich beendet wird, sondern auch ein Fehler in dieses Fehlermodell eingetragen wird. So werden möglichst viele Syntaxfehler

gesammelt und dem Benutzer präsentiert.

Eine weitere Anforderung, welche aus der Benutzung des Parsers im Editorkontext entstand, war, dass die exakte Position eines Tokens im Editor wichtig ist. Wenn zum Beispiel zu der Definition eines Nichtterminals gesprungen werden soll, dann ist die Kenntnis über die Position in der Datei erforderlich. Das Modell der Grammatik musste entsprechend angepasst werden. Dazu wurde eine abstrakte Klasse *JSGFToken* eingeführt, von welcher alle anderen das Grammatikmodell repräsentierenden Klassen erben. Diese Klasse enthält außerdem eine Information darüber, ob der Teil, der durch eine Unterklasse von *JSGFToken* repräsentiert wird, fehlerhaft ist.

Um den konkreten Parser austauschbar zu machen und seine Benutzung zu vereinfachen, wird eine Klasse nach dem Vermittler-Entwurfsmuster mit Namen *JSGFParserMediator* benutzt. Sie bietet polymorphe Schnittstellen, welche flexibel genutzt werden können und sich um die konkrete Verwendung des Parsers kümmern.

Das UML-Modell in Abbildung 3.2 zeigt schematisch den Aufbau der Parser-Architektur. Die Klassen, welche zum *CommentManager* gehören[1], werden in Abschnitt 3.7.2 näher erläutert. Die Klasse *ParserConfiguration* stellt eine abstrakte Fabrik[2] dar, welche durch konkrete Fabriken konfigurierbar ist. Diese Fabriken[3] erben von der abstrakten Klasse *TokenFactory* und sind für das Erzeugen von Repräsentationen von Nichtterminalen und semantischen Tags für die Modell-Repräsentation der Grammatik zuständig. Wann immer der Parser ein Nichtterminal erkannt hat, wird über die statische Methode *getTokenFactory()* eine Referenz auf die gerade zu benutzende *TokenFactoy* geholt. Über diese werden dann die Nichtterminale mit der Methode *buildNonTerminal(String)* erzeugt. Genauso verhält es sich für die Methode *buildSemanticTags*, welche zum Erzeugen von semantischen Tags benutzt wird. Diese Implementierung war notwendig, da der Parser zwar auf JSGF-Grammatiken arbeiten sollte, aber auch die erweiterte Syntax von Tapas-Grammatiken verstehen sollte. Er muss also erweiterbar sein. Da der Parser aber generiert wird, ist seine Erweiterbarkeit nicht durch einfache Vererbung lösbar. Eine austauschbare Fabrik schafft die benötigte Flexibilität für die Erweiterbarkeit.

Die Klassen *JSGFGrammarLexer* und *JSGFGrammarParser* repräsentieren den Lexer und den Parser. Der Lexer benutzt das Fehlermodell, implementiert durch die Klasse *ErrorModel* und die Klasse *CommentManager*. Der Parser nutzt auch das Fehlermodell zur Verwaltung der von ihm gefundenen Fehler und zusätzlich eine *TokenFactory* zur Erzeugung von Nichtterminal- und *Tag*-Repräsentationen.

[1] *CommentManager, ILexerCommentListener, TodoCommentListener, TodoScanner*

[2] [ ] Seite 87

[3] *TapasTokenFactory, JSGFTokenFactory*

3.3 Syntax-Highlighting

Die Benutzung von Texteditoren, erleichtert es nicht, die Übersicht bei der Entwicklung von Grammatiken zu behalten. Text, der in Farbe und Form immer gleich erscheint, macht es schwer, syntaktische und semantische Strukturen der Grammatik zu erfassen. Dem Benutzer sollte durch Hervorhebungen in der zu bearbeitenden Grammatik das Verständnis erleichtert werden. Durch ein besseres und schnelleres Verständnis kann die Fehlerrate bei der Entwicklung verkleinert werden.

```
#JSGF V1.0 ISO-8859-1 german;

grammar generic.g1;
import <a.c.*>;

/*
============================================
generic rules for german grammar
============================================

*/
/* TODO add gq*/

    public <statement>  = <yes> | <no>  ;
    <yes>    = ja [bitte];
    <no>     = nein [danke]
               | nein ohne;
    public <testRule> = <VOID>|<NULL>;
```

Abbildung 3.3: Syntax-Highlighting im JSGF-Editor

Zur besseren Lesbarkeit und damit zum schnelleren Verständnis der Grammatiken sollten Schlüsselwörter, Nichtterminale, Kommentare und Tags farbig und in ihrer Gestalt hervorgehoben werden.

Da in Tapas oder anderen JSGF-Formaten innerhalb von Nichtterminalen oder Tags andere Hervorhebungen wünschenswert sein könnten, sollten diese leicht durch ein Plugin erweiterbar sein.

Ein Beispiel für die Umsetzung dieser Anforderungen im JSGF-Editor zeigt Abbildung 3.3. Die in dieser Abbildung dargestellten Farben für die Hervorhebung sind über eine Optionsseite, welche über die Standardeinstellungsseite aufgerufen werden kann, konfigurierbar.

3.3.1 Wege in Eclipse

Ein Dokument in einem Editor besteht aus mehreren disjunkten Partitionen. Die Partitionierung wird von einem *IDocumentPartitioner* erstellt. Dazu wird ein *IPartitionTokenScanner*

benutzt, welcher aus einem Datenstrom, wie ein Lexer, Token extrahiert. In diesem Fall heißen die Token Partitionen und bilden eine Unterteilung des Dokumentes in festgelegte Bereiche. Das Eclipse-Framework bietet eine Implementierung des Scanners, welche nur noch um Regeln, die reguläre Ausdrücke repräsentieren, erweitert werden muss[[illegible]].

Um Syntax-Highlighting in einem *Editor* in Eclipse verfügbar zu machen, gilt es, eine Implementierung des *IPresentationReconciler* zu erstellen. Dieser definiert Methoden, die einen *Damager* und einen *Repairer* zu einer Partition zurückgeben.

Die Implementierung des *IPresentationReconciler* ruft bei der Evaluierung des eingegebenen Textes den für die Partition registrierten *Damager* auf. Das passiert durch einen Hintergrund-Thread, welcher aktiviert wird, wenn keine Eingaben durch den Benutzer gemacht werden. Dieser gibt die Region, repräsentiert durch Anfang und Ende in einem Dokument, zurück, bei welcher eine neue Hervorhebung berechnet werden muss. Diese Information wird dann an den *Repairer* gegeben, welcher die Stellen im Text mit entsprechenden Farben attributiert.

3.3.2 Konkrete Umsetzung

Die Aufgabe ist durch zwei Strategien umgesetzt. Zum einen werden JSGF-spezifische Partitionen gebildet. Aufbauend auf diesen wird dann eine Standard-Implementierung eines *Damager- / Repairer*-Gespanns verwendet, der *DefaultDamagerRepairer*. Dieser beinhaltet sowohl *Damager*, als auch *Repairer*. Der Repairer benutzt einen so genannten Scanner, der wie ein Lexer den Eingabestrom in Token zerteilt und diesen Token eine Hervorhebung zuordnet.

Weiterhin wird die selbe Strategie benutzt, um Schlüsselwörter hervorzuheben.

Eine andere Strategie wird benutzt, um Nichtterminale und Tags hervorzuheben. Die Hervorhebung dieser Teile der Grammatik soll erweiterbar sein. Darum wurde ein Extension-Point definiert, über den man besondere Hervorhebungen für diesen Teil der Grammatik nachladen kann.

Es kommt in diesem Fall auch ein *DefaultDamagerRepairer* zum Einsatz. Dieser wird mit einem vom Eclipse-Framework mitgelieferten Scanner bestückt, welcher mit so genannten *IRules* konfigurierbar ist. Beim *IRule*-Interface muss man eine Methode *evaluate* implementieren, welche einen Eingabestrom als Parameter übergeben bekommt und ein attributiertes Token zurückliefert. Diese Regeln werden fortlaufend auf die Eingabe angewendet, bis der vom *Repairer* zu bearbeitende Teil durchlaufen ist.

Über den Extension Point kann man nun Regeln für Nichtterminale und Tags definieren. Die Nichtterminale <*VOID*> und <*NULL*> haben in JSGF eine besondere Bedeutung, darum wäre es wünschenswert, wenn sie besonders eingefärbt wären. Für diese zwei Nichtterminale ist eine Erweiterung für das Syntax-Highlighting beispielhaft mit der Klasse *Key-*

WordHighlightExtension implementiert worden.

3.4 Outline

Da Grammatiken sehr groß werden können, ist eine grafische Übersicht über ihre Struktur eine Möglichkeit, um einen Überblick über sie zu bekommen. Eine solche Struktur kann dann zur Navigation innerhalb der Grammatiken verwendet werden. Dabei sollten die Informationen in dieser Struktur auf das Nötigste reduziert werden, damit ein Vorteil aus der verkürzten Darstellung entsteht. Dies ist in Grammatiken leicht möglich, da man dort zwischen drei Hauptarten von Ausdrücken unterscheiden kann. Dies ist zum einen eine Import-Anweisung, welche am Anfang der Grammatik auftaucht. Zum anderen gibt es Kommentare, die innerhalb der ganzen Grammatik auftauchen. Und es gibt Definitionen von Regeln. Diese beginnen nach den Import-Anweisungen.

