

Bachelorarbeit

**Eclipse-Plugin zur optimierten
Erstellung und Wartung von
Grammatiken für das
Spracherkennungssystem
*ESMERALDA***

von

HENDRIK TER HORST

vorgelegt bei

M. Sc. Christian Munier

M. Sc. Leon Ziegler

im Studiengang Kognitive Informatik

der Technischen Fakultät

der Universität Bielefeld

Bielefeld, November 2012

Abstract

Speech recognizers nowadays play an increasingly important role in our daily lives. In order to reduce the complexity of natural language in a speech recognition process and to achieve significantly better results, a problem-specified grammar is used. Also the speech recognizer ESMERALDA which was designed at the University of Bielefeld makes use of this technique. In this Bachelor thesis an Eclipse-Plugin has been developed which aims to simplify the creation and maintenance of a grammar. A subsequent study showed that the use of the developed plugin simplifies the use of grammar in many terms.

Abstract

Spracherkenner spielen heutzutage eine immer wichtigere Rolle in unserem Alltag. Um die Komplexität der natürlichen Sprache in einem Spracherkennungsprozess zu verringern und deutlich bessere Ergebnisse zu erzielen, wird eine problemspezifizierte Grammatik verwendet. Auch der an der Universität Bielefeld entworfene ESMERALDA-Spracherkenner arbeitet mit dieser Technik. In dieser Bachelorarbeit wurde ein Eclipse-Plugin entwickelt, welches das Erstellen und Warten einer Grammatik erleichtern soll. In einer anschließenden Studie konnte gezeigt werden, dass der Einsatz des entwickelten Plugin den Umgang mit der Grammatik in vielen Hinsichten vereinfacht.

Inhalt

1	Einleitung	1
1.1	Motivation und Zielstellung	1
1.2	Leitfaden	3
2	Grundlagen	4
2.1	ISR-Grammatik	4
2.1.1	Aufbau und Struktur	4
2.1.2	Besonderheiten und Zusatzsymbole	5
2.1.3	Beispiel	6
2.2	Anforderungen an die Software	7
2.3	Anwendungsfall	8
2.4	Anforderungsanalyse	9
2.5	Verwandte Arbeiten	11
2.6	Benutze Technologien	13
3	Umsetzung	15
3.1	Funktionalitäten	15
3.1.1	Syntaxhighlighting	15
3.1.2	Automatische Formatierung	16
3.1.3	Fehlermarkierung	17
3.1.4	Hoverinformation	17
3.1.5	Content Assistent	18
3.1.6	Hyperlinks	19
3.1.7	Visualisieren von Regeln	20
3.1.8	Wortproblem und Syntaxbäume	21
3.2	Pluginarchitektur	22

3.3	Der ISR-Grammatik-Editor	25
3.4	Implementierung und Abläufe	28
3.4.1	Funktionalität über Extension Points	28
3.4.2	Funktionalität über Aktionen	34
3.4.3	Funktionalität über direkte Implementierung	37
4	Evaluation	40
4.1	Ziele	40
4.2	Methoden und Durchführung	41
4.3	Auswertung	42
4.3.1	Korrektur einer fehlerhaften Grammatik	43
4.3.2	Erstellen einer ISR-Grammatik	46
4.3.3	Lösen des Wortproblems und Erstellen von Syntaxbäumen	49
4.3.4	Funktionalitäten	53
4.4	Verbesserungsmöglichkeiten	57
5	Fazit und Ausblick	58
	Anhang	61
	Literatur	67

1 Einleitung

1.1 Motivation und Zielstellung

Spracherkennung spielt heutzutage in unserem Umfeld eine immer größere Rolle. Sei es bei automatisierten Telefonhotlines oder bei der Steuerung eines modernen mobilen Devices. Mit zunehmender Bedeutung der Robotik hat die Spracherkennung auch dort mittlerweile einen hohen Grad an Notwendigkeit erreicht. Als Teilgebiet der angewandten Informatik, beschäftigt sich die Spracherkennung mit der automatisierten Erfassung von gesprochener Sprache. Natürliche Sprache automatisiert zu erfassen, stellt jedoch weiterhin ein großes Problem dar, dessen Herausforderung vor allem in der Komplexität ihrer Struktur und Mehrdeutigkeit begründet ist.

Ein viel versprechender Ansatz dieses Problem zu lösen, besteht darin, den Spracherkenner mit einer spezifizierten Grammatik zu verknüpfen, der diese Komplexität einschränkt. Der, an der Universität Bielefeld entworfene Spracherkenner *ISR*¹, welcher auf dem Mustererkennungsframework *ESMERALDA*²[1] basiert, versucht dieses Problem unter Verwendung der ISR-Grammatik zu reduzieren. Um die Spracherkennung zu optimieren, wird eine Grammatik erstellt, die von vornherein nur bestimmte Satzstrukturen zulässt. Die erfassten Daten eines Sprechers brauchen daher nicht mehr gegen den gesamten Sprachumfang geprüft werden, sondern nur gegenüber den, in einer Grammatik beschriebenen Regeln. Dies bedeutet implizit, dass zur Erkennung komplexer Satzmuster eine komplexe Grammatik zugrunde liegen muss.

In dieser Bachelorarbeit wird das Problem des Erstellens komplexer Grammati-

¹IncrementalSpeechRecognizer: Inkrementelle Spracherkenner

² *Environment for Statistical Model Estimation and Recognition on Arbitrary Linear Data Arrays*

ken behandelt. Eine wesentliche Motivation an dieser Aufgabe zu arbeiten, resultiert aus dem vielschichtigen Anwendungsfall des Robocup@Home³-Wettkampfes. In diesen Wettkämpfen muss ein Roboter der Universität Bielefeld autonom Aufgaben erledigen, die ihm über den ESMERALDA-Spracherkenner zugänglich gemacht werden. Dabei ist die Erstellung komplexer und die Wartung unbekannter ISR-Grammatiken schwierig. Der Gebrauch eines Standard-Texteditors wie *gedit*⁴ (*GNOME Editor*) oder *vim*⁵ (*VI Improved*) ist in der Praxis nicht sinnvoll, da er keine ausreichende Unterstützung, zum Beispiel bei der Syntaxprüfung der zu erstellenden Grammatik, bietet. Da der Prozess des Kompilierens, währenddessen auch die Prüfung der syntaktischen Korrektheit der Grammatik stattfindet, viel Zeit braucht und über dies selten eine sinnvoll interpretierbare Fehlermeldung ausgibt, bietet sich eine Prüfung der Syntax schon direkt bei der Erstellung einer Grammatik an. Des Weiteren ist das Lösen des *Wortproblems*⁶, selbst bei einer übersichtlichen Grammatik, ohne weitere Hilfsmittel sehr aufwändig.

Im Anwendungsfall des Robocups stehen die Anwender oft unter massivem Zeitdruck. Dies erfordert bei der Lösung der Aufgaben über einen längeren Zeitraum eine hohe Konzentration, die gleichzeitig nicht effizient genug für andere Problemlösungen genutzt werden kann. Durch den Einsatz eines Editors, der für die Lösung besonderer Aufgaben spezifiziert wurde, soll die Fehleranfälligkeit reduziert und damit dem Zeitverlust entgegen gewirkt werden.

Da der ESMERALDA-Spracherkenner an der Universität Bielefeld konzipiert wurde und zurzeit nur in bestimmten Einsatzbereichen genutzt wird, zum Beispiel beim Robocup, wurde bisher kein Editor zur effektiven Erstellung und Wartung von ISR-Grammatiken entwickelt. Die auf dem Markt zur Verfügung stehenden Editoren, unabhängig davon, ob als Onlineservice oder als Software, die sich mit dem Thema der kontextfreien Grammatik beschäftigen, sind nicht kompatibel, mit der ISR-Grammatik, die durch eine spezifische Notation, Sonderzeichen und Sonderregeln definiert ist.

Die Idee der Implementierung eines eigenen Editors liegt also nahe und ist sinn-

³<http://www.ai.rug.nl/robocupathome/>

⁴<http://projects.gnome.org/gedit/>

⁵<http://www.vim.org/>

⁶Als Wortproblem bezeichnet man das Problem entscheiden zu können, ob ein gegebenes Wort durch eine Grammatik erstellt werden kann oder nicht.

voll. Dabei ist es jedoch nicht zwingend notwendig, eine StandAlone-Software zu entwickeln, da einige Editor-Frameworks bereits durch selbst geschriebene Plugins so erweitert werden können, dass eine angemessene Problemlösung möglich ist.

Das Ziel dieser Bachelorarbeit ist es, auf Basis eines Eclipse Plugin, einen Editor zu entwickeln, der den Umgang mit der ISR-Grammatik, sowohl für Anfänger als auch für Fortgeschrittene, vereinfacht und effizienter macht.

1.2 Leitfaden

Der Aufbau der Bachelorarbeit gliedert sich in fünf Kapiteln.

Kapitel Eins erläutert die Motivation und die Zielstellung.

Das zweite Kapitel zeigt die theoretischen Grundlagen der ISR-Grammatik auf und gibt erste Einblicke in die Art der Softwaremodellierung. Zunächst wird auf den Aufbau und die Struktur der ISR-Grammatik eingegangen und dabei ihre besonderen Eigenschaften erklärt. Des Weiteren werden Anforderungen an die Software erstellt, die den Umgang mit der ISR-Grammatik erleichtern sollen. Eine mögliche Problemstellung wird in der Anwendergeschichte dargestellt. Anschließend werden die Anforderungen an die Software analysiert.

Im dritten Kapitel werden anhand der im zweiten Kapitel entworfenen Anforderungen, die Definition der notwendigen Funktionalitäten die Softwarespezifikation erstellt. Ein weiterer Schwerpunkt liegt auf der Beschreibung der Architektur und der Implementierung der Software.

Zur Evaluierung der im dritten Kapitel erstellten Software wurde eine Benutzerstudie durchgeführt. Darin wurde der Frage nachgegangen, ob die Software zu allen aufgestellten Anforderungen und Problemen eine Lösung bietet. Die Ergebnisse und die Auswertung der Studie werden im vierten Kapitel präsentiert und kritisch diskutiert.

Das letzte Kapitel bildet die Zusammenfassung der Arbeit und gibt einen Ausblick auf mögliche Erweiterungen der Funktionalität des Editors, die zum Teil aus weiteren Vorüberlegungen resultieren und zum anderen Teil aus den Erkenntnissen und den Ergebnissen der Benutzerstudie hervorgegangen sind.

2 Grundlagen

2.1 ISR-Grammatik

Die ISR-Grammatik ist eine Sammlung von kontextfreien Regeln, welche einen Ausschnitt einer natürlichen Sprache repräsentiert. Ihren Einsatz findet sie in dem an der Universität Bielefeld entwickelten ESMERALDA-Spracherkenner. Sie dient dazu, unbrauchbare oder syntaktisch nicht gewollte Satzstrukturen von vornherein beim Erkennungsprozess der Sprache auszuschließen und damit die Leistung der Spracherkennung zu steigern. Dazu wird sie als Schnittstelle zwischen dem Spracherkenner und einem Interpretationsmodul eingesetzt.

Obwohl die ISR-Grammatik durch kontextfreie Regeln definiert ist, können bei Bedarf besondere Symbole zur Erweiterung der Grammatik eingesetzt werden. Die unterschiedlichen Symbole und ihre Bedeutungen werden im Abschnitt 2.1.2 beschrieben. Zum besseren Verständnis wird zuvor auf den Aufbau und die Struktur der Grammatik eingegangen.

2.1.1 Aufbau und Struktur

Der Aufbau einzelner Regeln der ISR-Grammatik ist durch ihre kontextfreie Eigenschaft größtenteils festgelegt(vgl. *Chomsky Hierarchie 2*)¹. Daher wird im Folgenden nur auf die symbolische Repräsentation der Grammatikelemente eingegangen.

`“$Greeting: hello | hi | hi $Name ;”`

zeigt exemplarisch den Aufbau einer Regel. Im Folgenden wird auf die Bedeutung der einzelnen Bausteine der Regel eingegangen. Das von dem Dollarsymbol und dem Doppelpunkt begrenzte *Non-Terminal Greeting* ergibt seine Deklaration

¹http://de.wikipedia.org/wiki/Chomsky-Hierarchie#Typ-2-Grammatik_.28kontextfreie_Grammatik.29

und stellt somit den Anfang einer Regel dar. Typischerweise dürfen zwei Deklarationen nicht denselben Namen tragen. Das nachfolgende Element `hello` wird als *Terminal* bezeichnet. Das Pipe-Symbol `|` repräsentiert das klassische kontextfreie Oder-Symbol. Die Benutzung eines Non-Terminals stellt das vorletzte Element der Beispielregel dar. Das Non-Terminal `Name`, gekennzeichnet durch das Dollar-Symbol, zeigt auf eine weitere Regel innerhalb der Grammatik. Der Anfang dieser Regel würde wie folgt aussehen: `$Name:`. Das Ende jeder Regel wird durch ein Semikolon gekennzeichnet.

Terminale und Non-Terminale werden aus dem regulären Ausdruck

```
[a-zA-Z\_"] [a-zA-Z\_-\\"0-9]*
```

beschrieben, wobei Non-Terminale mit dem Präfix `$` gekennzeichnet sind. Die Startregel wird durch `$$S:` gegeben.

Das gerade beschriebene Beispiel zeigt nur den Standard einer Regel. Die Besonderheiten und Zusatzsymbole der Grammatik, deren Aufbau und Wirkungsweise, wird im folgendem Abschnitt erläutert.