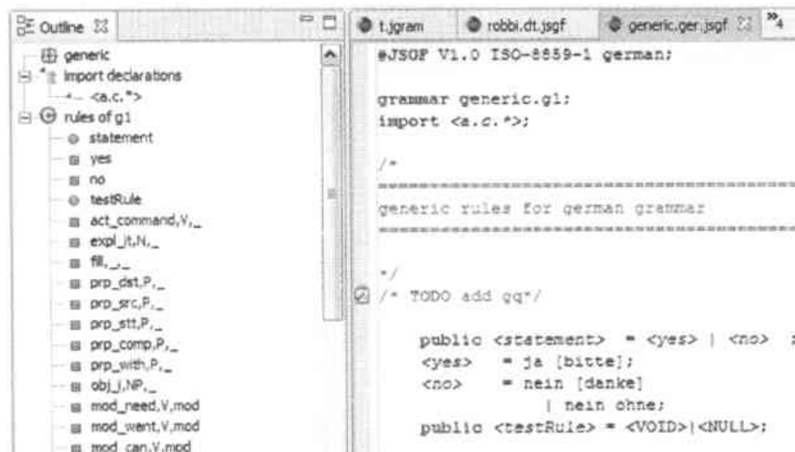

Abbildung 3.4: Outline im JSGF-Editor

So kann eine Art der Informationsreduktion das Weglassen von Kommentaren sein. Dies kann sinnvoll sein, da zur Übersicht keine Detail-Informationen über die Grammatik benötigt werden.

Desweiteren können die Import-Anweisungen auf das Zeigen der zu importierenden Regeln beschränkt werden. Da Import-Anweisungen von Natur aus sehr kurz sind, kann man sonst keine Reduktion mehr erreichen.

Bei Regeldefinitionen kann eine Beschränkung auf den Regelnamen gewinnbringend sein. Die rechte Seite der Regel enthält Informationen, die für das Gewinnen der Übersicht unbedingt notwendig sind.

Desweiteren kann man, durch den Einsatz kleiner Grafiken, eine semantische Einordnung des reduzierten Restes vornehmen. Kleine grüne Kreise können so für öffentliche Regeln stehen, während rote Rechtecke für eine nicht öffentliche Regel stehen können.

Abbildung 3.4 zeigt beispielhaft, wie so eine Übersicht im JSGF-Editor aussieht. Dabei wird die Struktur der JSGF-Grammatik in einem Baum angezeigt. Die Knoten sind jeweils einzelne Import-Anweisungen und Regeln. Klickt man eine davon an, so springt der Editor an die Stelle der Definition. Diese Art der Darstellung wird in der Eclipse-Terminologie Outline genannt.

3.4.1 Wege in Eclipse

Da Outlines in Eclipse oft zur Navigation in großen Strukturen verwendet werden, gibt es bereits eine View, welche die grundlegenden Funktionen realisiert. Man muss diese nur noch mit dem für die Anwendung speziellen Verhalten konfigurieren.

Für diese View muss lediglich das Baumelement konfiguriert werden. Dazu muss man eine Klasse schreiben, welche sich um das Beschaffen der Daten kümmert, die im Baumelement visualisiert werden sollen. Es ist weiterhin eine Klasse nötig, welche die Daten in die Baumdarstellung überführen kann. Die Daten können dabei beliebig sein. Durch dieses Konzept der Arbeitsteilung lassen sich die Aufgaben der Datenbeschaffung und die Aufgabe der Datendarstellung aufteilen.

Außerdem muss man das Verhalten der View bei Klicks bestimmen. So soll bei vielen Baumdarstellungen an eine Codestelle gesprungen werden, die durch das jeweilige Blatt im Baum repräsentiert wird.

Beim Outline-View-Framework sorgt eine Klasse, welche das *ITreeContentProvider* Interface implementiert, für das Bereitstellen der Daten. Die Interpretation der Daten zu Text und Bildern übernimmt eine Implementierung des *ILabelProvider* Interface.

3.4.2 Konkrete Umsetzung

Die Umsetzung dieses Teils der Arbeit besteht aus der Implementierung der vorgestellten Interfaces und dem Schreiben des Controllers. Dies wurde in den Klassen *JSGFContentProvider*, *JSGFLabelProvider* und *JSGFContentOutline* umgesetzt.

3.5 Sprung zur Definition

Das Finden der Definition von Nichtterminalen in großen Grammatiken kann schwierig sein, da sie weit weg von ihrer Benutzung definiert sein können. Bei der Grammatikentwicklung

besteht der Bedarf, von der bearbeiteten Stelle, an der ein Nichtterminal verwendet wird, zu dessen Definition zu springen, um die Definition zu sehen oder dort Änderungen vorzunehmen. Nach dem Sprung zur Definition wird die Entwicklung an der Stelle vor dem Sprung fortgesetzt. Das bedeutet, dass auch der Bedarf danach besteht, wieder zurück springen zu können.

Konkret bedeutet das, dass es möglich sein soll, nach Drücken der Taste F3[4], während der Cursor über einem Nichtterminal verharrt, zu dessen Definition zu springen. Weiterhin soll es möglich sein, wie im Internet-Browser durch Drücken von ALT-Links in der Sprunghistorie zurück oder mit ALT-Rechts in der Sprunghistorie vorwärts zu springen.

3.5.1 Wege in Eclipse

In Eclipse muss zum Springen innerhalb des Editors nur die Methode *setHighlightRange* der Editor-Klasse aufgerufen werden. Vor und nach Aufruf der Klasse müssen Markierungen in der Sprunghistorie gesetzt werden, damit das Framework weiß, an welche Stelle es springen muss, um zur letzten Sprungposition zu kommen.

All dies kann in einer so genannten *Action* umgesetzt werden, die dem Command-Pattern nach Gamma[ ] entspricht. Sie wird vom Framework bei entsprechendem Tastendruck aufgerufen.

Die Verbindung von Tastenkombination und *Action* wird deklarativ und prozedural gelöst. Deklarativ läuft dabei die Beschreibung der *Action*, um sie später leicht in andere Strukturen, wie beispielsweise Toolbars, zu integrieren sowie die Festlegung der zu drückenden Taste. Prozedural wird die *Action* an eine konkrete Klasse in der Methode *createActions()* gebunden.

3.5.2 Konkrete Umsetzung

Bei der konkreten Umsetzung ist interessant, wie das Nichtterminal, über dem sich der Cursor befindet, im Grammatik-Modell gefunden wird.

Dies wird durch eine Anfrage an das Grammatik-Modell gelöst. Wenn die Action aufgerufen wird, ist bekannt, an welcher Position sich der Cursor im Editor befindet. Über die *getNonTermAt(int offset)*-Methode kann in der *Grammar*-Klasse des Grammatik-Modells das Nichtterminal, welches sich an der Position des Cursors befindet, gefunden werden. Diese Anfrage verzweigt sich in dem Grammatik-Modell, bis schließlich das Nichtterminal gefunden ist.

[4]dies ist die Taste mit der das auch im JDT geht

3.6 Anzeigen der Definition

Nicht immer ist es wünschenswert, zur Definition eines Nichtterminals zu springen. Oft reicht es, dessen Definition zu kennen. Dies kann vor allem dann hilfreich sein, wenn an einer dem Entwickler unbekannten Grammatik gearbeitet wird.

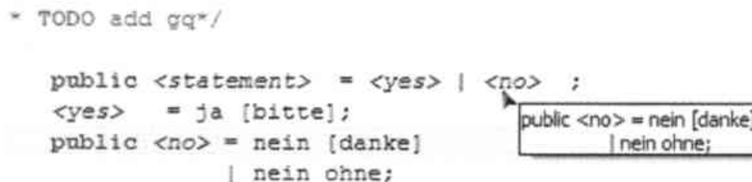

Abbildung 3.5: Anzeige von Definitionen

Dies kann wie in Abbildung 3.5 erreicht werden. Dazu wird der Mauszeiger über einem Nichtterminal zum Stehen gebracht, woraufhin eine Box neben dem Mauszeiger erscheint, welche den Inhalt der Definition anzeigt.

3.6.1 Wege in Eclipse

In Eclipse wird dies durch die Implementierung des ITextHover-Interfaces, welches eine Methode *getHoverInfo(ITextViewer, IRegion)* bereitgestellt, ermöglicht. Diese gibt nach Auswertung einen String zurück, welcher zur Anzeige der Informationen genutzt wird.

3.6.2 Konkrete Umsetzung

Die Gewinnung der darzustellenden Informationen analog zu dem Vorgehen, wie es in Abschnitt 3.5 beschrieben wurde.

3.7 Aufgabenliste

Eine Grammatik wird manchmal von mehreren Menschen entwickelt. Es wird zu unterschiedlichen Zeiten an ihr gearbeitet. In der Entwicklung können manche Nichtterminale nicht sofort definiert werden. Manche Dialogstränge können nicht sofort im Detail modelliert werden. In diesem Fall ist es hilfreich, wenn es möglich wäre, sich die unfertigen Stellen markieren zu können.

Es besteht die Möglichkeit, dies durch Notizen auf einem separaten Blatt Papier zu lösen. Das hätte aber den Nachteil, dass es nicht einfach wäre, an die Stelle zu springen, zur der die Notiz gehört. Selbst das Merken der Zeilennummer ist keine große Hilfestellung, da diese

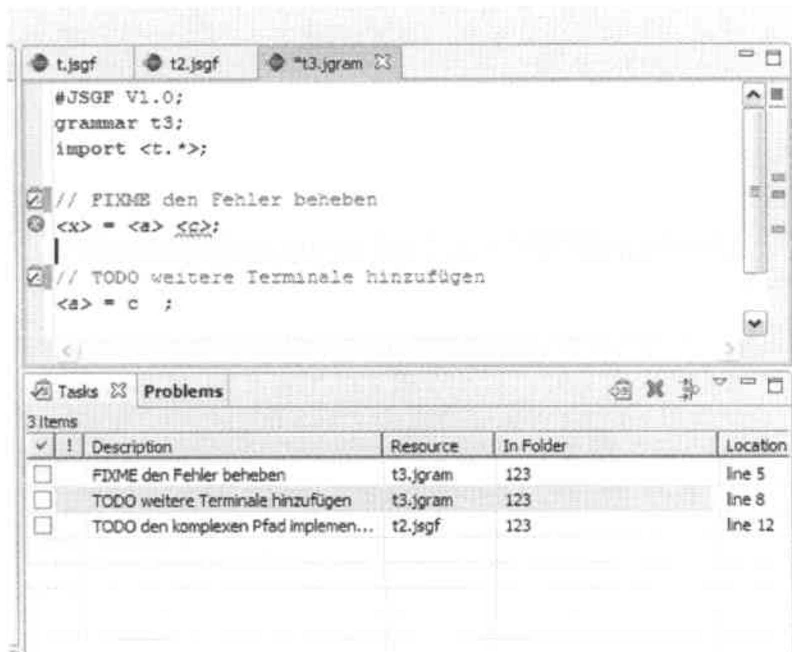

Abbildung 3.6: Aufgaben Liste im JSGF-Editor

Information schnell veraltet, da sich die Position der Stelle, die gemerkt wurde, jedesmal verschieben würde, wenn Änderungen an Teilen der Grammatiken gemacht würden, die über dieser Stelle liegen. Auch der Austausch mit anderen Entwicklern wäre so erschwert.

Es könnten Notizen in Kommentare geschrieben werden. Kommentare sollen diese Funktion erfüllen. Die Verwendung von standardisierte Schlüsselwörter ist für das Wiederauffinden der markierten Stellen hilfreich. Ein Schlüsselwort könnte zum Beispiel *TODO* sein. Im Rest der Zeile könnte dann die zu merkende Aufgabe stehen.

Dieser Ansatz ist grundsätzlich nicht falsch, jedoch besteht weiterhin das Problem, die zu bearbeitenden Stellen zu finden. Zum Finden könnte man die Suchfunktion des Editors verwenden. Dies könnte aber sehr mühevoll sein, wenn reger Gebrauch von dieser Art von Annotation gemacht würde. Die Arbeit steigt sogar noch weiter, wenn verschiedene Schlüsselwörter verwendet wurden. Der Einsatz verschiedener Schlüsselwörter ist sinnvoll, wenn sich Stellen zur Reparatur eines Fehlers von Stellen einer unfertigen Arbeit unterscheiden sollen.

Eine zentrale Stelle im Editor, welche diese Annotationen sammeln könnte, würde das Auffinden dieser erleichtern. Da Grammatiken aus mehreren Dateien bestehen können, sollte diese Hilfe auch über Dateigrenzen hinweg arbeiten können.

Wenn ein Eintrag aus dieser Sammlung ausgewählt wird, sollte das entsprechende Dokument an der Stelle geöffnet werden, an der sich die ausgewählte Annotation befindet.

Wenn die Aufgabe, für welche die Annotation stand, erledigt ist, so ist das intuitive

Verhalten, den Kommentar zu löschen und nicht zusätzlich noch den Eintrag in der Liste zu löschen. Der Editor sollte intuitiv zu bedienen sein. Also sollte dieser Automatismus auch verfügbar sein.

Abbildung 3.6 zeigt, wie das Umsetzen der Aufgabenliste im JSGF-Editor aussieht.

Weiterhin sollten die Schlüsselwörter konfigurierbar sein. Für den englischsprachigen Benutzer mag *TODO* ein guter Merker sein, ein deutschsprachiger Benutzer würde vielleicht *Aufgabe* bevorzugen. Die Menge der Schlüsselwörter sollte nicht künstlich beschränkt sein, da man nicht vorhersehen kann, wie viele Schlüsselwörter der Benutzer haben möchte.

Für die Konfiguration der Schlüsselwörter bietet es sich an, diese in den Dialog zu integrieren, der auch von Eclipse zur Konfiguration genutzt wird.

3.7.1 Wege in Eclipse

In Eclipse lassen sich solche Aufgaben leicht in das Framework integrieren. Es muss lediglich ein so genannter Marker erstellt werden. Marker sind Attributierungen von Textstellen, welchen Informationen wie Wichtigkeit, Art und eine Nachricht oder auch andere Informationen zugeordnet werden können. Diese werden an eine Datei angefügt und das Framework kümmert sich darum, dass Marker des Typs *IMarker.TASK* in der Task-View angezeigt werden. Wenn im Folgenden ein Eintrag aus der Task-View angeklickt wird, dann kümmert sich das Framework auch darum, dass die richtige Datei an der richtigen Stelle geöffnet wird. Das Erstellen und Entfernen der Marker wird nicht vom Eclipse-Framework übernommen, sondern muss implementiert werden.

Der Dialog für die Konfiguration der Schlüsselwörter kann durch das Überschreiben der *FieldEditorPreferencePage*-Klasse erzeugt werden. Diese kann relativ leicht konfiguriert werden, so dass sie sich in den Konfigurationsdialog von Eclipse integriert und die nötigen Eigenschaften konfigurierbar sind.

3.7.2 Konkrete Umsetzung

Man könnte die Erkennung der Tags in den Lexer einbauen, der benutzt wird, um die Grammatik zu lesen, da dieser bereits die Grammatik liest. Das würde bedeuten, dass jedes Mal, wenn ein neues Schlüsselwort hinzugefügt würde, der Lexer neu erzeugt werden müsste. Dadurch würde der Lexer weniger leicht austauschbar werden, da dieses zusätzliche Verhalten der TODO-Schlüsselworterkennung nicht ohne weiteres auf einen anderen Lexer-Generator übertragbar ist. Die Schwierigkeit stellt dabei die unterschiedliche Syntax und das wechselnde Konzept der Lexer-Generatoren dar.

Eine andere Möglichkeit wäre es gewesen, einen zweiten handgeschriebenen Lexer laufen zu lassen. Dann würde viel Arbeit wiederholt werden, was zu teuer ist.

Die Lösung, die hier gewählt wurde, ist zwischen diesen beiden besprochenen Lösungen angesiedelt. Wenn der Lexer, welcher zum Erkennen der Grammatik benutzt wird, bestimmte Sequenzen erkennt, dann werden damit verknüpfte Funktionen aufgerufen. Wenn nun ein Kommentar gefunden wird, ist dies eine Möglichkeit, einen eigenen Lexer zu integrieren, welcher die speziellen Tags erkennt.

Dies wurde mit dem Beobachter-Entwurfsmuster[5] umgesetzt. Es gibt eine Klasse *CommentManager*, welche das Subjekt darstellt. Hier können sich Objekte registrieren, welche das *ILexerCommentListener*-Interface implementieren. Wenn nun ein Kommentar vom Lexer gefunden wird, so wird die Methode *refresh* des *CommentManager* aufgerufen, welche die *refresh*-Methoden aller registrierten Beobachter aufruft. Diese können dann innerhalb des Kommentars nach *TODO-Tags* suchen und geben schließlich eine Liste der gefundenen Tags zurück, welche zur Erzeugung der Marker genutzt wird.

Die Erzeugung des Konfigurationsdialoges bestand in der Implementierung der entsprechenden Methoden in der *FieldEditorPreferencePage*-Klasse.

3.8 Textvervollständigung

Beim Schreiben großer Grammatiken, welche über mehrere Dateien verteilt sein können, ist es nicht immer leicht, sich an die definierten Nichtterminale zu erinnern. Es ist weiterhin nicht wünschenswert, lange Nichtterminale ganz auszuschreiben. Lange Nichtterminal kommen zu großen Teilen in Tapas-Grammatiken vor. Zum schnellen Entwickeln ist es unerlässlich, dass die entwickelnde Person bei diesen Aufgaben unterstützt wird.

Eine Möglichkeit dies zu tun, ist, dem Benutzer, sobald dieser anfängt Buchstaben einzugeben, automatisch oder nach Aufforderung Vorschläge über den Rest der Eingabe zu unterbreitet.

Diese Vorschläge sollten zum jeweiligen Kontext passen. Eine Aufgabe, welche beim Erstellen jeder neuen Grammatik gelöst werden muss, ist das Schreiben des self identifying headers. Hier muss das *Encoding* der Datei angeben werden. Weiterhin ist die Sprache, welche die Grammatik abdeckt, dort einzustellen. Nun kann es für nicht geübte Grammatikschreiber schwer sein, die Kürzel aller Encodings auswendig zu kennen. Auch die korrekten Kürzel für eine Sprache, mitsamt ihrer speziellen Ausprägung, ist nicht jeder entwickelnden Person auf Anhieb geläufig. Darum ist hier eine Vervollständigungshilfe angebracht.

Eine weitere Stelle, an der man sich eine Hilfe wünschen würden, sind Import-Anweisungen.