2.1.2 Besonderheiten und Zusatzsymbole

Joker-Symbol Das Joker-Symbol ist ein spezielles Terminal der Grammatik, welches durch die Zeichenfolge `!*` beschrieben wird. Ist der Joker Teil einer Regel, kann vom Compiler an genau dieser Stelle, jedes beliebige Terminal eingesetzt werden, um den Satz zu vervollständigen.

Technisches Non-Terminal Auch das technische Non-Terminal ist eine Besonderheit der ISR-Grammatik. Durch ein zweites `$` im Präfix wird aus einem gewöhnlichen, ein technisches Non-Terminal. Ein typischer Fall, ist das Start-Symbol `$$S`. Dies ist für die Interpretation des Satzes irrelevant, würde aber dennoch immer an oberster Stelle eines Syntaxbaums als Non-Terminal stehen. Als technisches Non-Terminal wird es beim Erstellen jedoch ignoriert, hilft aber beim Prozess des Parsens.

Ignore-List Die Ignore-List ist eine weitere Besonderheit der ISR-Grammatik. Diese enthält eine Liste von Terminalen, die beim Validieren eines Satzes vom Compiler ignoriert werden können. Typischerweise sind dies Füllwörter oder Unterbrechungen wie *ehh*, *ehm* oder *ah*. Ihr Aufbau ist mit dem Aufbau einer gewöhnlichen Regel vergleichbar. Durch `%Ignore=` wird dem Compiler mitgeteilt, dass es sich im Folgendem um eine Ignore-List handelt, diese wird ebenfalls durch ein Semikolon geschlossen. Im Gegensatz zu einer Regel, sind innerhalb der Ignore-List nur Terminale erlaubt. So kann durch eine entsprechende Wahl von Terminalen erreicht werden, dass die zwei abweichenden, aber semantisch gleichen Sätze „*I like red cups*“ und „*I like ehh red cups*“ vom Compiler äquivalent geparkt werden.

2.1.3 Beispiel

Im Folgenden ist ein Beispiel² einer ISR-Grammatik zu sehen. Anschließend werden Beispielsätze gezeigt, die mit dieser Grammatik erstellt werden können.

```
$TIME : $DAY $TIME_CIRCA $TIME_PRECI ;
$DAY : today ;
$TIME_CIRCA : in the morning | ;
$TIME_PRECI : at $TIME_HOUR oclock
              | at $TIME_HOUR point $TIME_MIN | ;

$TIME_HOUR : one | two | three ;
$TIME_MIN : zero | fifteen | thirty ;
$$S : $TIME ;
```

Aus dieser Grammatik lassen sich unter anderem folgende Sätze bilden „*Today*“, „*Today in the morning*“ oder „*Today at one oclock*“. Die Abbildung 2.1 stellt die Herleitung des dritten Satzes in einem Syntaxbaum dar.

²Quelle: Manualpage des ISR-Spracherkenner

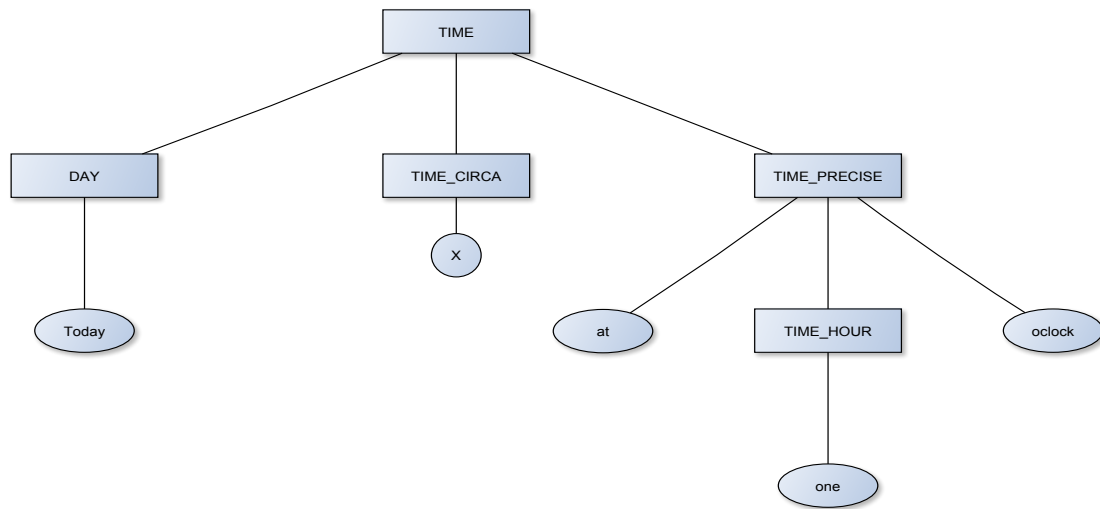


Abbildung 2.1: In dieser Abbildung ist ein Syntaxbaum dargestellt, indem die Herleitung des Satzes „**Today at one oclock**“ zu sehen ist.

2.2 Anforderungen an die Software

Eine Software, die den Umgang mit der ISR-Grammatik erleichtern soll, muss bestimmte Anforderungen erfüllen. Die Art, in der dem Nutzer Unterstützung geboten werden soll, ist eine grundlegende Entscheidung, die zunächst getroffen werden muss. Da die ISR-Grammatik durch eine Ansammlung textueller Regeln definiert ist, liegt es nahe, einen Texteditor zu entwerfen. Zu den Mindestanforderungen an einen Editor gehören unter anderem die Basisfunktionalitäten Kopieren, Ausschneiden, Einfügen und Suchen. Eine weitere sinnvolle Anforderung ist eine interne Dateiverwaltung.

Als grundlegende Voraussetzung gilt, dass der Editor übersichtlich und leicht zu benutzen ist, um dadurch eine möglichst kurze Einarbeitungsphase in den Editor zu gewährleisten.

Darüber hinaus, und das trifft auf die existierenden Editoren nicht zu, muss er eine Funktionsvielfalt bieten, die es lohnenswert macht, gerade diesen Editor zur Bearbeitung von ISR-Grammatiken zu benutzen. Um Erweiterungen oder Verbesserungen vornehmen zu können, sollte der zugrundeliegende Editor in seiner Architektur so offen sein, dass weitere Funktionalitäten leicht in das Programm integriert werden können.

Über allen Funktionalitäten und den damit angebotenen Hilfen steht das Ziel einer effizienteren Arbeitsweise im Vordergrund. Dies gilt sowohl für Anfänger als auch für erfahrene Benutzer.

2.3 Anwendungsfall

Im Rahmen des Robocup-Events ist Alice dazu eingeteilt worden, die erforderlichen ISR-Grammatiken für die Spracherkennung zu erstellen. Ihre erste Aufgabe ist es, alte Grammatiken nach ihrer Korrektheit zu prüfen und gegebenenfalls zu korrigieren. Da der benutzte Editor ihr keine Hilfestellung bietet, fehlerhafte Dateien direkt zu erkennen, muss sie jede zu prüfende Grammatik manuell nach Fehlern durchsuchen. Als Hilfestellung greift sie auf den Compiler der ISR-Grammatiken zurück. Nach einiger Zeit ist dieser mit dem Parsen einer Grammatik fertig. Alice durchsucht die Log-Ausgaben und findet schließlich eine Angabe, die auf einen Fehler hinweist. Alice öffnet die defekte Grammatik und sofort erscheint eine unstrukturierte Ansammlung von Regeln, die diese Grammatik spezifizieren. Als sie zu der Zeile navigieren möchte, wo der Fehler sich befinden soll, fällt ihr auf, dass die Zeilenangabe des Compilers nicht stimmen kann oder sich nicht auf diese Datei bezieht, da die angegebene Datei nicht so viele Zeilen hat. Daher ist sie gezwungen, die komplette Grammatik letztendlich doch manuell zu durchsuchen. Eine direkte Fehleranzeige innerhalb der Grammatik hätte ihr den Fehler direkt nach dem Öffnen der Datei angezeigt und ihr damit viel Zeit erspart.

Nachdem Alice alle Grammatiken syntaktisch geprüft hat, besteht ihre nächste Aufgabe darin, die Semantik zu validieren. Dies bedeutet für sie zu verifizieren, ob jede Grammatik alle benötigten Sätze abbilden kann. Ist dies nicht der Fall, muss sie die Regeln der Grammatik so erweitern, dass die fehlenden Sätze abgebildet werden können. Alice öffnet die erste Grammatik und beginnt mit dem ersten Satz, indem sie das Startsymbol der Grammatik sucht. Im weiteren Verlauf, muss Ali-

ce immer wieder durch die Grammatik navigieren, um die benötigten Non-Terminale zu finden. Des öfteren kommt sie dabei durcheinander, sodass sie nach Beendigung der Aufgabe nicht sicher sein kann, ob wirklich alle benötigten Sätze von der Grammatik gebildet werden können. Das langsame Navigieren durch die Grammatik kostet sie dabei zusätzlich viel Zeit.

Da Alice mit dieser Aufgabe beschäftigt ist, soll Bob eine weitere Aufgabe von Alice übernehmen. Eine kürzlich erstellte Aufgabe im Regelbuch des Robocup-Events erfordert eine komplett neue Grammatik, die von Bob erstellt werden soll. Er bekommt ein Pool von Sätzen, die von der Grammatik abgebildet werden sollen. Da Bob keine Erfahrungen mit der ISR-Grammatik hat, fällt es ihm schwer einen Überblick über die syntaktische Struktur zu erlangen. Nach anfänglichen Problemen schafft er es, die Grammatik zu erstellen. Jedoch stellt sich ihm nun das gleiche Problem wie zuvor Alice. Er kann sich nicht sicher sein, ob die Grammatik sowohl semantisch als auch syntaktisch einwandfrei funktioniert.

Nachdem Alice ihre Aufgaben bearbeitet hat, möchte sie einen genauen Überblick über die von Bob entworfene Grammatik erhalten. Da Alice eher visuell ambitioniert ist, fällt es ihr schwer einen Überblick über die ihr unbekannte Grammatik zu bekommen.

2.4 Anforderungsanalyse

Um angemessene Anforderungen stellen zu können, denen das Programm gerecht werden soll, muss zunächst analysiert werden, welche Zielgruppen und unter welchen Voraussetzungen das Programm benutzt wird. Einen Anhaltspunkt liefert das in der Motivation in Kapitel 1.1 beschriebene Robocup@Home-Event. Dazu wird an der Universität Bielefeld ein Robocup-Projekt angeboten, dessen Team jedes Jahr überwiegend neu zusammengesetzt wird. Daher kann angenommen werden, dass diese Studenten keine oder nur wenig Erfahrung mit der ISR-Grammatik haben. Das ESMERALDA-Spracherkennungssystem, welches diese Grammatik be-

nutzt, wird aber auch in verschiedenen anderen Projekten eingesetzt, die überwiegend von festen Mitarbeitern der Universität geleitet werden. Diese Anwender werden, im Gegensatz zu vielen Studierenden, die mit der Software arbeiten, schon einige Erfahrungen mit der ISR-Grammatik gemacht haben und daher fundierte Kenntnisse besitzen. Aufgrund dieser Annahmen können nun angemessene Anforderungen formuliert werden.

Einen besonderen Stellenwert hat die leichte Erlernbarkeit, da sich jedes Semester neue, unerfahrene Studierende mit dem Spracherkenner befassen werden.

Anforderung 1: Die Software bedarf nur einer kurzen Einarbeitungsphase.

Folglich sollte auch die Handhabung nicht zu kompliziert sein das heißt, eine intuitive Bedienung ist zwingend notwendig.

Anforderung 2: Die Software ist intuitiv bedienbar.

Aus der Analyse ergibt sich auch, dass das Programm eine bessere Übersicht über eine gegebene Grammatik schaffen sollte und zusätzlich das Navigieren durch große Grammatiken präzisiert.

Anforderung 3: Die Software gibt einen guten Überblick über die bestehenden Grammatiken.

Um auch unerfahrenen Nutzern ein effizientes Arbeiten zu ermöglichen, sollten komplizierte oder immer wiederkehrende Aufgaben durch die Funktionalitäten erleichtert oder sogar ganz übernommen werden können.

Anforderung 4: Die Software löst oder unterstützt die Aufgabenbearbeitung.

Eine weitere Anforderung an das Programm ist es, dem Anwender aktiv Lösungsmöglichkeiten anzubieten, deren Verwendung bei anderen Editoren fundierte Kenntnisse voraussetzten würde.

Anforderung 5: Die Software bietet einen Überblick über die gegebenen Lösungsmöglichkeiten.

Darüber hinaus sollte durch das Programm eine schnellere und effizientere Arbeitsweise ermöglicht werden.

Anforderung 6: Die Software beschleunigt das Erstellen einer Grammatik und reduziert die Fehleranfälligkeit.

Resümierend lässt sich konstatieren, dass all diese Funktionen und Anforderungen sowohl Anfängern als auch erfahrenen Nutzern helfen sollen, die Grammatik besser zu verstehen. Der Umgang mit der Grammatik soll erleichtert und effizienter werden.

2.5 Verwandte Arbeiten

Dieses Kapitel beschäftigt sich mit Software, die ähnliche Aufgabenbereiche behandeln. Durch den detaillierten Anwendungsfall aus Kapitel 2.3 gingen Informationen hervor, die bei der Modellierung der Software beachtet werden müssen. Diese Informationen können nun benutzt werden, um herauszufinden ob es möglicherweise schon bestehende Software gibt, die verwendet oder deren Funktionalitäten man erweitern kann. Dabei wurde bei der Auswahl der vorzustellenden Software darauf geachtet, dass sie möglichst den aktuellen Stand der Forschung, auf diesem Gebiet wiedergibt. Der Fokus dieses Kapitels ist auf drei verschiedene Produktarten gelegt, welche sich grundsätzlich in der Art wie sie den Anwender Unterstützen unterscheiden.