[5][ ] Seite 293

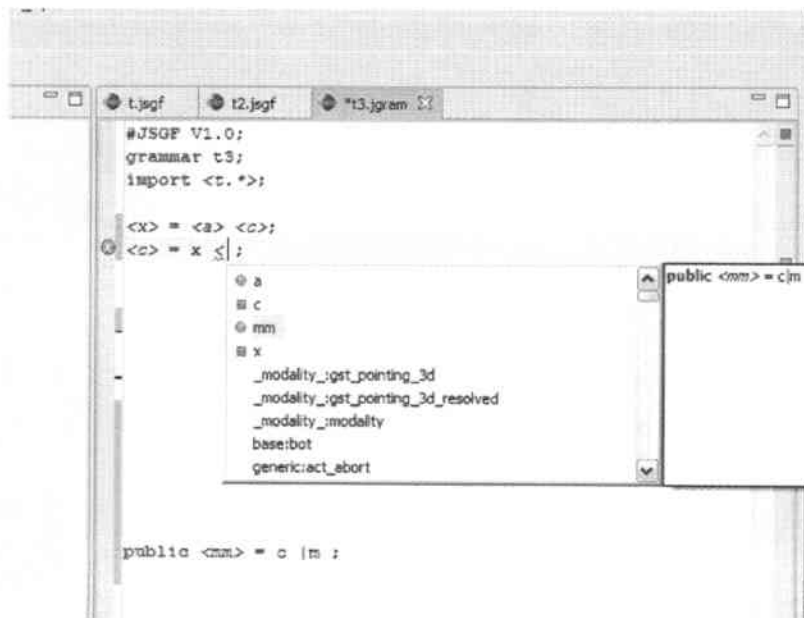

Abbildung 3.7: Ein Beispiel für die Textvervollständigung

Hier gilt es, die Regeln, welche importiert werden sollen, zu finden. Diese stecken oft in unbekannten Grammatiken. In der ersten Stufe muss zu Anfang die richtige Grammatik innerhalb eines Paketes identifiziert werden. Es sollte also nach der Eingabe der spitzen Klammer neben dieser eine Auswahlbox erscheinen, in der die im Projekt verfügbaren Grammatiken sichtbar sind. Diese Auswahl sollte sich mit der Eingabe jedes Buchstaben verkleinern. Wenn die Grammatik eindeutig eingegeben ist, sollte nach Eingabe eines Punktes eine Liste der öffentlichen Regeln dieser Grammatik erscheinen, so dass eine Regel ausgewählt werden oder die Auswahl durch Eingabe weniger Anfangsbuchstaben eingeschränkt werden kann.

Die größte Wirkung im Bereich der Entwicklungsgeschwindigkeit sollte aber die automatische Vervollständigung im Bereich der Nichttermiale bieten. Hier sollte nach Eingabe einer spitzen Klammer eine Auswahl der Nichtterminale erscheinen, welche sich nach Eingabe der Anfangsbuchstaben verkleinert. Zusätzlich sollte in einer Box neben der Auswahlbox eine Box erscheinen, in der zu erkennen ist, wie die Regel, die gerade ausgewählt wurde, definiert ist. Hier sollte auch eine Hervorhebung des Textes stattfinden. Diese zusätzliche Informationsbox verschafft der entwickelnden Person einen schnellen Überblick über die Regeln, die diese Person benutzen möchte. So muss ein Nichtterminal nicht erst hingeschrieben werden, sondern es kann sofort gesehen werden, wie dieses definiert ist.

Diese Funktion soll erweiterbar sein, da beispielsweise in Tapas-Grammatiken eine andere Art von Vervollständingung von Nichtterminalen benötigt wird. Dort ist zudem eine Vervollständigung der semantischen *Tags* vonnöten.

Bei einem Tapas-Nichtterminal besteht der erste Teil aus der semantischen Definition, welche aus der Ontologie importiert wird. Es wäre gut, wenn dieser Teil nicht in einer Datei nachgeschlagen werden müsste, sondern wenn dieser in der Auswahlliste verfügbar wäre. Desweiteren sollten die in der Datei definierten Nichtterminale in dieser Auswahl verfügbar sein, da der Editor noch nicht weiß, ob es sich nicht um ein strukturelles Nichtterminal handelt. Wenn aber das erste Komma eingegeben wird, ist klar, dass ein Vektor-Nichtterminal referenziert wird. Nun ist der syntaktischen Teil der Regel zu bearbeiten. Hier könnte eine Liste der immer verwendeten syntaktischen Typen zur Auswahl stehen. Wenn dieser Typ ausgewählt ist, kann automatisch ein Komma eingefügt, die Regel automatisch mit einer geschlossenen spitzen Klammer abgeschlossen und der Cursor vor dieser platziert werden. Dann kann die benutzende Person den zweiten syntaktischen Typ angeben.

In Abbildung 3.7 sieht man die Benutzung dieser Textvervollständigungsfunktion beim Schreiben eines Nichtterminals.

3.8.1 Wege in Eclipse

In Eclipse muss für die Vervollständigung des Eingabetextes eine Klasse geschrieben werden, welche das *IContentAssistant*-Interface implementiert. Ein Objekt dieser Klasse kann für bestimmte Partitionen mit Objekten einer Klasse konfiguriert werden, welche das *IContentAssistProcessor*-Interface implementieren. Wenn der Cursor in einer bestimmten Partition ist und der Benutzer die Standard-Tastenkombination STRG+SPACE drückt oder ein vorher festgelegtes Zeichen eingegeben wird, so wird der der Partition zugeordnete, *IContentAssistProcessor* aufgerufen. Dort entscheidet die Methode *computeCompletionProposals* darüber, welche Vervollständigungsoptionen angezeigt werden.

Für das Fenster, welches die Definition der Nichtterminale anzeigt, muss eine Klasse geschrieben werden, welche das *IInformationControlCreator*-Interface implementiert. Diese Klasse ermöglicht Syntaxhervorhebungen in diesem Fenster.

3.8.2 Konkrete Umsetzung

Die Berechnung der Vervollständigungsoptionen erfolgt in mehreren Schritten. Die Implementierung des *IContentAssistProcessor* heißt *JSGFContentAssistantProcessor*. Zuerst muss, ausgehend von der Cursorposition, bestimmt werden, in welchem Teil der Grammatik sich der Cursor befindet. Kommentare können ganz ausgeschlossen werden, da der *JSGFContentAssistantProcessor* nicht für diesen Partitionstyp registriert ist.

Es wäre weiterhin möglich, dass der Cursor sich in einem semantischen Tag befindet, was sich durch eine Frage nach dem Partitionstyp an dieser Stelle an das *Dokument* klären lässt.

Durch Prüfung des Zeilenanfangs kann festgestellt werden, ob sich der Cursor im Import oder self identifying header befindet. Der Zeilenanfang ist in einen Fall "*import*" und im anderen Fall "♯*JSGF*".

Wenn diese Information bekannt ist, lässt sich aus dem Prefix des betreffenden Grammatik-Elementes ein Vorschlag errechnen. Dies wird im Fall der Nichtterminale durch eine Suche über die in der Grammatik verwendeten und in die Grammatik importierten Nichtterminale erreicht. Ähnlich funktioniert die Vervollständigung von Importdeklarationen. Lediglich für den self identifying header musste ein anderes Verfahren gewählt werden, bei dem, aufgrund der Anzahl der schon in der Zeile vorhandenen Wörter, einen Vorschlag zu berechnen wird.

3.9 Automatische Fehlererkennung

Es kostet viel Zeit, zum Entdecken von syntaktischen und semantischen Fehlern jedes Mal den Compiler zu starten. Dieses Vorgehen ist sehr ineffizient, da oft zwischen verschiedenen Programmen gewechselt werden muss. Ein Compilerlauf kann lange dauern und die Interpretation der Fehlermeldung und die Zuordnung zum eigentlichen Fehler kann schwierig sein. Da Menschen per Definition Fehler machen, könnte eine Funktion, die das Auffinden von diesen beim Schreiben der Grammatik ermöglicht, zu einer großen Beschleunigung des Entwicklungsprozesses führen.

Microsoft Word bietet bereits seit Jahren die Möglichkeit der automatischen Rechtschreibfehlererkennung. Dabei werden Fehler rot unterstrichen, wie es schon aus der Schulzeit bekannt ist. Diese vertraute Art der Fehlermarkierung soll auch im JSGF-Editor Einzug halten. Es sollen nicht nur syntaktische Fehler gefunden werden, sondern auch semantische. Dadurch soll die Anzahl der Übersetzerläufe minimiert werden, da nun sichtbar ist, ob eine Grammatik syntaktisch und semantisch richtig ist.

Um den Überblick über die vorhandenen Fehler zu behalten, gibt es eine *View*, welche wie die *Task-View* die Fehler sammelt und anwählbar macht.

In Abbildung 3.8 wird ein JSGF-Editor dargestellt, in dem fünf Fehler gefunden wurden. Diese werden durch Unterstreichungen unter der Fehlerstelle, Markierungen an der Seite und Einträge in einer Liste gekennzeichnet.

3.9.1 Wege in Eclipse

In Eclipse kommen zum Markieren der Fehler, wie in Abschnitt 3.7 beschrieben, Marker zum Einsatz. Um diese Marker zu erzeugen, läuft während der Eingabe ein Thread mit, welcher darauf wartet, dass es für längere Zeit keine Eingabe gibt. In diesem Fall ruft der Thread die

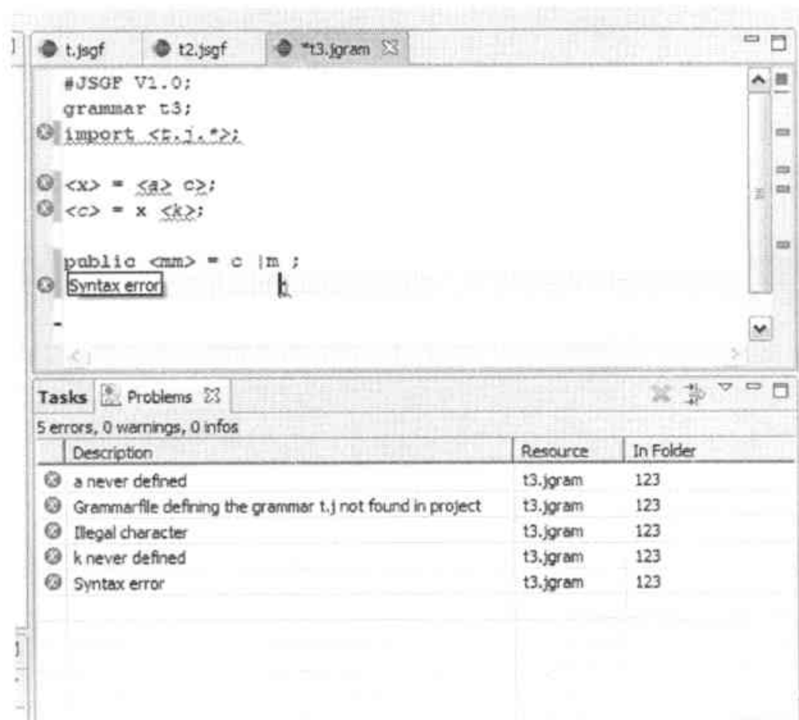

Abbildung 3.8: Erkannte Fehler

reconcile-Methode einer Klasse auf, welche das *IReconcilingStrategy*-Interface implementiert und im Framework über die *SourceViewerConfiguration* registriert ist. Dort kann dann die Fehlererkennung stattfinden.