Xtext Es gibt Software auf dem Markt, welche sich mit dem Erstellen von eigenen Sprachen beschäftigt. Ein sehr bekanntes Framework ist ***Xtext***³, welches für die Programmierumgebung Eclipse entwickelt wurde.

Xtext bietet in der Benutzung viele Vorteile. Besonders hervorzuheben sind die vielen Tutorials und eine sehr gute Dokumentation, welche den Einstieg erleichtern.

³<http://www.eclipse.org/Xtext/>

Zudem werden durch die Benutzung von Xtext viele Funktionen implizit bereitgestellt. So wird automatisch ein Syntax-Parser über die *ANTLR*⁴-Technologie, Syntaxhighlighting, und einen Content Assistant zur erstellten Sprache mitgeliefert.

Dies kann durchaus in vielen Projekten ein Vorteil sein, bietet für einen ISR-Grammatik-Editor aber nur bedingt Hilfe. Dadurch, dass es sich um eine kontextfreie Grammatik handelt, entstehen unter anderem dynamische *Keywords*⁵, welche wiederum bei dem Syntaxhighlighting berücksichtigt werden müssen. Dies ist aber zum jetzigen Zeitpunkt in Xtext nur unkomfortabel lösbar. Auch der mitgelieferte Parser, müsste überschrieben werden, um eigene Fehlermeldungen an den Nutzer übermitteln zu können.

Der Hauptgrund, warum Xtext den hier gestellten Anforderungen nicht genügt, liegt in der Fokussierung auf der Erstellung von Sprachen oder Grammatiken. Die Einbindung von zusätzlichen, Editor unabhängigen, Funktionalitäten wie zum Beispiel eine Hilfestellung zum Lösen des Wortproblems, eine Visualisierung oder erweiterte Exportfunktionen können nicht integriert werden.

Parser-Editor Neben Frameworks und anderen Hilfsmitteln zur Erstellung eines Editors gibt es eine Reihe von Editoren, deren Nachteil oft darin besteht, dass sie nicht ausreichend erweiterbar sind. Ein Beispiel hierfür bildet der ***Parser-Editor***⁶, welcher an der Universität Hannover entwickelt wurde. Dieser bietet eine Programmierumgebung für kontextfreie Grammatiken. Erwähnenswert macht diesen Editor die gute Darstellung der Grammatik. Sie kann mit Hilfe von Syntaxhighlighting und weiteren Hilfsinformationen einfacher verstanden und bearbeitet werden. Der Schwerpunkt dieses Editors liegt jedoch nicht auf der Erstellung einer Grammatik, sondern wie schon aus dem Namen ersichtlich, auf der Erstellung von Parsern und Kompilern für eine kontextfreie Grammatik. Da der Parser-Editor eine eigene, von der ISR-Grammatik unterschiedliche, Notation verwendet, kann er zur Erstellung von ISR-Grammatiken jedoch nicht genutzt werden.

⁴<http://www.antlr.org/>

⁵Als Keywords werden Wörter oder Zeichenmuster genannt, die in ihrer Sprache eine besondere Bedeutung haben

⁶<http://www.psue.uni-hannover.de/ParserSuite/plugin.php>

Onlinesoftware Eine Produktsuche im Internet für Online-Editoren zur Gestaltung kontextfreier Grammatiken bringt wenig brauchbare Ergebnisse. Die meisten Editoren ähneln dem im Link⁷ zu findenden Beispiel. Der Vorteil einer Onlinesoftware besteht jedoch darin, dass sie über jeden internetfähigen Rechner direkt zugänglich ist und keine Installation vorgenommen werden muss. Der große Nachteil besteht aber in ihrer Abgeschlossenheit. So sind die meisten Tools nur auf die vorgegebene Syntax angepasst und bieten keine Möglichkeit, das Programm zu erweitern.

2.6 Benutze Technologien

Zunächst wurde festgelegt, dass keine StandAlone-Software entwickelt werden sollte, stattdessen wurde ein Plugin für eine schon bestehende Software präferiert. Die Entwicklungsumgebung *Eclipse*⁸ bietet mit seiner offenen Architektur eine sehr gute Grundlage für eine Erweiterung über Plugins [2]. Daraus resultierend wurde für die Implementierung die Programmiersprache *Java*⁹ gewählt. Dies hat zusätzlich zwei wesentliche Vorteile: zum einen bietet Eclipse eine sehr gute Dokumentation und viele Beispielprojekte, zum anderen werden durch die Wahl ein Plugin zu implementieren, bereits implizit viele Anforderungen an das Programm erfüllt. Ein Editor-Plugin für Eclipse impliziert alle in der Anforderung aufgezählten Grundbausteine. Zwar müssen hinsichtlich der Übersichtlichkeit einige Abstriche gemacht werden, die es bei einer StandAlone-Software nicht gegeben hätte, jedoch werden diese größtenteils dadurch kompensiert, dass Eclipse für viele Programmierer bereits eine gewohnte Umgebung darstellt (siehe Anforderung eins und zwei der Analyse in Kapitel 2.4). Auch bietet Eclipse eine effiziente Dateiverwaltung und viele Schnittstellen, um eigene Funktionalitäten in das Plugin zu integrieren sodass die dritte Anforderung aus der Analyse (siehe Kapitel 2.4) ebenfalls erfüllt ist.

Im weiteren Verlauf werden Technologien vorgestellt, die benutzt worden sind, um die Implementierung von Funktionalitäten des Plugins zu unterstützen.

⁷<http://smlweb.cpsc.ucalgary.ca/>

⁸<http://www.eclipse.org/>

⁹<http://www.oracle.com/technetwork/java/index.html>

Zest Zest ist ein Tool zur Visualisierung von Strukturen und Graphen. Die im letzten Abschnitt der Anwendergeschichte (siehe Kapitel 2.4) vorgestellte Visualisierung kann mit Hilfe des **Zest-Frameworks** realisiert werden. Die Entscheidung, weshalb auf Zest zurückgegriffen wurde, hat mehrere Gründe. Dazu zählt zum Beispiel, dass Eclipse eine einfache Lösung bietet Zest einzubinden. Darüber hinaus ist es dadurch, dass die Visualisierung über *SWT*¹⁰ und *Draw2d*¹¹ geregelt wird, für das Eclipse-Plugin in der Hinsicht hilfreich, dass es eine nahtlose Einbettung, in das System von Eclipse ermöglicht.¹²

SOCS Grammar Editor Das Projekt Grammar Editor ist eine Bibliothek von Java-Klassen, die einen Parser und eine Visualisierung von Syntaxbäumen bereitstellt. Um verschiedene Sätze gegenüber einer ISR-Grammatik zu validieren, wurde auf die Funktion einiger Klassen der Bibliothek zurückgegriffen.¹³

¹⁰<http://www.eclipse.org/swt/>

¹¹<http://www.draw2d.org/draw2d/>

¹²<http://www.eclipse.org/gef/zest/>

¹³<http://ozark.hendrix.edu/~burch/socs/software.html>

3 Umsetzung

3.1 Funktionalitäten

Die Ergebnisse aus der Anforderungsanalyse (Kapitel 2.4) haben deutlich gezeigt, dass es viele nützliche Hilfen gibt, die ein Editor zum Erstellen einer ISR-Grammatik bereitstellen kann. Zusätzlich wurde die Auswahl wichtiger Funktionalitäten durch das *IBM-Tutorial “Create a commercial-quality Eclipse IDE”*¹ bekräftigt. Im Folgenden werden Funktionalitäten vorgestellt, die alle Anforderungen abdecken. Um den Einstieg in die Materie zu erleichtern und den Zusammenhang mit der Anforderungsanalyse zu zeigen, beginnt dieses Kapitel mit einer allgemeinen Erklärung jeder implementierten Funktionalität. Zudem wird gezeigt, durch welche Eigenschaft die Funktionalität eine bestimmte Anforderung erfüllt.

3.1.1 Syntaxhighlighting

Syntaxhighlighting ist ein elementarer Bestandteil dieses Editors. Durch farbliche Hervorhebung werden Keywords markiert. Im Gegensatz zu vielen Programmiersprachen gibt es in der ISR-Grammatik nur wenige statische Keywords, dafür unbegrenzt viele dynamische. Statische Keywords sind vorab festgelegt, zu ihnen gehören `%IGNORE`, `$$$`, `!*`, `:`, `;` und `|`. Dynamische Keywords entstehen erst beim Erstellen der Grammatik. Jede Deklaration und jedes Non-Terminal sind dynamisch erzeugte Keywords. Eine farbige Repräsentation der Grammatik hilft dem Nutzer, sich einfacher in ihr zurecht zu finden. Sie erhöht die Lesbarkeit und hilft ihm, wichtige von unwichtigen Bestandteilen der Grammatik zu unterscheiden und reduziert somit die Fehleranfälligkeit (Anforderung sechs, Kapitel 2.4).

¹<http://www.ibm.com/developerworks/opensource/tutorials/os-ecl-commplgin1/os-ecl-commplgin1-pdf.pdf>

3.1.2 Automatische Formatierung

Um der Anforderung drei aus der Analyse (Kapitel 2.4) zu entsprechen wird ein weiteres wichtiges Werkzeug dem Editoren hinzugefügt, die automatische Formatierung. Diese bietet dem Nutzer die Möglichkeit, die geschriebenen Regeln automatisch in ein vorgegebenes Standardformat zu bringen. Um dies zu definieren, ist es notwendig Strategien, zu entwickeln nach dem die automatisierte Formatierung vorgehen soll. Eine solche Strategie kann zum Beispiel beinhalten, dass nach bestimmten Symbolen Umbrüche eingefügt werden.

Ein eigenes Format ohne editorseitige Unterstützung einzuhalten, würde vermehrt Zeit in Anspruch nehmen. Ein automatischer Formatierer erhöht auf einfachem Weg die Lesbarkeit der Grammatik. Zusätzlich gewährleistet er, dass alle Nutzer dasselbe Format benutzen und genau den definierten Standard einhalten. Wird die Autoformatierung regelmäßig eingesetzt, erhalten weitere Nutzer einen schnellen Überblick über ihnen unbekannte Grammatiken.

Abbildung 3.1 zeigt Regeln einer Grammatik dessen Formatierung dem entwickelten Standard entsprechen. Des Weiteren wird die Funktionsweise des Syntaxhighlighting verdeutlicht und gibt Aufschluss über die Farbwahl der Regelemente.

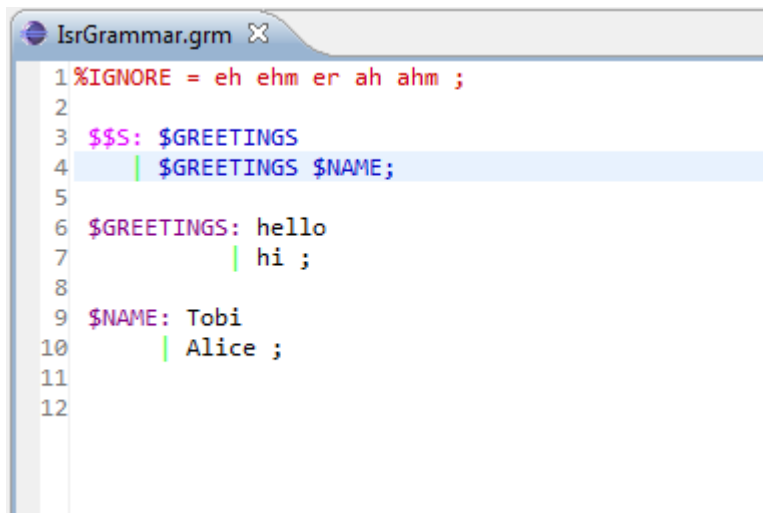


Abbildung 3.1: Syntaxhighlighting einer standardformatierten Grammatik im ISR-Grammatik-Editor.

3.1.3 Fehlermarkierung

Die Prüfung der Syntax von Sprachen und vor allem die Visualisierung der auftretenden Fehler, ist ein wichtiges und mächtiges Werkzeug in einer Entwicklungsumgebung. Ob es sich bei den Sprachen um eine Programmiersprache wie C, C++ oder Java handelt oder ob sie eine kontextfreie Grammatik wiedergeben, macht in dieser Hinsicht keinen Unterschied. Ein solches Werkzeug sollte daher auch in diesem Editor nicht fehlen.

Da für die ISR-Grammatik ein Compiler schon existiert, stellt sich die Frage, welchen Vorteil ein integrierter Parser mit sich bringt. Zwei Vorteile gehen aus Beobachtungen hervor. Die Ergebnisse des integrierten Parsers sind sehr viel leichter zu interpretieren, da er Fehler in den Regeln direkt anzeigt und gegebenenfalls Verbesserungsvorschläge anzeigt. Der Benutzer erkennt also, die Ursache des Fehlers und kann ihn direkt beheben. Des Weiteren hat der interne Parser gegenüber dem externen eine extrem kurze Kompilierzeit. Der Vorteil einer direkten Fehlermarkierung ist klar ersichtlich. Da durch sie schon während der Grammatikerstellung Syntaxfehler direkt vermieden werden, bringen sie eine enorme Zeitersparnis im Prozess der Syntaxvalidierung. Damit entspricht sie den Anforderungen drei bis sechs der Analyse in Kapitel 2.4.

3.1.4 Hoverinformation

Als Hoverinformation werden temporär angezeigte Fenster bezeichnet, die situationsspezifische Informationen beinhalten. Durch Positionierung des Mauszeigers über einen längeren Zeitraum auf ein in der Regel vorkommendes Element, erscheint an selber Stelle eine kontextsensitive Hoverinformation. Bei Veränderung der Position wird sie automatisch ausgeblendet.