Beim Beginn eines neuen Projektes sollen oft bestehende Grammatikdateien in das Projekt importiert werden.

Das Hinzufügen von neuen Grammatikdateien in ein Projekt kann diesen Mechanismus aushebeln, da die *ReconcilingStrategy* nur ausgewertet wird, wenn eine Datei verändert wird. Es wird dann also kein Grammtik-Modell für diese Datei berechnet. Das Hinzufügen einer neuen Datei, welche andere Dateien importiert, ist auch nicht möglich, da es ja noch kein Modell der anderen Dateien im Projekt gibt. Es wird demzufolge etwas gebraucht, das beim Start der Entwicklungsumgebung und beim Hinzufügen von Dateien einen konsistenten Zustand erzeugt.

Diese Komponente wird in Eclipse *Builder* genannt. Ein Builder wird eigentlich benötigt, um Dateien im Hintergrund, während der Eingabe, kompilieren zu lassen. Dabei kann man komplexe Abhängigkeiten berücksichtigen, um nur die nötigsten Dateien zu kompilieren. Da aber die Grammatik-Dateien nicht kompiliert werden, muss nur ein Modell erstellt werden, welches im Speicher bleibt. Der Builder ist im Falle des JSGF-Editors also dafür verantwort-

lich, dafür zu sorgen, dass es immer ein Modell zum ganzen Projekt gibt.

Eclipse kann mehrere Builder für ein Projekt verwalten, so dass in diesem Projekt nicht nur JSGF-Dateien, sondern z.B. auch Java-Dateien bearbeitet werden können. Builder werden in der Beschreibung des Projektes registriert, welche in einer XML-Datei liegt. Da der Benutzer diese Datei nicht von Hand editieren soll, kann man einem Projekt mehrere so genannte *Natures* zuordnen. Ein Projekt könnte so beispielsweise eine Java-Nature und eine JSGF-Nature haben. Eine Nature kümmert sich darum, dass einem Projekt die entsprechenden Module zugeordnet werden, die für das Projekt sinnvoll sind; so zum Beispiel, dass dem Projekt der richtige Builder zugeordnet wird. Wenn die Nature wieder vom Projekt entfernt wird, so ist diese auch für das Entfernen dieser Module verantwortlich.

3.9.2 Konkrete Umsetzung

Die Umsetzung dieser Funktion erstreckt sich über zwei Plugins. Die syntaktische Fehlererkennung übernimmt, wie in Abschnitt 3.2 beschrieben, der Parser. Dabei wurde darauf geachtet, dass in das Parser-Plugin kein Eclipse-Code Einzug findet. Die gefundenen Fehler werden in der Klasse *ErrorModel* als *SemanticSyntaxError* gespeichert. Das *ErrorModel* wird der *Grammar*-Klasse des Modells der Grammatik als Eigenschaft zugeordnet.

Nun kann es jedoch sein, dass, wie im Fall von Tapas, bei Nichtterminalen und Tags besondere Fehlererkennungen zum Einsatz kommen müssen. Darum muss das Fehlermodell in diesem Teil erweiterbar sein. Dies wird dadurch erreicht, dass die Objekte, welche Tokens und Tags im Grammatik-Modell repräsentieren, von einer Klasse nach dem Fabrik-Entwurfsmuster erzeugt werden. Diese Fabrik ist über eine Extension des Parser-Plugins austauschbar, so dass man in diesem Fall von einer *Abstrakten Fabrik* spricht.

Semantische Fehler werden im JSGF-Editor-Plugin gefunden, da hier Wissen über ein Projekt vorhanden sein muss, um beispielsweise Importkonflikte auflösen zu können. Die *Grammar*-Klasse ist nicht im JSGF-Editor-Plugin angesiedelt. Objekte dieser Klasse werden vom Parser erzeugt, um eine Repräsentation der Grammatik zu konstruieren. Da es zu keiner Vermischung des Parsers mit den Komponenten aus dem Eclipse-Framework kommen soll, wurde eine Klasse *CompilationUnit* eingeführt, welche als Attribut die *Grammar*-Klasse, eine Referenz auf die Grammatikdatei und eine Liste von Referenzen von referenzierten *CompilationUnits* hat.

Weiterhin ist es nötig, Fehlertypen für Erweiterungen des Editors nachzuladen und manchmal andere Fehlertypen von JSGF-Grammatiken zu deaktivieren. Dies ist dann sinnvoll, wenn die Erweiterung des Editors Grammatiken akzeptiert, die nicht ohne weiteres der Spezifikation der JSGF-Grammatiken entsprechen. Dies ist beispielsweise bei Tapas der Fall (siehe Abschnitt 2.1.2).

Darum wurde folgende Umsetzung gewählt: Es gibt eine Klasse *SemanticErrorManager*, an welcher sich Klassen, die das *IErrorType*-Interface implementieren, registrieren können. Diese Klassen untersuchen eine CompilationUnit auf bestimmte Fehler. Das *IErrorType*-Interface definiert eine Methode *findError(CompilationUnit compUnit)*, welche vom *SemanticErrorManager* in einer Schleife aufgerufen wird, in der über alle registrierten Fehlerarten iteriert wird. Die semantischen Fehler werden zusammen mit den syntaktischen Fehlern in der *Grammar*-Klasse gespeichert.

Der *SemanticErrorManager* ist durch eine Klasse, die von ihm erbt, an einem Extension Point erweiterbar. Damit kann für andere Editortypen eingestellt werden, welche semantischen Fehler erkannt werden. Weiterhin können so weitere Fehlertypen nachgerüstet werden.

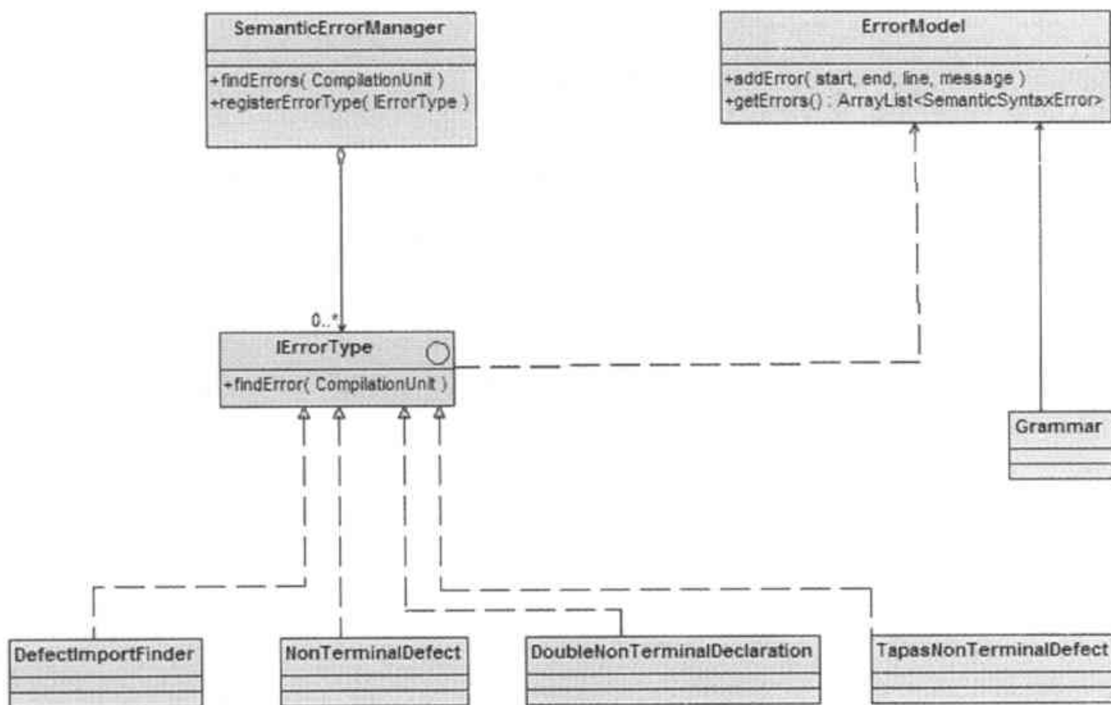

Abbildung 3.9: UML-Diagramm Fehler

Das Diagramm 3.9 zeigt die Struktur der Fehlerarchitektur in einer Übersicht.

3.10 Automatische Fehlerbereinigung

Um die Geschwindigkeit in der Entwicklung noch weiter zu erhöhen, ist es nicht nur nötig, Fehler zu entdecken, sondern auch für bekannte Fehlertypen einen Lösungsvorschlag zu generieren, den der Benutzer mit einem Tastenanschlag umsetzen kann.