Ist das Wort von einer Fehlermarkierung umgeben, wird die Fehlerbeschreibung und gegebenenfalls ein Lösungsvorschlag angezeigt (siehe Anforderung fünf, Kapitel 2.4). Ist dies nicht der Fall, wird geprüft, ob es sich um ein Non-Terminal handelt. In diesem Fall besteht die Information aus der Deklaration des Non-Terminals. Trifft keines von beidem zu, wird kein Fenster geöffnet.

Hoverinformationen ermöglichen dem Anwender, auf eine einfache und schnelle Art und Weise, einen Überblick über die kontextsensitive Information eines Regel-

elementes zu erhalten.

Abbildung 3.2 zeigt die Fehlermarkierung des ISR-Grammatik-Editors. Um an den Fehlertext zu gelangen, wurde die Funktionalität der Hoverinformation benutzt. Diese ist in dem gelben Rechteck unter Zeile Acht zu sehen. Rote Linien deuten auf einen fatalen Fehler hin, wohin gegen gelbe Linien Warnungen darstellen. Das Non-Terminal **GERMAN** hat keine zugehörige Deklaration, das Non-Terminal **NAME** hingegen ist unbenutzt.

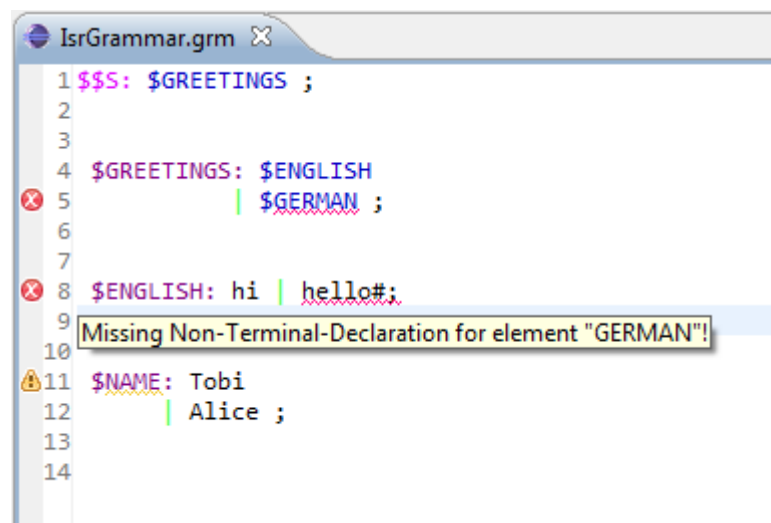


Abbildung 3.2: Zwei verschiedene Arten von Fehlermarkierungen. Die gelben Markierungen stellen Warnungen dar, rote zeigen fatale Fehler an. Der Fehlertext wird als Hoverinformation dargestellt.

3.1.5 Content Assistant

Die Funktion eines Content Assistant besteht darin, dem Benutzer eine kontext-sensitive Auswahl an Programmiermöglichkeiten anzubieten. Der ISR-Grammatik-Editor kann unter anderem Deklarationen, Non-Terminals oder Sondersymbole bereitstellen. Oft wird der Content Assistant auch zur automatisierten Vervollständigung benutzt. Die Möglichkeiten, den Assistenten intelligent zu gestalten, sind praktisch unbegrenzt. Zum Beispiel zeigt ein optimierter Assistent dem Programmierer fehlende Deklarationen oder ordnet die Vorschläge nach ihrer Bedeutsamkeit. Darüber hinaus kann er vorgefertigte Konstrukte oder Hinweise auf fehlende

Konstrukte anbieten.

Der Content Assistant ist vor allem dann eine große Hilfe, wenn der Anwender sich nur wenig mit der gegebenen Syntax auskennt. Doch auch für erfahrene Benutzer ist er hilfreich, denn er beschleunigt das Programmieren in vielen Situationen und reduziert durch das Einsetzen vorgefertigter Konstrukte die Fehleranfälligkeit (siehe Anforderung sechs Kapitel 2.4).

Die Abbildung 3.3 zeigt den Content Assistant bei Gebrauch. Zu sehen ist, dass das Non-Terminal **GERMAN** wie in Abbildung 3.2 keine zugehörige Deklaration besitzt. Der Content Assistant wurde zwischen zwei Regeln aufgerufen und zeigt als erstes Hilfskonstrukt die fehlende Deklaration.

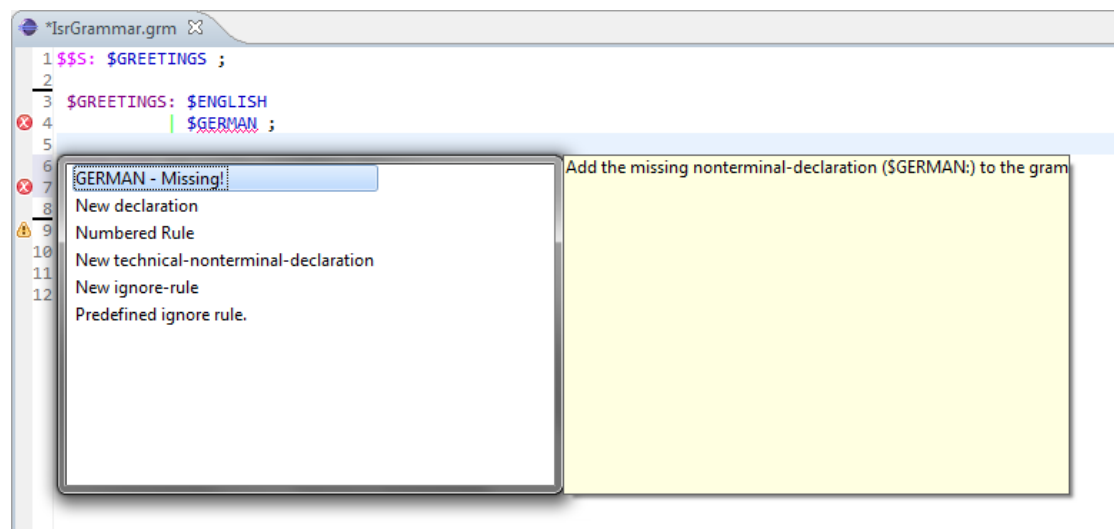


Abbildung 3.3: Content Assistant bei einem Aufruf zwischen zwei Regeln. Die fehlende Deklaration des Non-Terminals **GERMAN** als erste Auswahlmöglichkeit.

3.1.6 Hyperlinks

Hyperlinks werden in Eclipse beim Betätigen der strg-Taste dynamisch erstellt und per Mausklick aufgerufen. Im klassischen Java-Editor werden sie benutzt, um zum Beispiel von einem Methodenaufruf zur Implementierung dieser Methode zu gelangen, ohne dabei die Grammatik durchsuchen zu müssen. Da es auf den ersten Blick innerhalb der ISR-Grammatik keine Methoden gibt, kommt die Frage auf,

ob eine Portierung in einen Editor, der zur Erstellung einer ISR-Grammatik dient, sinnvoll ist.

Beim genauen Betrachten fällt jedoch auf, dass Parallelen zu einem Methodenaufruf existieren. Nimmt man an, dass eine Deklaration eine Methode und das dazu gehörende Non-Terminal, innerhalb einer Regel, ein Methodenaufruf sei, so folgt daraus, dass es auch bei diesem Editor sinnvoll ist, die Funktionalität der Hyperlinks zu integrieren.

Um den Anforderungen vier und sechs der Anforderungsanalyse aus Kapitel 2.4 näher zu kommen, bieten Hyperlinks gute Unterstützung. Denn durch sie wird es möglich, schnell und präzise zwischen Regeln zu navigieren. Dem Anwender wird dadurch das Finden der Referenz von Non-Terminalen durch den Editor abgenommen. Muss der Benutzer einer Deklaration etwas hinzufügen, reicht ein einfacher strg-Klick auf das referenzierende Non-Terminal. Dies spart besonders dann Zeit, wenn die Grammatik unbekannt ist oder sehr komplex erscheint.

3.1.7 Visualisieren von Regeln

Oft stellt es für Anwender, vor allem für Anfänger, ein Problem dar, komplexe Grammatiken zu verstehen. Selbst für einen Überblick reicht es gegebenenfalls nicht aus, wenn die Grammatik nur als Text vorliegt. Eine gute Unterstützung stellt eine farbige Visualisierung der Grammatik dar. Im Vordergrund steht dabei die Übersichtlichkeit. Eine geeignete Art der Visualisierung bietet das *Railroad*²-Diagramm, welches durch seine übersichtliche Darstellung dem Anwender eine adäquate Alternative zur textuellen Grammatik bietet (siehe Anforderung drei, Kapitel 2.4).

²<http://de.wikipedia.org/wiki/Syntaxdiagramm>

Die Abbildung 3.4 zeigt die Umsetzung der Visualisierung im ISR-Grammatik-Editor. Durch Einhalten der farbigen Darstellung ist leicht zu erkennen, um welche Regelemente es sich im Einzelnen handelt. Visualisiert wird nur der selektierte Text.

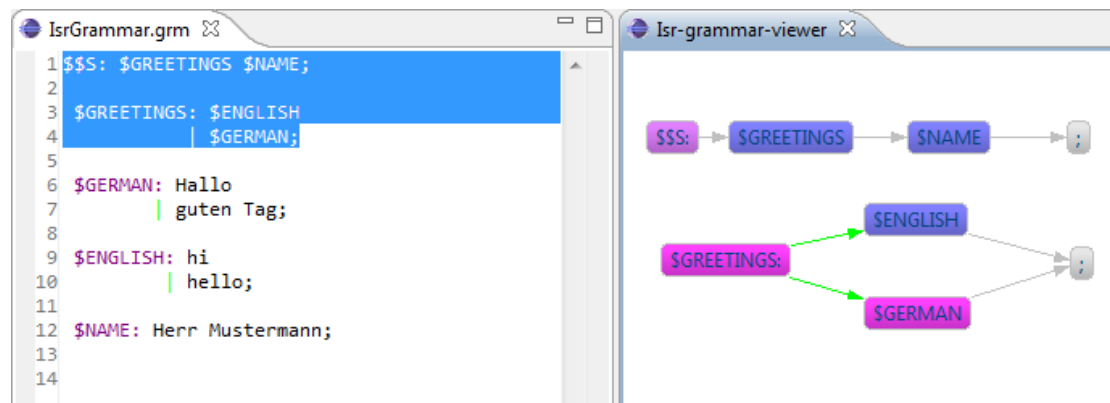


Abbildung 3.4: Visualisierung zweier Regeln einer Grammatik. Links im Bild ist die Selektion der Regeln, rechts die Darstellung.

3.1.8 Wortproblem und Syntaxbäume

Da der Editor speziell für die ISR-Grammatik entworfen wurde und diese einen Teil einer natürlichen Sprache repräsentiert, ist es interessant, auf einfachem und schnellem Weg, das Wortproblem zu lösen.

Doch zu wissen, dass ein Satz in einer Grammatik enthalten ist, reicht in manchen Situationen nicht aus. Sollen Sätze von einem System semantisch verstanden werden, verlangt dies nach einer genauen Interpretation der Satzstruktur. Der Aufbau eines Satzes und die dahinter liegende Struktur, ist nicht leicht aus der Grammatik abzulesen. Daher ist es nützlich, dass der Anwender bei solchen Aufgaben durch die Visualisierung der Syntaxbäume unterstützt wird (Anforderung vier, Kapitel 2.4). Aus dem erstellten Syntaxbaum erkennt der Anwender konkret, welches Terminal aus welchem Non-Terminal hervor gehen muss, damit der zu prüfende Satz gegenüber der Grammatik abgebildet werden kann.

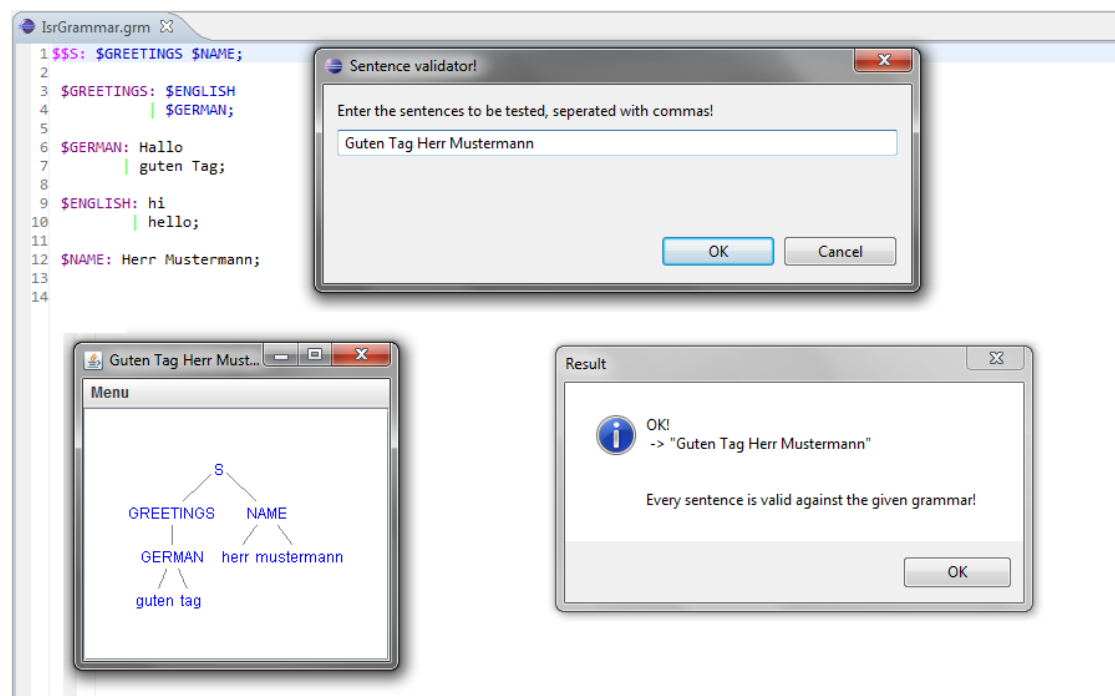


Abbildung 3.5: Die Abbildung zeigt das automatische Lösen des Wortproblems. Oben die Eingabe, unten das Ergebnis, nach drücken des OK-Button

Die Abbildung 3.5 zeigt die Benutzung und das Ergebnis dieser Funktionalität. Rechts oben bietet das System ein Dialogfenster zur Eingabe der Sätze. Im unteren Abschnitt der Abbildung ist das Ergebnis zu sehen. Rechts befindet sich eine Liste der geprüften Sätze, links eine optionale Visualisierung der Lösung des Wortproblems als Syntaxbaum.