Dieses Konzept ist aus dem JDT entlehnt. Dort werden durch Drücken der Tastenkombination *STRG+1*, mit Cursor über einem Fehler, Vorschläge für die Korrektur des Fehlers berechnet. Dabei erscheint, genauso wie dies bei der Textvervollständigung passiert, eine Box, welche die Vorschläge präsentiert. Nach der Wahl eines Vorschlages wird dieser umgesetzt.

Diese Funktion ist nicht nur hilfreich bei der Beseitigung von Fehlern. Sie wird von vielen Entwicklern gezielt genutzt, um sich Tipparbeit zu ersparen. So werden zum Beispiel oft in Methodenrümpfen Methoden aufgerufen, welche noch nicht existieren. Diese werden selbstverständlich als Fehler markiert. Nun reichen zwei Tastenanschläge, um einen leeren Methodenrumpf für die jeweils fehlende Methode zu generieren.

Dieses Verfahren zur Entwicklungsbeschleunigung ist auch im JSGF-Editor nutzbar. Fehler können dann nicht nur automatisch repariert werden. Dieser Automatismus kann genutzt werden, um durch die gezielte Produktion von Fehlern Tastenanschläge zu sparen.

3.10.1 Wege in Eclipse

Eclipse nennt diese Art der automatischen Fehlerkorrektur *Quickfix*. Diese Quickfixes werden durch Implementieren des *IMarkerResolution*- Interfaces zur Verfügung gestellt. Implementierungsklassen müssen die *run*-Methode implementieren, welche die Reparatur der fehlerhaften Stelle vornimmt.

Verwaltet werden Quickfixes von einem *IMarkerResolutionGenerator*, welcher auf Anfrage mit einem *IMarker* als Parameter ein Array von *IMarkerResolution* zurückgibt.

Mit diesen Klassen kann man Quickfixes schon in soweit implementieren, dass in der Fehlerübersicht Fehler mit der rechten Maustaste anklickbar werden. Nach der Wahl der *QuickFix*-Funktion werden die verfügbaren Lösungsvorschläge präsentiert. Wenn der Benutzer einen Lösungsvorschlag gewählt hat, wird die *run*- Methode des Quickfixes gestartet und der Fehler beseitigt.

Sollen die Fehlervorschläge hingegen aus dem Editorfenster heraus mit *STRG+1* aufrufbar sein, so muss ein *SourceViewer* implementiert werden. Dieser gibt bei dieser Eingabe von *STRG+1* einen *ContentAssistant* zurück, welcher die Quickfixes präsentiert.

3.10.2 Konkrete Umsetzung

Die Umsetzung ist im UML-Diagramm 3.10 dargestellt und wird hier erläutert. Das *IMarkerResolution* Interface wird durch das *IQuickFix*- Interface erweitert. Dieses fügt noch zwei Methoden[6] hinzu, welche das Setzen des Cursors nach der Anwendung der Quickfixes

[6] *getCursorPosition()* und *getSelectionLength()*

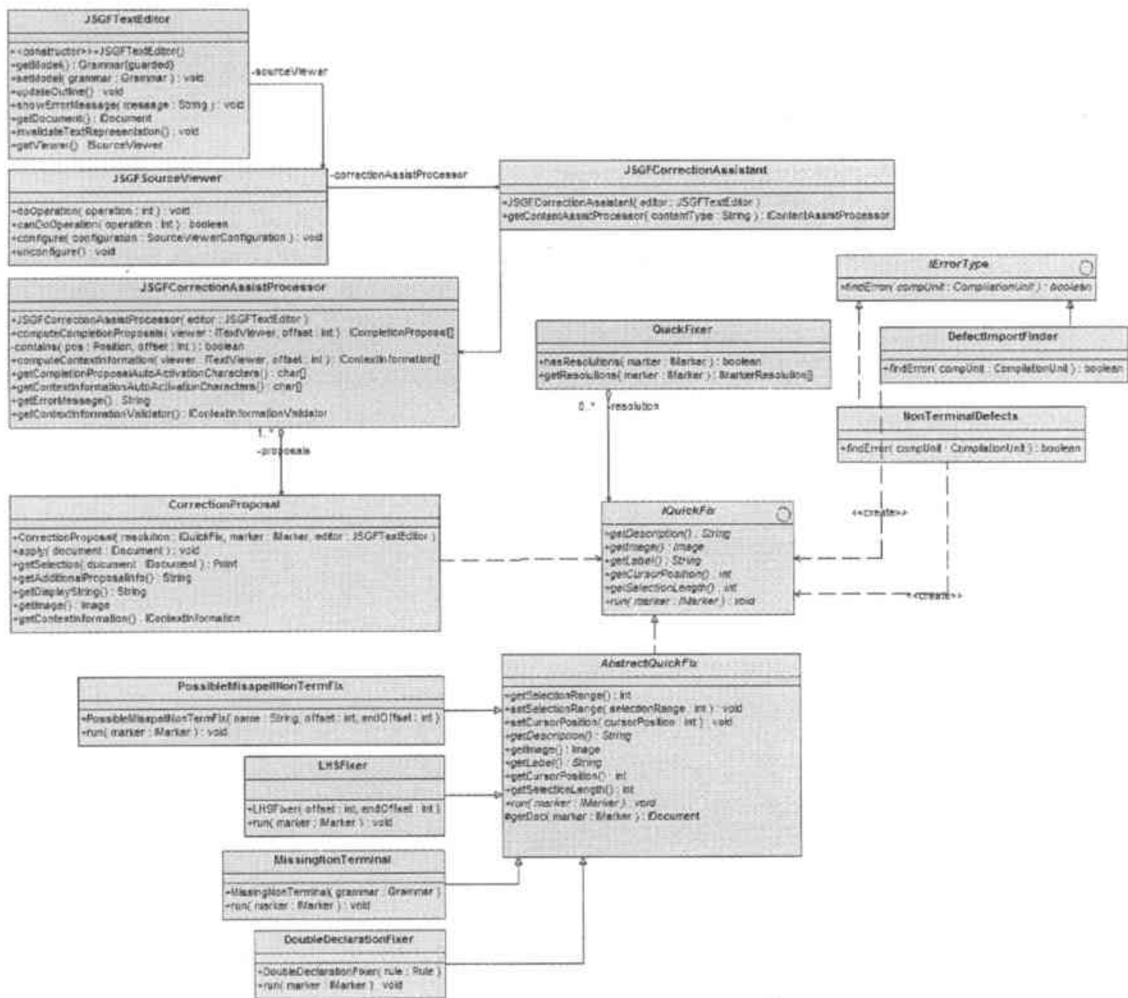

Abbildung 3.10: UML-Diagramm der Fehlerbereinigungsarchitektur

ermöglichen. Alle Lösungen also, welche zu bestimmten Fehlern implementiert werden sollen, müssen das *IQuickFix*-Interface implementieren. Die abstrakte *AbstractQuickFix*-Klasse stellt eine Implementierung der wichtigsten Methoden dar und bietet zudem leichten Zugriff auf ein Dokument, welches zu einem *IMarker* gehört. Die Quickfixes werden innerhalb der fehlererkennenden Klassen, passend zum Fehler, erzeugt. Insgesamt sind in der Studienarbeit Lösungen zu sechs verschiedenen Fehlerarten (siehe Tabelle 3.1) implementiert worden.

Die *IQuickFix*-Objekte werden beim Erstellen der Fehler von den jeweiligen Klassen erzeugt und dem Fehlertyp *SemanticError* zugeordnet. Wenn nun *IMarker* erzeugt werden, um Fehler darzustellen, so wird den *IMarker*n, zu welchen es eine Lösung gibt, diese als Attribut mitgegeben.

Fehlerart	**angebotene Lösung**	Implementierungs-Klasse
Nichtterminal nicht definiert	automatisch neues Nichtterminal erstellen	*MissingNonTerminal*
Nichtterminal nicht definiert	Generierung von Vorschlägen zur Änderung des Nichtterminals zu einem Nichtterminal, welches ähnlich im Sinne der Levenstein-Distanz[[illegible]] ist	*PossibleMisspellNonTermFix*
Doppelte Definition eines Nichtterminals	Löschen der doppelten Definition, welche sich unter dem Cursor befindet	*DoubleDeclarationFixer*
Minor Syntax Type vergessen	Vorschläge zum Ergänzen des Minor Syntax Type	*MajorMinorQuickFix*
Major Syntax Type vergessen	Vorschläge zum Ergänzen des Major Syntax Type	*MajorMinorQuickFix*
Nichtterminal auf linker Seite einer Regel mit Paketinformationen	automatisches Löschen der Paketinformationen	*LHSFixer*

Tabelle 3.1: Fehlerarten/-lösungen

Wenn der Benutzer *STRG+1* drückt, so aktiviert der *JSGFSourceViewer* den *JSGFCorrectionAssistant*, welcher dann vom *JSGFCorrectionAssistProcessor* die *CorrectionProposals* bekommt. Dies funktioniert so ähnlich wie bei der Textvervollständigung (Abschnitt 3.8). Lediglich die Erzeugung der Korrekturvorschläge verläuft anders als die Erzeugung der Textvervollständiungsvorschläge.

Der *JSGFCorrectionAssistProcessor* verschafft sich Zugang zur *MarkerHelpRegistry*, wo er mit einem *IMarker* durch die Methode *getResolutions* Lösungsvorschläge bekommen kann. Diese werden dann mit Objekten der Klasse *CorrectionProposal* verpackt, welche vom *JSGFCorrectionAssistant* als Eingabe akzeptiert werden.

Die Klasse *MarkerHelpRegistry* bedient sich der Klasse *QuickFixer*, um Lösungsvorschläge zu bekommen. Diese Klasse ist eine Implementierung des *IMarkerResolutionGenerator*- Interfaces und entpackt die Lösungen aus dem *IMarker*.