3.2 Pluginarchitektur

Betrachtet man die Eclipse-Architektur, so stellt man fest, dass die komplette Funktionalität auf Plugins beruht.[3] Um eine Kommunikation zwischen den verschiedenen Plugins zu ermöglichen, kann der Designer eines Plugin Extension Points definieren. Diese bieten wiederum anderen Plugins die Möglichkeit, auf die bestehenden Funktionalitäten zuzugreifen.

Um Funktionalitäten, die bereits in einem anderen Plugin implementiert wurden,

dem eigenem zur Verfügung zu stellen, muss der entsprechende Extension Point in einer impliziten Extension deklariert werden. Eine Extension gibt also den Gebrauch von Extension Points an und muss in der für das Plugin notwendigen Datei, `Plugin.xml` definiert werden. Der bereitgestellte Extension Point muss zusätzlich in das Manifest eingetragen werden.[4] Dieses dient dem Plugin um Referenzen, Abhängigkeiten und erweiterte Information, wie zum Beispiel die Versionsnummer oder den Namen des Plugins zu definieren.

Um nun ein eigenes Plugin zu aktivieren, bedarf es lediglich einer einzigen von Eclipse vorgeschriebenen Java-Klasse, dem `Activator`. Der `Activator` implementiert das Interface `org.osgi.framework.BundleActivator`. Durch dieses kann Eclipse die Methode `start` oder `stop` aufrufen, um das Plugin bei Bedarf zu starten oder zu stoppen.

Zur einfacheren Implementierung neuer Plugins bietet Eclipse viele nützliche Ober-Klassen an, sodass viele Methoden nicht mehr selbst implementiert werden müssen. Um diesen Vorteil zu nutzen, und da es sich bei dem vorgestellten Plugin um einen Editor handelt, ist es sinnvoll den `Activator` von `org.eclipse.ui.AbstractUiPlugin` abzuleiten.

Abbildung 3.6 verdeutlicht die Vorgehensweise der Plugin-Architektur.

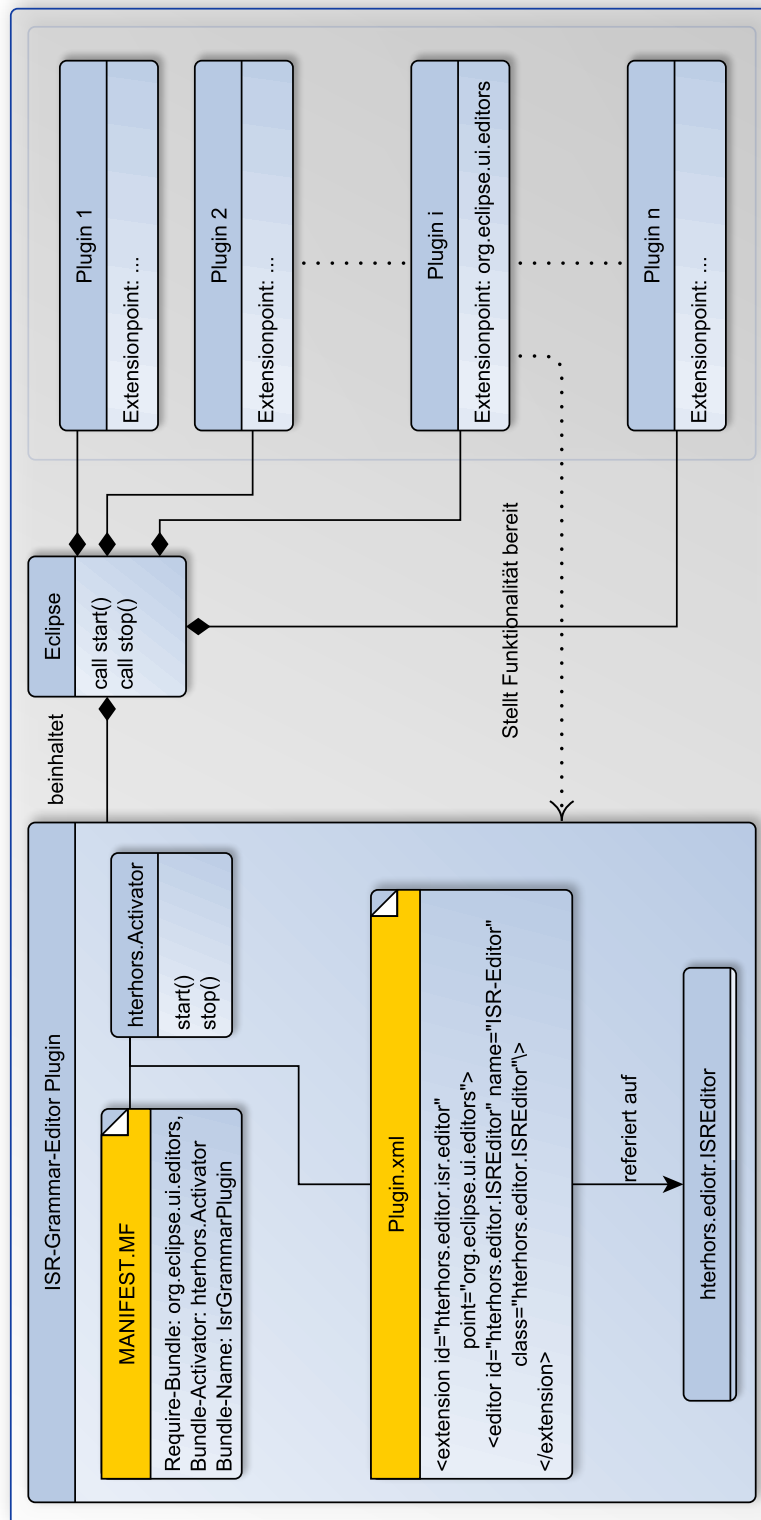


Abbildung 3.6: Die Abbildung zeigt die Plugin-Architektur. Links das neu erstellte Plugin, rechts die bereits vorhandene Plugins.

3.3 Der ISR-Grammatik-Editor

In der `Plugin.xml` wird nun die erste Extension eingetragen, um Eclipse mitzuteilen, dass es sich bei dem Plugin um einen Editor handelt. Dazu integriert man den Extension Point `org.eclipse.ui.editors` und macht einen entsprechenden Eintrag in das Manifest.

Typischerweise wird jeder Editor mit einer bestimmten Dateiendung verknüpft, sodass beim Öffnen einer derartigen Datei automatisch der richtige Editor verwendet wird. In diesem Fall ist `.grm`, die vorgeschriebene Endung einer ISR-Grammatik. Auch diese Information wird in der Extension eingetragen.

Um den vollen Funktionsumfang des ISR-Grammatik-Editors zu erreichen, werden drei Java-Klassen angelegt: der `ISREditor` selbst, ein Dokumenten-Provider, der `ISRDocumentProvider` und eine Konfiguration, die `ISRSourceViewerConfiguration`.

Im Folgenden wird kurz auf den Verwendungszweck dieser Klassen eingegangen. Abbildung 3.7 veranschaulicht die Editor-Architektur.

Die Klasse `ISREditor` Der `ISREditor` bildet das Grundgerüst des ISR-Grammatik-Editors. Durch die Ableitung von der Oberklasse `org.eclipse.ui.editors.text.TextEditor` stehen diesem die Grundfunktionen eines Texteditors und die Möglichkeit diese zu erweitern, zur Verfügung. Dazu werden die `ISRSourceViewerConfiguration` und `ISRDocumentProvider` über spezifizierte `setter`-Methoden mit dem `ISREditor` verknüpft.

Die Klasse `ISRDocumentProvider` Der `ISRDocumentProvider` hat die Aufgabe, dem Editor den Text zur Verfügung zu stellen. Dazu wird der Inhalt durch `createDocument` aus einer gegebenen Datei gelesen. Als nächstes wird der Inhalt partitioniert, dafür wird er in Abschnitte eingeteilt, die jeweils mit einem Label, dem *content type*³, versehen werden. Die Partitionierung erfolgt nach spezifizierten Regeln, die im `IsrPartitionScanner` definiert werden. Betrachtet man die

³Als content type werden die Namen der Partitionierungsabschnitte bezeichnet.

Struktur der ISR-Grammatik, fällt auf, dass sie nur zwei unterschiedliche Bereiche aufzeigt. Für eine angemessene Partitionierung wurden die zwei content types `ISR_IGNORE` und `ISR_RULE` definiert. Da jedoch der Fall auftreten kann, dass die gegebenen Regeln nicht den kompletten Text abbilden können, wird automatisch ein zusätzliches Default-Label vom Provider hinzugefügt:

`__dftl_partition_content_type`.

Konkret bedeutet dies, dass jede Ignore-Regel das Label `ISR_IGNORE`, jede Deklaration `ISR_RULE` und alle nicht abgedeckten Bereiche, zum Beispiel die Zwischenräume der Regeln, `__dftl_partition_content_type` als Label erhalten.

Die Klasse `ISRSourceViewerConfiguration` Die `ISRSourceViewerConfiguration` liefert ihren Beitrag durch die Definition von Funktionen und kümmert sich um die korrekte Anzeige des im `ISRDocumentProvider` bereitgestellten Textes. Durch das Überschreiben vorhandener Methoden der Oberklasse `FileDocumentProvider` wird das Plugin um weitere Funktionalitäten erweitert und ermöglicht so eine gezielte Darstellung.

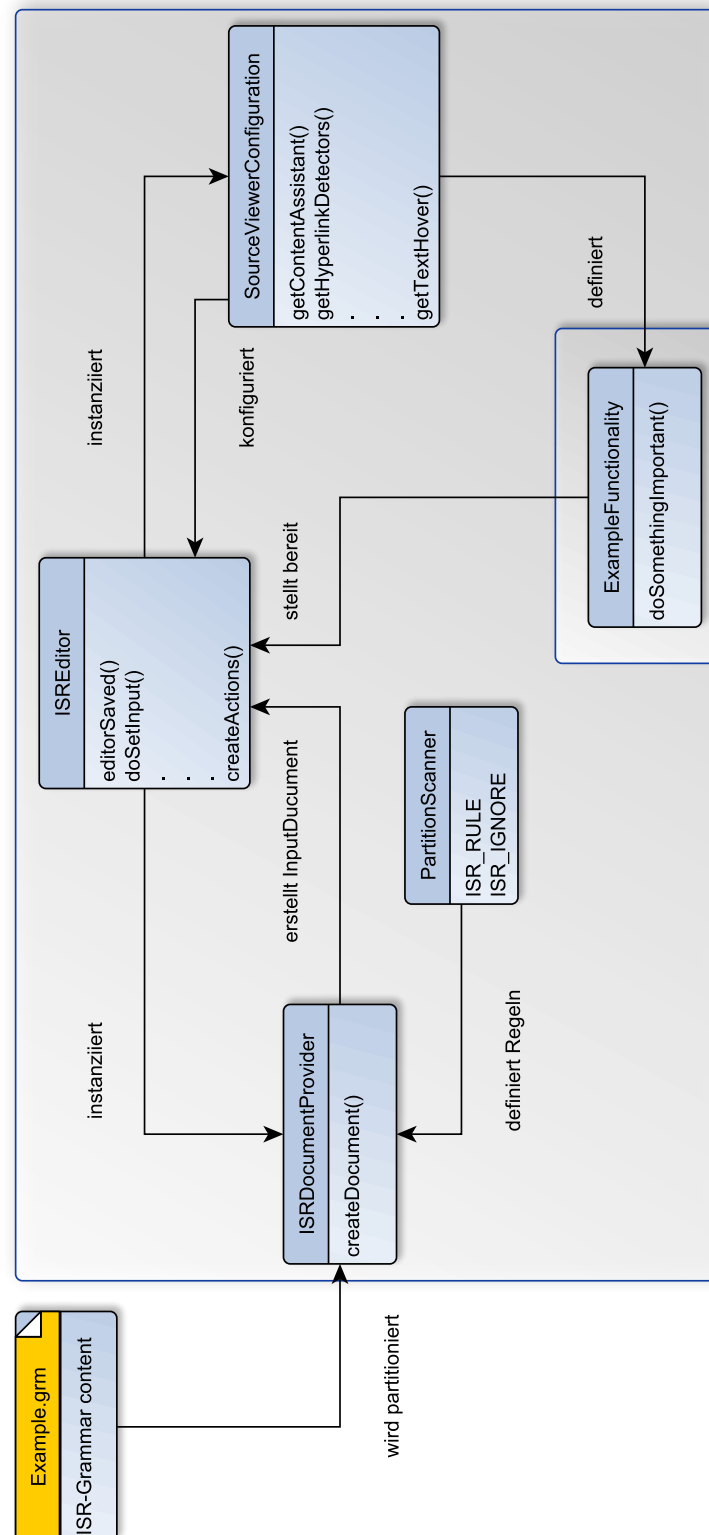


Abbildung 3.7: Die Editor-Architektur

3.4 Implementierung und Abläufe

Das folgende Kapitel geht auf den Ablauf und die Implementierung der zuvor genannten Funktionalitäten aus Kapitel 3.1 ein. Bevor eine Funktionalität in das Plugin implementiert werden kann, muss zunächst festgestellt werden, auf welche Weise die Implementierung geschehen kann. Dazu wurden im Wesentlichen drei Ansätze benutzt.