Kapitel 4

Zusammenfassung und Ausblick

Dieses Kapitel soll zusammenfassen, was erreicht wurde und einen Ausblick auf Arbeiten geben, welche den Editor weiter verbessern können.

4.1 Zusammenfassung

In dieser Studienarbeit ist ein erweiterbarer Editor entstanden, welcher die Erstellung von Grammatiken in hohem Maße beschleunigen kann. Dies wurde erreicht durch die Nutzung des etablierten Eclipse-Frameworks.

Funktionen, welche die Übersicht erhöhen, wurden implementiert. Dies sind das Syntax-Highlighting, das Anzeigen von Regeldefinitionen und die Outline. Durch eine Erhöhung der Übersicht kann sich der Anwender schneller in großen Grammatiken zurechtfinden.

Die Navigation innerhalb der Grammatiken wurde durch folgende Funktionen verbessert: Sprung zur Definition, Aufgabenliste, Liste mit anwählbaren Fehlern und Outline. Das schnelle Gelangen zu den gewünschten Stellen in der Grammatik beschleunigt das Suchen, so dass sich der Entwickler auf seine eigentliche Aufgabe konzentrieren kann.

Schließlich wurden Funktionen zur Fehlervermeidung und Fehlerbeseitigung erstellt. Dies ist zum einen die automatische Textvervollständigung. Mit ihr ist es möglich, das Ausschreiben von langen Nichtterminalnamen zu beschleunigen. Dadurch wird zudem die Häufigkeit von Tippfehler verkleinert, da der Anwender weniger tippen muss. Die Fehlererkennungsfunktion erkennt sowohl syntaktische, als auch semantische Fehler beim Schreiben. Dies hilft dem Entwickler, Fehler vor der Ausführung des Dialog-Managers zu erkennen. Diese Fehler können dann komfortabel mit der Fehlerkorrekturfunktion entfernt werden. Hierbei werden dem Entwickler Fehlerkorrekturvorschläge passend zum erkannten Fehler angeboten. Diese werden dann mit wenigen Tastendrücken automatisch umgesetzt.

Die Aufteilung des Editor in mehrere Plugins erhöht die Wiederverwendbarkeit der ein-

zelnen Komponenten und erleichtert die Erweiterbarkeit. Dadurch ist es in Zukunft leicht möglich, den Editor zu pflegen und durch weitere Arbeiten zu verbessern.

4.2 Ausblick

Im Rahmen der Studienarbeit konnten durch den begrenzten Zeitrahmen nicht alle denkbaren Möglichkeiten für die Unterstützung des Benutzers bei der Entwicklung von Grammatiken ausgeschöpft werden. In manchen Bereichen müssen erst Benutzererfahrungen darüber gesammelt werden, welche weiteren Funktionen überhaupt sinnvoll sind.

Die Fehlererkennung und Fehlerbeseitigung bei Tapas-Grammatiken ist noch ausbaubar. Hier können sowohl mehr Fehlertypen zum Entdecken implementiert werden, als auch mehr Lösungsvorschläge zu diesen Fehlern entwickelt werden.

Die Eingabevervollständigung kann so gestaltet werden, dass sie den Kontext stärker einbezieht, so dass vor allem in Tapas-Grammatiken passendere Vorschläge generiert werden.

Weiterhin könnte an einer Evaluation gearbeitet werden, welche die aufgestellte These untermauert, dass diese Entwicklungsumgebung die Entwicklung von Grammatiken erleichtert und beschleunigt.

Der eingesetzte Zerteiler *Cup* kann durch einen leichter wartbaren Parser ersetzt werden. Für Parser wie *antlr* existieren Eclipse-Plugins, welche das Bearbeiten der Parser-Grammatiken erleichtern. Weiterhin ist die Ausgabe von Parser-Fehlern verbesserbar. Cup unterstützt keine automatischen Fehlermeldungen, welche eine Aussage darüber treffen, welches Token als nächstes erwartet wird. Diese Fehlermeldungen könnten dem Benutzer beim Verstehen der Fehler, die er produziert hat, helfen.

Anhang A

Abkürzungen

BNF	Backus-Naur-Form
EBNF	die erweiterte BNF
JDT	Java Development Tools
SWT	Standard Widget Toolkit
JAR	Java Archive
UML	Unified Modeling Language
JSGF	Java Speech Grammar Format
XML	Extensible Markup Language

Anhang B

Handbuch

Der JSGF-Editor und der Tapas-Editor stellen Funktionen wie Syntax-Highlighting, automatische Textvervollständigung, automatische Fehlererkennung und -beseitigung zur Beschleunigung der Grammatikentwicklung zur Verfügung. Abbildung B.1 bietet einen Eindruck über die Möglichkeiten der Editoren. Die Installation der Editoren und die Funktionen werden in diesem Handbuch erläutert.

B.1 Installation

Zum Betrieb der Editoren ist eine Eclipse-Umgebung in der Version 3.1[1] und Java[2] in der Version 1.5 erforderlich. Wenn diese Software installiert wurde, sind die gelieferten JAR-Dateien[3] in das Plugin-Verzeichnis zu kopieren. Das Plugin-Verzeichnis hat den Namen *plugins* und befindet sich im Eclipse-Hauptverzeichnis. Nach dem Neustart von Eclipse[4] stehen die neuen Editoren zur Verfügung.

B.2 Anlegen eines JSGF-Projektes

Zur Nutzung des JSGF-Editors ist ein Projekt anzulegen. Dies wird über das Eclipse-Menü gemacht: *File→New→Project*.

Im folgenden Dialog ist nur noch der Name für das Projekt zu wählen. Das angelegte Projekt ist nun um eine JSGF-Nature zu erweitern. Dazu öffnet ein Rechtsklick auf das

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

[2] `http://java.sun.com`

[3] JSGFEditor_0.9.1.002.jar, tapas.util.jsgf.parser_1.0.2.002.jar, tapasEditor_1.0.2.jar, de.uka.ira.isl.tadeus.core_0.9.1.jar

[4] evtl. muss Eclipse einmalig nach der Installation der Plugins von der Kommandozeile aus folgendermaßen gestartet werden: eclipse.exe -clean

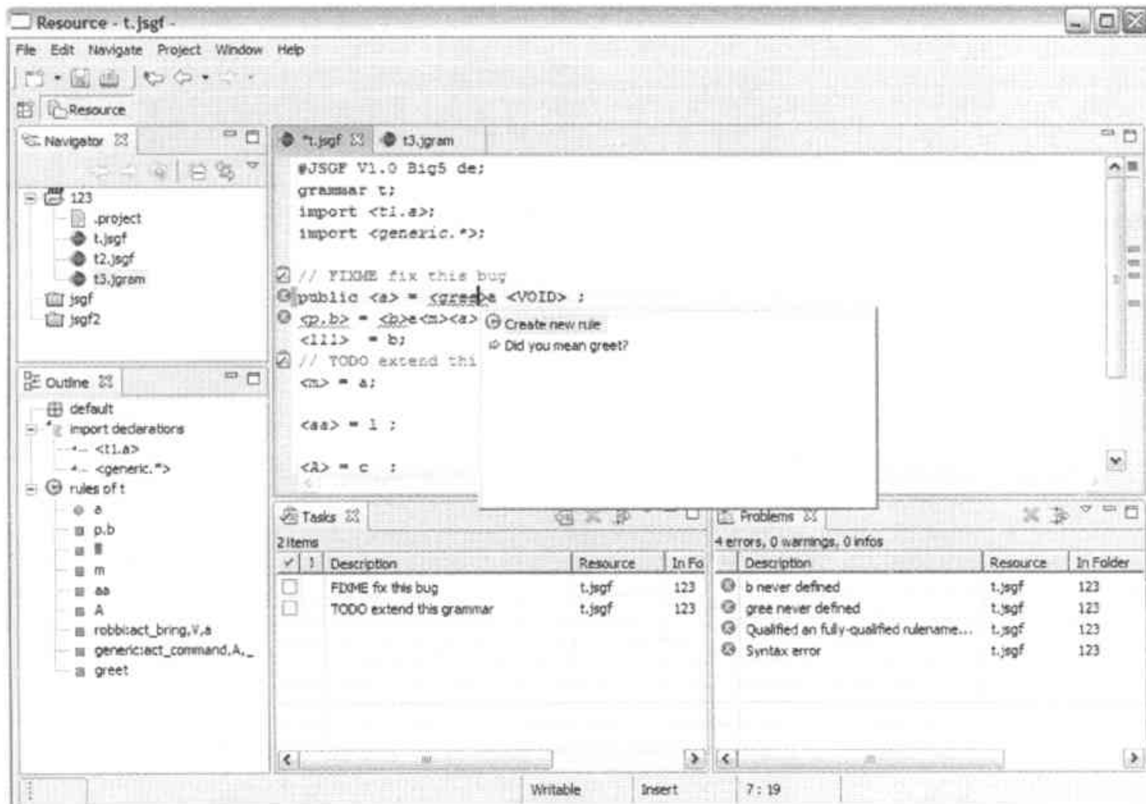

Abbildung B.1: Syntax-Highlighting, Outline, Aufgabenliste, automatische Fehlererkennung / -beseitigung, Fehlerübersicht

Projekt-Symbol im *Navigator* ein Kontextmenü, aus dem *Toggle-JSGF-Nature* ausgewählt werden muss.

Nun ist die Entwicklungsumgebung bereit, um JSGF-Dateien zu bearbeiten. Um eine JSGF-Datei anzulegen ist *File→New→File* aus dem Eclipse-Menü zu wählen. Es folgt die Wahl des Dateinamens, der als Suffix *.jsfg* oder *.jgram* haben muss. Im Folgenden öffnet sich ein Editor, in dem die Entwicklung beginnen kann.