Funktionalität über Extension Points Die Implementierung von Funktionalität über Extension Points bietet eine Anbindung externer Funktionalitäten in das eigene Plugin, diese Funktionalitäten sind editorunabhängig wie zum Beispiel erweiterte Menüeinträge.

Funktionalität über Aktionen Funktionalitäten welche über Aktionen implementiert werden, sind dagegen editorabhängig und können nur dann benutzt werden, wenn der im Plugin enthaltenen ISR-Grammatik-Editor aktiv, also von Eclipse durch die Methode `start` aufgerufen, ist. Diese Variante wird benutzt um globale Funktionalitäten von Eclipse durch lokal definierte zu überschreiben, dazu müssen sie vorab vom Plugin in das Eclipse-System installiert werden. Ein Beispiel hierfür ist die benutzerdefinierte automatische Formatierung.

Funktionalität über direkte Implementierung Eine weitere Anbindung weiterer Funktionalitäten bietet die direkte Implementierung. Diese benötigt keine Vorinstallation und ist dennoch editorspezifisch. Hierunter fallen meist passive Eigenschaften des Editors wie zum Beispiel das Syntaxhighlighting.

Im Folgenden wird zuerst genauer auf jeden Ansätze eingegangen und die Implementierungen der Funktionalitäten beschrieben, welche sich auf diesen Ansatz beziehen.

3.4.1 Funktionalität über Extension Points

Eine Erweiterung der `Plugin.xml` stellt einen geeigneten Weg dar, den Editor mit zusätzlicher Funktionalität zu ergänzen. Eclipse bietet selbstständig einen breiten

Pool von Extension Points, auf die zugegriffen werden kann. Die grundlegende Idee ist, dass durch die Definition neuer Extensions in der Plugin.xml, dem Editor Funktionalität aus anderen Plugins bereitzustellen.

Dazu wird innerhalb der Plugin.xml ein *Command* definiert, welcher den Extension Point `org.eclipse.ui.commands` anspricht. Innerhalb dieses Commands wird eine eindeutige ID definiert. Über diese ID können nun weitere Extensions mit dem Command verknüpft werden. Als pluginseitige Schnittstelle wird als nächstes ein weiterer Extension Point angesprochen, der `org.eclipse.ui.handlers`. Innerhalb diesem ist es möglich, eine referenzierende Klasse vom Typ `org.eclipse.core.commands.AbstractHandler` anzugeben. Diese Klasse stellt eine Methode bereit, die beim Benutzen der Funktionalität aufgerufen wird und von der alle weiteren Berechnungen ausgehen. Durch sie wird die implementierungsseitige Schnittstelle angeboten. Alternativ können bestimmte Extensions eine referenzierende Klasse direkt angeben. Drei der gegebenen Funktionalitäten werden durch diese Einbettungsmethode implementiert, die Hyperlinks, die Erstellung der Syntaxbäume und die farbige Visualisierung der Regeln. Abbildung 3.8 zeigt die Anbindung einer Funktionalität der fiktiven Klasse `ExampleHandler`.

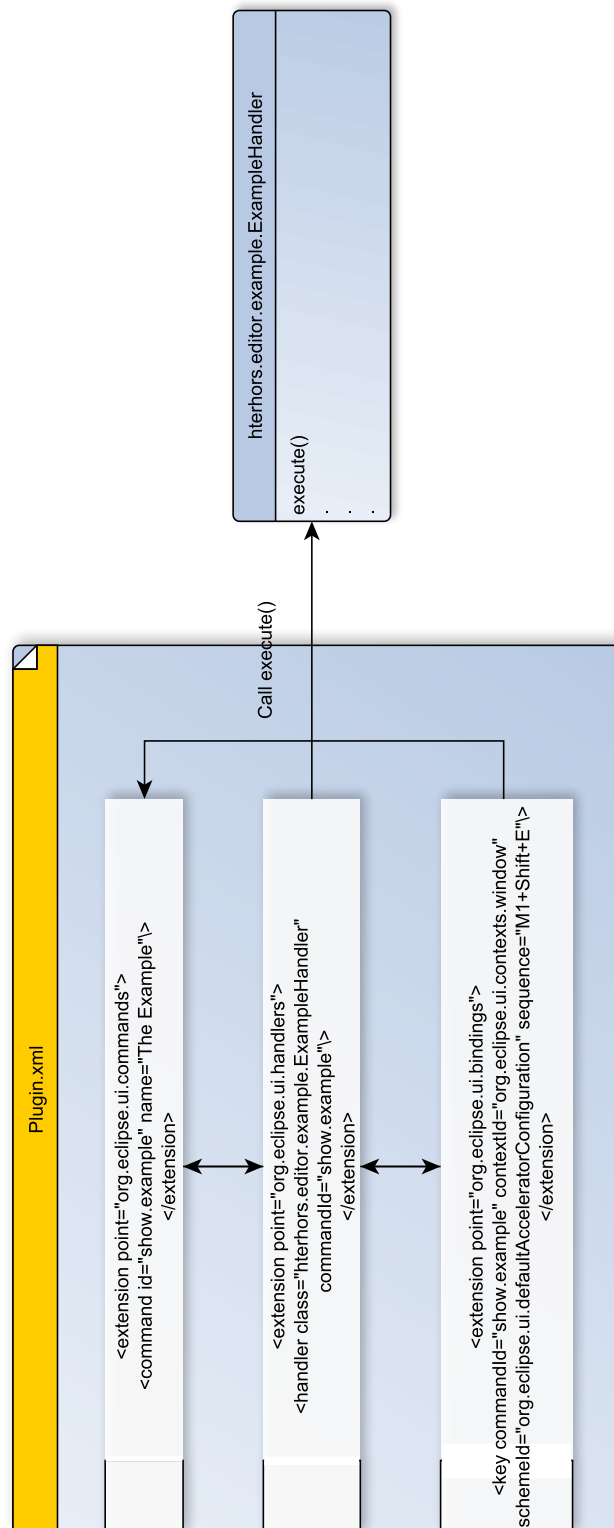


Abbildung 3.8: Diese Abbildung zeigt die Anbindung einer Funktionalität der Klasse `ExampleHandler` über Extension Points.

Hyperlinks Dem Editor wird es ermöglicht, durch den Extension Point `org.eclipse.ui.workbench.texteditor.hyperlinkDetectors`, auf die durch ein anderes Plugin bereit gestellte Funktionalität der Hyperlinks zuzugreifen und diese modifiziert zu verwenden. Das Hyperlink-System besteht aus zwei grundlegenden Java-Klassen.

Die Klasse `IsrHyperLinkDetector`, welche das Interface `org.eclipse.jface.text.hyperlink.IHyperlinkDetector` implementiert, wird als referenzierende Klasse direkt im Extension Point angegeben. Der Detektor enthält selbst nur eine wichtige Methode, `detectHyperlinks`. Sie überprüft ein Wort auf einen möglichen Hyperlink. Um dies zu realisieren, bekommt die Methode als Parameter die Position des Mauszeigers und kann dadurch das zu prüfende Wort ermitteln. Handelt es sich um ein Non-Terminal, wird daraufhin ein `IsrHyperLink` erstellt und über den Rückgabewert der Methode an Eclipse weitergeleitet.

`IsrHyperLink` ist die zweite benötigte Klasse. Sie implementiert das Interface `org.eclipse.jface.text.hyperlink.IHyperlink`. Durch dieses enthält sie die Methode `open`, die aufgerufen wird sobald der Nutzer einen Hyperlink betätigt. Sie bekommt als Parameter das angeklickte Element und ermittelt den zugehörigen Gegenpart des Hyperlinks. Existiert dieser, werden zwei Methoden des `ISREditors` aufgerufen. Die Methode `selectAndReveal` selektiert das gefundene Element und `setFocus` setzt auf diesen den Fokus des Editors. Ohne passenden Gegenpart bleibt die Aktion unberücksichtigt. Eine Veranschaulichung des Hyperlinksystems, in Form eines Klassendiagramms, ist in Abbildung 3.9 zu sehen.

Wortproblem und Syntaxbäume Da es bereits viele Algorithmen gibt, die einen Lösungsweg für das Wortproblem bereitstellen[5][6], wurde auf die Implementierung einer eigenen Lösung verzichtet und auf eine bestehenden zurückgegriffen. Dazu wurden selektiv Klassen aus dem Projekt *Grammar Editor* aus der *SOCS*⁴[7] benutzt.

Die Möglichkeit, Satzstrukturen gegen eine Grammatik zu validieren, ist nicht im Standard-Repertoire eines Editors enthalten. Es existiert also kein direkter Extension Point, der benutzt werden könnte, um die zu implementierende Funktionalität

⁴Science Of Computing Suit

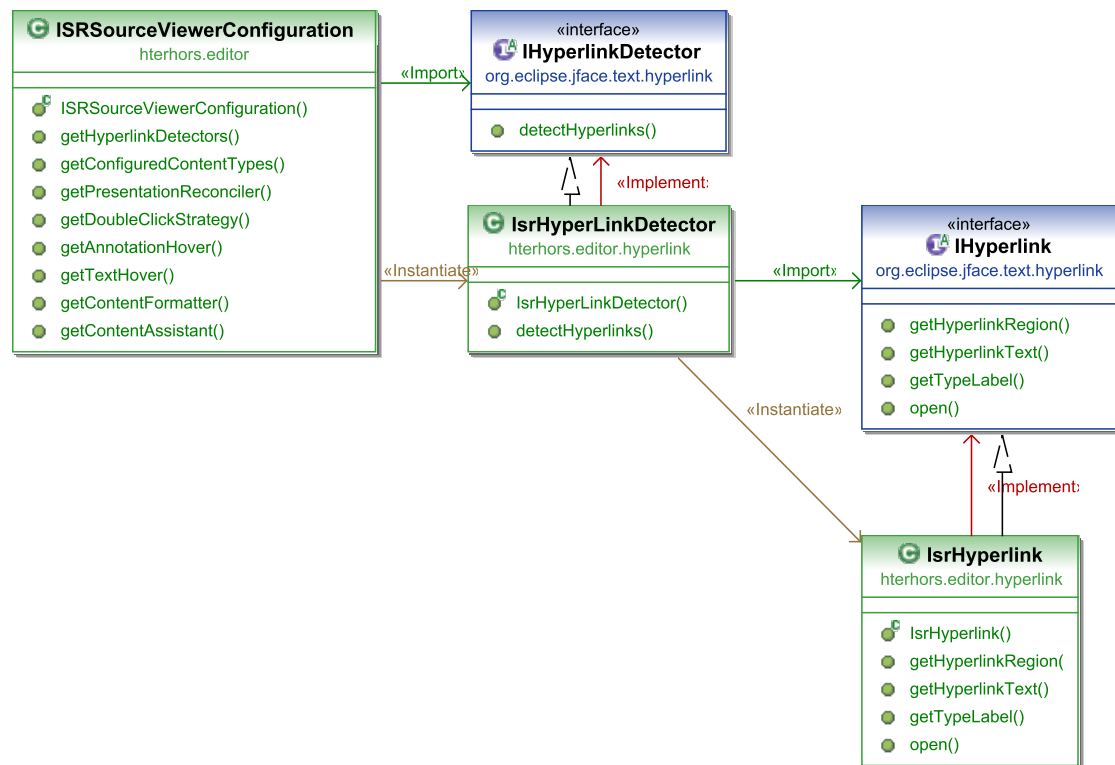


Abbildung 3.9: Klassendiagramm des Hyperlinksystems

anzubinden. Eine dennoch adäquate Art der Anbindung besteht darin, einen eigenen Menüpunkt im Eclipse-Menü zu erstellen, durch welchen die Funktionalität aufgerufen werden kann. Dazu wurde die Anbindung nach dem, in der Einleitung (siehe Kapitel 3.4.1) beschriebenen Verfahren realisiert. Die referenzierende Klasse, welche in dem Extension Point `org.eclipse.ui.handler` definiert wurde, ist die Klasse `SetGraphSettingsHandler`. Zur effizienteren Nutzbarkeit wurde der Extension Point `org.eclipse.ui.bindings` erstellt, in dem ein Tastenkürzel definiert und ebenfalls über die ID mit den anderen Extension Points verknüpft wurde. Des Weiteren wurde, um den erstellten Command mit dem Eclipse-Menü zu verbinden, der Extension Point `org.eclipse.ui.menus` erstellt und dieser erneut über die spezifische ID mit dem Command verknüpft. Auf der Seite des Editors

dient die Klasse `SetGraphSettingsHandler` als Handler⁵. In ihr wird zunächst die bestehende Grammatik in die klassische Form der kontextfreien Grammatiken gebracht. Dies bedeutet, dass dazu alle Besonderheiten und Zusatzzeichen wie in Kapitel 2.1.2 beschrieben, die keinen Einfluss auf die Struktur einer Regel haben, entfernt werden. Darüber hinaus wird die Grammatik so umgeformt, dass sie als Eingabe für die Klassen der benutzten Bibliothek verwendet werden kann, welche sich um die Lösung des Wortproblems und die Visualisierung der Syntaxbäume kümmern.

Beim Auslösen der Funktionalität, zum Beispiel beim Betätigen des Tastenkürzels, öffnet sich zunächst ein `org.eclipse.jface.InputDialog`, über welchen der Benutzer Sätze angeben muss, die bei Bestätigung durch die Klassen der eingebundenen Bibliothek auf das Wortproblem geprüft werden. Das Resultat ist eine Liste von booleschen Werten zum jeweiligen Satz. Des Weiteren, sofern das Wortproblem gelöst wurde, wird eine Visualisierung des Syntaxbaumes erstellt (siehe auch Abbildung 3.4).

Visualisierung von Regeln Die Visualisierung einer Grammatik wird als zusätzliches Fenster in der *Eclipse Workbench*⁶ angezeigt. Dies hat den Vorteil, dass sie nach Belieben vom Benutzer geöffnet oder geschlossen werden kann.

Auch diese Funktionalität befindet sich nicht im Pool der Extension Points, welcher von Eclipse bereitgestellt wird. Daher wird auch diese Funktionalität erneut durch einen Eintrag im Menü von Eclipse mit dem Plugin verankert. Die weitere Implementierung auf Seiten der `Plugin.xml` verläuft homogen zu der Implementierung der vorherigen Funktionalität, mit Ausnahme von zwei individuell angepassten Definitionen. Dabei handelt es sich um das Tastenkürzel und den als Schnittstelle fungierende Handler, `SetGraphSettingsHandler`. Die Methode `execute`, welche beim Benutzen aufgerufen wird, regelt alle weiteren Berechnungen. Zunächst wird, über die Methode `findView` mit einer ID die zugehörige Klasse `IsrGraphView` erstellt und mittels `showView` angezeigt. Sie implementiert das Interface `org.eclipse.ui.part.ViewPart`. Dadurch wird die Klasse zu einem wei-

⁵Als Handler werden die Java-Klassen bezeichnet, die als Schnittstelle zwischen einem Command innerhalb der `Plugin.xml` und der Implementation der Funktionalität dient.

⁶<http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Fconcepts%2Fconcepts-2.htm>

teren möglichen Fenster der Eclipse Workbench. Durch die Methode `showView` wird eine Instanz von `IsrGraphView` erstellt, wodurch automatisch die Methode `createPartControl` der Instanz aufgerufen wird. Die Funktion dieser Methode stellt das Errechnen des visuellen Kontextes dar. Dazu bedient sie sich der statischen Methode `IsrGraphBuilder.getGraph`, die aus der aktuellen Grammatik mittels des Zest-Frameworks einen Graphen erstellt, der im geöffneten Fenster angezeigt wird (siehe Abbildung 3.4).

3.4.2 Funktionalität über Aktionen

Ein weiterer Weg, zusätzliche Funktionalität in das Plugin einzubinden besteht darin, sie in der Methode `createActions` zu definieren, welche im `ISREditor` bereitgestellt wird. Durch die Definition innerhalb des Editors werden zwei Effekte erzielt. Der eine Effekt ist, dass die Funktionalität zum Standard des Editors wird. Dies bedeutet, dass sie editorabhängig ist. Der andere Effekt ist die Verknüpfung mit der globalen Aktion von Eclipse, sodass über diesen Weg die global Funktionalität von der lokalen, also selbst implementierten, überschrieben wird. Ein Vorteil der hier beschriebenen Methode ist, dass unter anderem Tastenkürzel und Menüeinträge automatisch übernommen werden. Um die globale Funktionalität zu überschreiben erfordert diese Vorgehensweise eine Installation der definierten Aktion, die in der Klasse `ISREditorContributor` vorgenommen wird. Eine konkrete Implementierung erfolgt in der `ISRSourceViewerConfiguration`, in dem die funktionalitätsspezifische Methode der Oberklasse überschrieben wird. Die überschriebene Methode wird dann von Eclipse aufgerufen, sobald auf die Funktionalität zugegriffen wird.

Automatische Formatierung Zur eigenen Implementierung einer automatisierten Formatierung wurde die Methode `getContentFormatter` überschrieben. Sie erstellt eine Instanz der Klasse `org.eclipse.jface.formatter.ContentFormatter`. Bevor die Methode den erstellten Formatierer zurückgibt, müssen zuvor definierte Formatierungsstrategien mit dem Formatierer verknüpft werden. Dazu wird auf die, in der Partitionierung erstellten content types, zurückgegriffen. Zu jedem content type wurde eine eigene Formatierungsstrategie entworfen und durch die Me-

thode `setFormattingStrategy` dem Formatierer einschließlich der dazugehörigen Strategie zugewiesen.

Die `IgnoreFormattingStrategy` wird benutzt, wenn es sich bei dem content type um `ISR_IGNORE` handelt. Da dieser Partition-Abschnitt keine besonderen Symbole oder komplizierte Konstrukte enthalten kann (siehe Kapitel 2.1.2), da innerhalb einer Ignore-List nur Terminale erlaubt sind, folgt daraus eine sehr einfache Strategie. Alle einfach oder mehrfach hintereinander auftretenden *Whitespaces*⁷ werden durch ein einzelnes Leerzeichen ersetzt. Zusätzlich werden am Ende des Abschnittes zwei Umbrüche hinzugefügt. Diese gewährleisten einen Abstand zum nächsten Abschnitt.

Wird die `RuleFormattingStrategy` eingesetzt, handelt es sich bei dem Abschnitt um einen Partition-Abschnitt mit dem content type `ISR_RULE`. Zuerst werden auch hier alle überflüssigen Whitespaces entfernt und gegen ein Leerzeichen ersetzt. Des Weiteren wird vor jedem in der Regel auftretenden Oder-Symbol ein Umbruch hinzugefügt. Die Wortlänge des im Abschnitt darüber liegenden Wortes dient im zweiten Schritt als Anhaltspunkt dafür, wie weit das Oder-Symbol in der neuen Zeile nach rechts verrückt wird; auch dieser Abschnitt wird mit zwei Umbrüchen abgeschlossen.

Die `DefaultFormattingStrategy` eliminiert überflüssige Whitespaces zwischen den Regeln. Der daraus resultierende Effekt ist, dass der Abstand zwischen den Regeln konstant bleibt.

Beim Aufruf der automatischen Formatierung, wird jeder Abschnitt mit der passenden Formatierungsstrategie formatiert. Das Klassendiagramm in Abbildung 3.10 verdeutlicht noch einmal die Beziehungen zwischen den benutzten Klassen.

Content Assistant Sobald die Funktionalität des Content Assistant aufgerufen wird, greift Eclipse auf die Methode `getContentAssistant` zurück. Diese kümmert sich um die Erstellung einer Instanz des `IsrContentAssistant`. Zur Inhaltsberechnung werden auf diesem Assistenten Prozessoren registriert. Da der Assistent, ebenfalls wie der Formatierer, vom Typ der jeweiligen Partition abhängig ist, wird

⁷Als Whitespaces werden alle Zeichen einer Programmiersprache bezeichnet, die nicht gesehen werden aber dennoch (Speicher-)Platz beanspruchen. Darunter fallen zum Beispiel Leerzeichen, Umbrüche und Tabulatorzeichen.

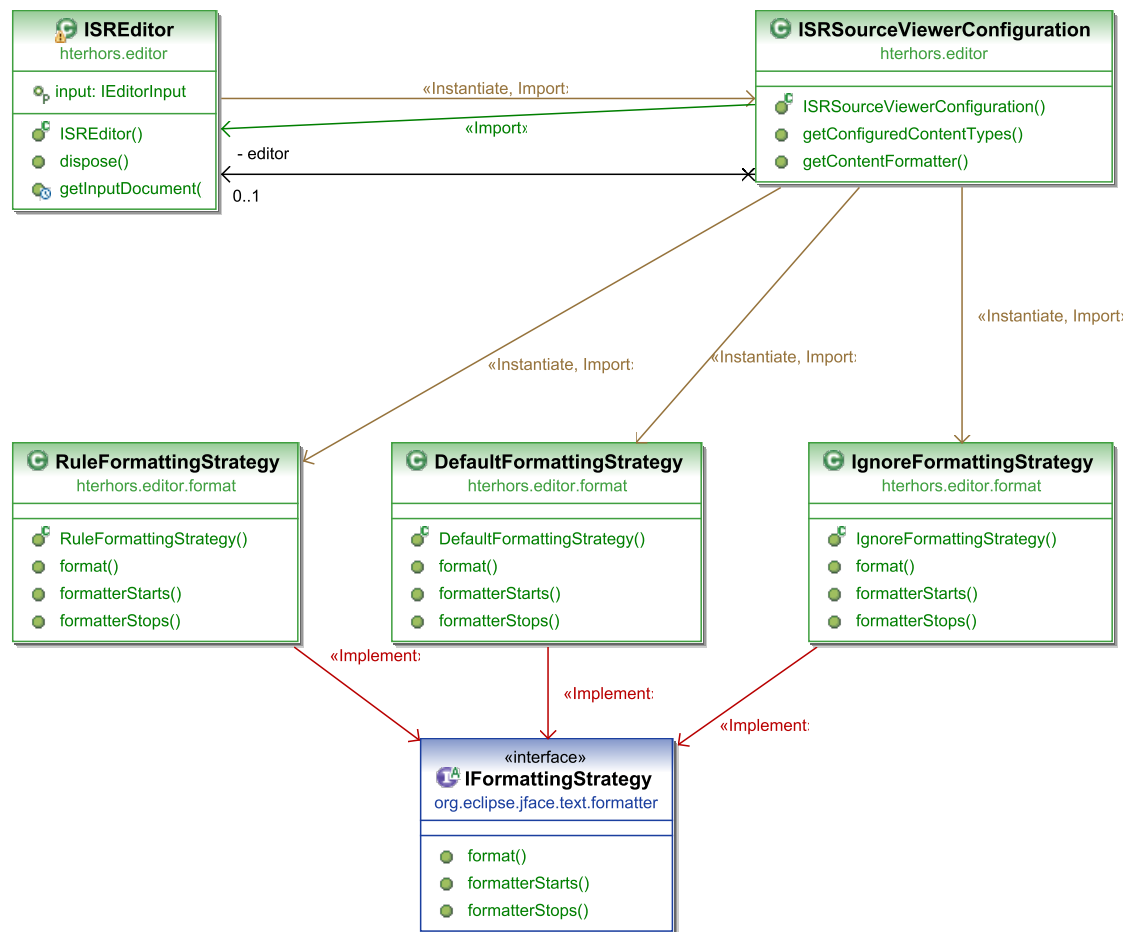


Abbildung 3.10: Diese Abbildung zeigt das Klassendiagramm des Systems der automatischen Formatierung.

jedem Prozessor ein Zuständigkeitsbereich in Form eines content type zugewiesen.

Es wurden also zu den drei existierenden, die drei Prozessoren `ISRContentAssistantIgnoreRuleProcessor`, `ISRContentAssistantDefaultProcessor` und `ISRContentAssistantRuleProcessor` erstellt. In der Methode `computeCompletionProposals`, welche sich durch das implementierte Interface `org.eclipse.jface.text.contentassist.IContentAssistProcessor` in jeder Prozessorklasse befindet, werden die spezifischen Vorschläge errechnet und dem `ISRContentAssistant` bereitgestellt.

3.4.3 Funktionalität über direkte Implementierung

Des Weiteren kann Funktionalität auch durch eine direkte Implementierung im `ISREditor` oder in der `ISRSourceViewerConfiguration` erfolgen, ohne einer vorherigen Definition und Installation der Aktionen. Dazu wurden entweder die im Editor vorhandenen Methoden oder wie im Abschnitt 3.4.2 zuvor, die Methoden der `ISRSourceViewerConfiguration` erweitert oder überschrieben. Auch diese Funktionalitäten sind editorspezifisch, im Gegensatz zu installierten Funktionen haben die hier implementierten eher passive Eigenschaften.

Fehlermarkierung Die Fehlermarkierung wird direkt bei der Erstellung einer Instanz des `ISREditor`s initialisiert. Dazu wird ein Thread gestartet, der parallel zum Editor läuft und in regelmäßigen Abständen die Methode `validateAndMark` aufruft. Um ein überflüssiges Prüfen zu vermeiden und die Fehlermarkierung damit effizienter zu machen, ist der Thread nur dann aktiv, wenn der Inhalt des Editors verändert wird. Die Methode `validateAndMark` erstellt eine Instanz der Klasse `MarkingErrorHandler`, die den Inhalt des Editors vom Typ `org.eclipse.core.IFile` präsent hat, und ruft auf ihr nacheinander zuerst die Methode `removeExistingMarker` und darauf folgend, die Methode `setCurrentMarker` auf. Damit wird erreicht, dass alle bereits existierenden Marker zuerst entfernt und anschließend die neu errechneten hinzugefügt werden.

`setCurrentMarker` ruft dazu die statischen Methoden `MarkerContainer.getWarnings` und `MarkerContainer.getErrors` auf. Diese liefern jeweils eine Liste mit den aktuellen Markern zurück, die im darauf folgendem

Schritt der `IFile`-Instanz hinzugefügt werden.

Die Berechnung der Marker innerhalb des `MarkerContainer` wird durch die Hilfs-Klasse `IsrRuleValidator` realisiert. Sie implementiert verschiedene Methoden, die Fehler und Warnungen in der Grammatik ausfindig machen. Diese Methoden werden iterativ aufgerufen und erkannte Fehler und Warnungen in zwei separate Listen gespeichert. Im Folgenden werden diese Methoden beschrieben. `multipleDeclarations` überprüft die Grammatik auf doppelte Deklarationen. `checkValidStartSymbol` prüft das Vorkommen eines gültigen Startsymbols, sie erkennt aber auch doppelte Startsymbole.