B.3 Anlegen eines Tapas-Projektes

Zum Anlegen eines Tapas-Projektes muss zuerst ein Projekt angelegt werden, wie in Abschnitt B.1 beschrieben. Dann muss dem Projekt eine Tapas-Nature[5] hinzugefügt werden. Dies wird über das Eclipse-Menü gemacht: *File→New→Project*. Dort ist *Tapas→Tapas Pro-*

[5] eine Nature gibt die Art eines Projektes an. So gibt es beispielsweise eine Java-Nature oder eine JSGF-Nature

ject auszuwählen. Im folgenden muss *Next* und schließlich *Finish* angeklickt werden. Dann wird im *Navigator* mit der rechten Maustasten auf das neu entstandene Projekt geklickt und hier *Properties* ausgewählt. Nun wird im linken Teil des Fensters *Tapas* ausgewählt, um im rechten Teil die Position der Tapas-Properties-Datei festzulegen. Dieser Teil befindet sich noch in Entwicklung, weshalb sich die Art der Erzeugung von Tapas-Projekten ändern kann.

B.4 Funktionen

Der Editor bietet verschiedene Funktionen, um den Entwickler zu unterstützen.

Outline

Die Outline bietet eine Übersicht über die gerade bearbeitete Grammatik durch Informationsreduktion. Es werden die Namen der definierten Regeln und Imports angezeigt. Diese sind anklickbar, woraufhin der Editor an die entsprechende Stelle in der Grammatik springt.

Die Outline kann wie jede andere *View*[6] über *Window→Show View→Other→Outline* geöffnet werden.

Sprung zur Definition

Wenn zu der Definition eines Nichtterminals, welches sich unter dem Cursor befindet, gesprungen werden soll, so ist F3 zu drücken. Soll im Anschluss wieder zurück gesprungen werden, so geht das mit der Tastenkombination ALT+Links. Dies funktioniert, ähnlich wie in den meisten Internet-Browsern, mehrfach und mit ALT+Rechts auch in die andere Richtung.

Anzeigen der Definition

Soll die Definition eines Nichtterminals angezeigt werden, so ist der Mauszeiger über dieses zu führen, woraufhin eine Box neben dem Mauszeiger erscheint, welche die Definition enthält.

Aufgabenliste

Um sich Aufgaben zu merken und diese dateiübergreifend in einer zentralen Liste verfügbar zu machen, ist ein Kommentar zu schreiben. Dieser muss mit einem Schlüsselwort beginnen. Diese Schlüsselwörter sind, wie in Abbildung B.2 dargestellt, in den Eclipse-Optionen konfigurierbar (*Window→Preferences...*).

[6]Views werden die Fenster innerhalb von Eclipse genannt, die nicht der Editor sind

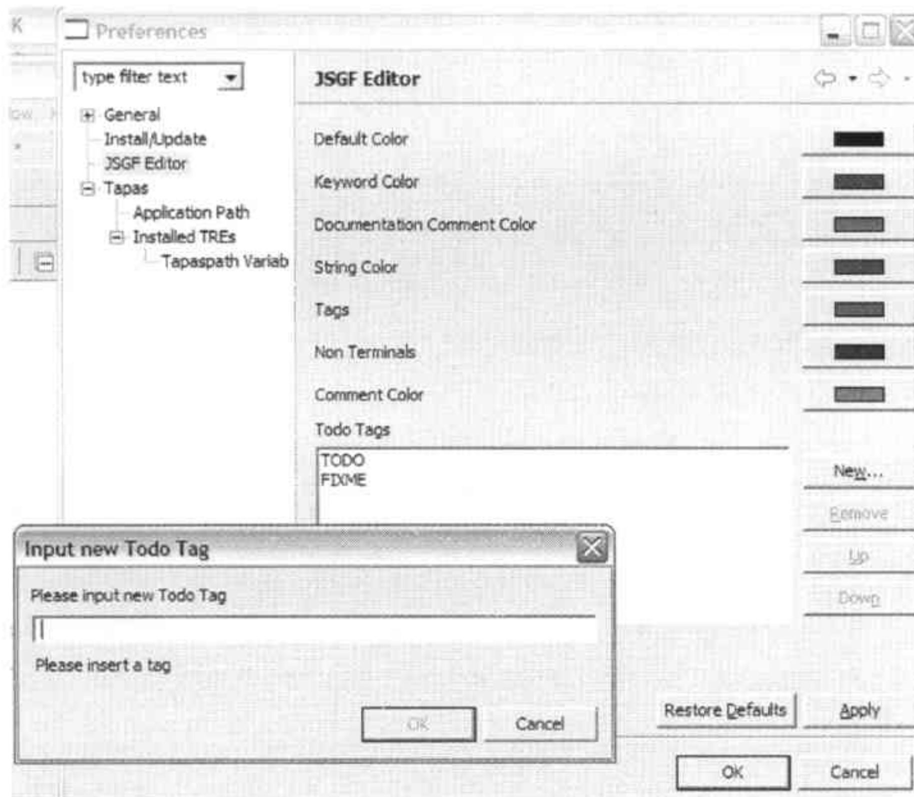

Abbildung B.2: Konfiguration der Schlüsselwörter für die Aufgabenübersicht und der Farben für das Syntax-Highlighting

Jedes Zeichen, welches hinter dem Schlüsselwort erscheint, wird in die Übersicht mit aufgenommen. Wenn auf einen Eintrag in der Übersicht geklickt wird, dann öffnet sich das dazugehörige Editorfenster und der Editor springt an die entsprechende Stelle, an der der Eintrag für die Aufgabe ist.

Textvervollständigung

Die Textvervollständigung wird durch das Drücken der Tastenkombination STRG+Leertaste an einer zu vervollständigenden Stelle ausgelöst. Sind bereits Buchstaben geschrieben, so wird versucht, den Vorschlag für die Textvervollständigung durch das existierende Prefix einzugrenzen. Mit den Pfeiltasten kann ein Eintrag ausgesucht und schließlich mit *Return* auswählt werden.

Automatische Fehlererkennung

Fehler werden automatisch erkannt und rot unterstrichen. Um die Fehlerursache zu erfahren, kann mit dem Mauszeiger über der Fehlerstelle oder am Rand über dem roten Kreis mit dem weißen Kreuz in der Mitte verharrt werden, so dass eine Box mit der Ursache des Fehlers als Inhalt erscheint.

Automatische Fehlerbereinigung

Für viele Fehler existiert eine automatische Fehlerbereinigung, welche, wenn sich der Cursor über einem erkannten Fehler befindet, mit STRG+1 aufgerufen werden kann. Es erscheint dann eine Box mit den Vorschlägen zur Beseitigung des Fehlers. Wird ein Vorschlag mit der Maus oder den Pfeiltasten ausgewählt, so wird der Vorschlag umgesetzt und der Fehler bereinigt.

B.5 Tastenbelegungsübersicht

Tastenkombination	Wirkung
STRG+1	Anzeigen der Lösungsvorschläge für Fehler
STRG+Leertaste	Anzeigen der Vorschläge für die Textvervollständigung
F3	Sprung zur Definition eines Nichtterminals, wenn sich der Cursor über einem Nichtterminal befindet
ALT+Links	Sprung zur letzten Position, von der aus das letzte Mal gesprungen wurde
ALT+Rechts	Sprung zur nächsten Position, zur der schonmal gesprungen wurde

Tabelle B.1: Tastenbelegungsübersicht

Literaturverzeichnis

[1] Sun Microsystems. Netbeans-Website. http://www.netbeans.org. 1.2

[2] IBM. Eclipse-Website. http://www.eclipse.org. 1.2

[3] Lutz Erk, Katrin und Priese. *Theoretische Informatik.* Springer Verlag, 2002. 2.1

[4] Gerhard Goos. *Vorlesungen über Informatik Band 3: Berechenbarkeit, formale Sprachen und Spezifikationen.* Springer, Jan 1997. 2.1

[5] Speech Works International Andrew Hunt. JSGF-Definition. http://www.w3.org/TR/jsgf/. 2.1.1

[6] Guy Steele und Gilad Bracha James Gosling, Bill Joy. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000. 2.1.1

[7] M. Denecke. Object-oriented techniques in grammar and ontology specification, in the workshop on multilingual speech communication. 2000. kyoto, japan: pp. 59-64., 2000. 2.1.2

[8] International Business Machines (IBM). Common Public License. http://www.eclipse.org/legal/cpl-v10.html. 2.2

[9] Azad Bolour. Notes on the Eclipse Plug-in Architecture. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin.architecture.html. 2.2.2

[10] Eclipse Foundation. Naming Conventions. http://dev.eclipse.org/naming.html. 3.1

[11] Gerwin Klein. Website JFlex. http://jflex.de/. 3.2

[12] Gerhard Goos und William Waite. *Compiler Construction.* Springer, Jan 1984. 3.2

[13] Andrew W. Appel. Website Cup. http://www2.cs.tum.edu/projects/cup/. 3.2

[14] Richard und Johnson Ralph Gamma, Erich und Helm. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. GAM e 95:1 1.Ex. 2, 3.5.1, 5

[15] Markus Keller und Christof Marti Tom Eicher. Raffinierte Rezepte - Neue Funktionen für Ihren Eclipse 3.0-Texteditor. `http://www.eclipse-magazin.de/itr/online_artikel/psecom,id,647,nodeid,230.html`. 3.3.1

[16] wikipedia. Levenshtein distance. `http://en.wikipedia.org/wiki/Levenstein_Distance`. 3.10.2