`hasAllNonTerminalsDeclared` überprüft, ob alle verwendeten Non-Terminals eine gültige Deklaration haben. `checkSyntax` prüft die Grammatik auf Syntaxfehler und unzulässige Zeichenfolgen. `warnUnusedNonterminalDeclaration` schaut nach nicht verwendeten Deklarationen. `warnSameNamedNTAndT` schaut nach Terminalen, welche gleichnamig sind wie existierende Non-Terminals.

Hoverinformation Die Verknüpfung der Hoverinformation wird durch die Überschreibung der Methode `getTextHover` innerhalb der `ISRSourceViewerConfiguration` realisiert. Diese wird beim Benutzen der Funktionalität von Eclipse aufgerufen. Sie erstellt eine Instanz von `GenerateHoverInfos`, welche durch ihr Interface `org.eclipse.jface.text.ITextHover` die Methode `getHoverInfo` bereitstellt. Diese dient zur Berechnung der anzuzeigenden Information. Die Berechnung erfolgt in zwei Schritten, zunächst wird aus der Position des Mauszeigers, welche die Methode bei ihrem Aufruf übergeben bekommt, das vom Nutzer gemeinte Regelement ermittelt. Handelt es sich bei dem Element um ein Non-Terminal, so besteht der Rückgabewert aus der Regel, welche durch die zugehörige Deklaration definiert ist; dessen Ermittlung den zweiten Schritt darstellt. Ist dies nicht der Fall, wird geprüft, ob das Wort von einer Fehlermarkierung umgeben ist. Besteht sie, wird alternativ über den `MarkerContainer` die Fehlerausgabe erworben und diese zurückgegeben. Trifft keines von beidem zu, liefert die Methode den Wert `null` und es wird keine Information angezeigt.

Syntaxhighlighting Um die Hervorhebung der Syntax aktuell zu halten, wird die Methode `getPresentationReconciler` in der `ISRSourceViewerConfiguration`

überschrieben. In ihr wird ein `PresentationReconciler` erstellt, der aus einem Fehlererkenner dem `PresentationDamager` und einem Korrektor dem `PresentationRepairer` besteht. Beide Klassen werden in dem `org.eclipse.jface.rules.DefaultDamageRepairer` zusammengefasst. Der `PresentationDamager` hat die Aufgabe, neu eingefügten Code aufzuspüren und die betroffene Stelle in der Regel dem `PresentationRepairer` mitzuteilen, so dass diese defekte Stelle repariert werden kann. Ähnlich zur Partitionierung arbeitet der `DefaultDamageRepairer` nach spezifizierten Regeln, welche die Farbe der Darstellung des jeweiligen Regelelementes festlegen. Alle benutzen Regeln sind in der Klasse `ISRSyntaxScanner` beschrieben, welche von `org.eclipse.jface.text.rules.RuleBasedScanner` ableitet. Da die Syntax-Hervorhebung partitionsabhängig ist, benötigt sowohl der Damager als auch der Repairer nicht nur den `ISRSyntaxScanner` und dadurch alle enthaltenen Regeln, sondern auch den content type des Partition-Abschnittes, in dem die Änderung vorgenommen werden soll.

4 Evaluation

Obwohl die Anforderungen an die Software aus einer durchdachten Anforderungsanalyse hervorgegangen sind, und darüber hinaus bei dem Entwurf der dazugehörigen Funktionalitäten auf zusätzlich externe Quellen eingegangen wurde, lässt sich nicht ausschließen, dass das Programm Fehler beinhaltet oder in der Nutzung optimiert werden könnte. Zudem ist es möglich, dass weitere Funktionalitäten von Vorteil sind oder dass bereits bestehende zu kompliziert realisiert wurden.

Eine Evaluation der entwickelten Software soll auf diese Fragen eine Antwort geben. Dieses Kapitel gliedert sich in vier Abschnitte. Im ersten Abschnitt wird auf die Zielstellung der Studie eingegangen. Der zweite Abschnitt beschreibt die Methoden, die zur Evaluation benutzt wurden. Eine Beschreibung der Durchführung findet sich im dritten Abschnitt, im letzten werden die Ergebnisse vorgestellt und diskutiert.

4.1 Ziele

Der Zweck der entwickelten Software ist es, dem Nutzer durch die gebotenen Funktionalitäten die Möglichkeit zu geben, einfacher, robuster und schneller mit der ISR-Grammatik umgehen zu können (siehe Kapitel 2.4). Daher liegt es nahe, den Einsatz dieser Funktionalitäten in einer Studie zu prüfen. Ein weiteres Ziel der Studie resultiert aus der Ungewissheit, ob die Anforderungsanalyse vollständig war. Auch wenn durch einen konkreten Anwendungsfall und einer Analyse der Probleme viele Anforderungen entstanden sind, ist nicht sicher, ob die aufgestellten Anforderungen und die daraus resultierenden Funktionalitäten, den Problemen der Praxis genügen. Des Weiteren zeigt die Evaluation Problemstellungen auf, die das Programm nicht oder nicht optimal unterstützen.

Doch auch positive Aspekte können durch sie hervorgehoben werden, indem gezeigt wird, dass die eingesetzten Funktionalitäten genau den Anforderungen entsprechen, für die sie implementiert wurden.

4.2 Methoden und Durchführung

Um die Software angemessen evaluieren zu können, wurde eine aufgabenbasierte Methodik gewählt. Dazu wurden aus dem Anwendungsfall in Kapitel 2.3, Aufgabenarten extrahiert und zu jeder Art eine Aufgabe erstellt. Dabei bestand viel Spielraum, sodass beim Erstellen der Aufgaben zusätzlich darauf geachtet werden musste, diese so realistisch wie möglich zu halten. Um anschließend einen Vergleich aufstellen zu können, wurden die Aufgaben von der Hälfte der Probanden mit und von der anderen Hälfte ohne Unterstützung des Plugins gelöst. Zusätzlich, um die definierten Ziele aus Kapitel 4.1 zu erreichen, beantworteten die Probanden nach der Bearbeitung der Aufgaben einen Fragebogen.

Bei der Erstellung des Fragebogens, wurde auf bereits bewährte Techniken¹ zurückgegriffen. Anschließend wurde eine Testphase mit einem Testprobanden durchgeführt. In dieser wurden sowohl die gestellten Aufgaben als auch die Fragen bearbeitet, um Unklarheiten herauszufinden. Die Auffälligkeiten innerhalb der Aufgabenstellung und des Fragebogens wurden anschließend geändert und einer weiteren Testperson zur Kontrolle gegeben. Sowohl der Fragebogen wie auch der Aufgabenbogen befinden sich im Anhang.

In der Durchführung erhielten alle Probanden drei Aufgaben, die sie nacheinander lösen sollten. In der ersten Aufgabe mussten die Teilnehmer eine schon bestehende Grammatik, welche Syntaxfehler beinhaltete, korrigieren, ohne dabei die syntaktische Struktur zu verändern; semantische Fehler sollten nicht behoben werden. In der zweiten Aufgabe musste eine Grammatik erstellt werden, die einen Pool von Sätzen abbilden sollte. Für die dritte Aufgabe wurde erneut ein Pool von Sätzen gegeben, zu denen das Wortproblem gegen eine schon bestehende Grammatik zu prüfen war. Zudem sollte von einer Satzstruktur ein Syntaxbaum erstellt werden.

¹http://iss.leeds.ac.uk/info/312/surveys/217/guide_to_the_design_of_questionnaires/5

Bevor mit der ersten Aufgabe begonnen werden konnte, bekam jeder Teilnehmer eine verbale Einführung in die ISR-Grammatik. Dazu wurde an Hand der Beispielgrammatik aus Kapitel 2.1.3 und der im Fragebogen enthaltenen Syntaxspezifikation die Struktur der Grammatik und die Bedeutung aller Regelemente erklärt. Bei der Wahl der Probanden wurde darauf geachtet, dass sie über die nötigen Kenntnisse des benutzen Editors verfügten. Um die Auswertung besser vergleichen zu können, wurde die Bearbeitungszeit der Probanden gestoppt. Des Weiteren wurden während der Bearbeitung Notizen zum Verhalten des Teilnehmers gemacht. Ferner durften sie während der Bearbeitung Fragen stellen, die allen Teilnehmern in gleicher Weise beantwortet wurden, um die Aussagekraft der Ergebnisse nicht zu beeinflussen.

4.3 Auswertung

Die Teilnehmer hatten zur Beantwortung der Fragen eine Bewertungsskala von einem bis maximal fünf Punkten zur Verfügung. Die Vergabe von fünf Punkten bedeutete eine sehr gute Bewertung, ein Punkt hingegen zeigte, dass viel Verbesserungspotenzial gesehen wurde.

Kenntnisse der gegebenen Hilfsmittel Zunächst wurden fünf allgemeine Fragen gestellt. Die ersten zwei bezogen sich auf die Teilnehmerkenntnisse im Umgang mit den gegebenen Werkzeugen. Die durchschnittliche Punktzahl der Probanden auf die Frage, wie gut sie mit einem Computer umgehen können, lag bei ungefähr 4,8. Die Frage, wie gut sie mit dem verwendeten Editor umgehen können, sei es Eclipse oder ein anderer Editor, beurteilten 90% der Befragten mit vier oder mehr Punkten. Die ausgeglichenen Kenntnisse über die Werkzeuge bildeten eine gute Voraussetzung für die Bearbeitung und anschließende Auswertung der gestellten Fragen.

Erfahrung mit kontextfreien Grammatiken Zudem war es wichtig zu wissen, wie viel Erfahrung die Teilnehmer im Umgang mit kontextfreien und speziell mit der ISR-Grammatik hatten. Das Wissen über kontextfreie Grammatiken lag im Schnitt deutlich höher als das Wissen über die ISR-Grammatik. Dieses Defizit

sollte durch die in Kapitel 4.2 beschriebene Einführung in die ISR-Grammatik unter Einbezug des Wissens über kontextfreie Grammatiken abgebaut werden. Im weiteren Verlauf der Studie zeigte sich, dass unabhängig vom Vorwissen und Erfahrung, die Aufgaben etwa gleich gut bewältigt werden konnten. Dies stellt ein erstes positives Indiz für eine gelungene Umsetzung der Anforderungen dar, welches durch die weiteren Ergebnissen bestätigt wird.

Verständnis und Realitätsbezug der Aufgabenstellung Die Aufgabenstellung wurde von allen Teilnehmern mit durchschnittlich 4,6 Punkten sehr gut verstanden. Eine wichtige Zusatzfrage, die nur von den Teilnehmern beantwortet werden konnte und sollte, die bereits Erfahrung mit der ISR-Grammatik gemacht haben, bildete die Frage nach dem Realitätsbezug der gestellten Aufgaben. 90% gaben an, dass sie die gestellten Aufgaben mit vier oder mehr Punkten als sehr realistisch einschätzen. Da die Aufgaben größtenteils aus dem Anwendungsfall aus Kapitel 2.3 extrahiert wurden, zeigt auch dies, dass der Anwendungsfall realistisch konstruiert wurde.

Im Folgenden werden nun die Antworten der drei Aufgaben ausgewertet und diskutiert. Zunächst wird jede Aufgabe aus Sicht der Plugin-Nutzer ausgewertet (Gruppe A), dann aus Sicht derer, die ohne Plugin gearbeitet haben (Gruppe B). Anschließend werden die beiden Ergebnisse verglichen und diskutiert.

4.3.1 Korrektur einer fehlerhaften Grammatik

Das Ziel dieser Aufgabe war es zu untersuchen, wie leicht syntaktische Fehler einer Grammatik erkannt und behoben werden können. Da es viele verschiedene Wege gab, die Aufgabe zu lösen, wurde die Bedingung gestellt, dass die Teilnehmer so wenig wie möglich an der syntaktischen Struktur der Grammatik ändern durften.

Die Benutzer des Plugin konnten sehr schnell einen Überblick über alle syntaktischen Fehler erlangen und beantworteten die Frage, wie leicht es ihnen fiel, im Schnitt mit 4,5 Punkte, wie in Abbildung 4.1 zu sehen ist. Dabei spielte vor allem die Funktionalität der Fehlermarkierung eine wichtige Rolle. Weniger als im Vorfeld zunächst angenommen, wurde für diese Aufgabe das Syntaxhighlighting als hilfreich erachtet. Der Mittelwert der benötigten Zeit lag in der Gruppe A bei

4,5 Minuten; hier konnte eine relativ breite Varianz von vier Minuten beobachtet werden. Manche Teilnehmer versuchten die Aufgabe schnell zu lösen, wohingegen andere stärker die Funktionalität des Editors nutzten und dazu geringfügig mehr Zeit benötigten. Das Ergebnis von durchschnittlich null Fehlern ist aber ein deutliches Indiz dafür, dass das Plugin für diese Art der Aufgabe eine sehr gute Unterstützung bot.

Auffallend war, dass die Teilnehmer, die ohne Plugin arbeiteten zwar langsamer waren aber ebenfalls eine relativ schnelle Bearbeitungszeit hatten. Sie wurden jedoch durchgehend von einer Unsicherheit über den aktuellen Stand der Aufgabe begleitet. Die Teilnehmer fragten mehrfach nach, ob die Aufgabe jetzt fertig bearbeitet sei. Die Durchschnittszeit lag bei 6,9 Minuten mit einer hohen Varianz von 7,7 Minuten. Diese Varianz erklärt sich dadurch, dass manche Probanden sich ihrer Fähigkeit, die Aufgabe gut lösen zu können sehr sicher waren, andere hingegen sehr zögerlich arbeiteten und die Grammatik mehrfach überprüften. Keinem Teilnehmer gelang es jedoch, die Fehler vollständig aus der Grammatik zu entfernen. Dementsprechend wurde die Frage, wie leicht die Aufgabe gelöst werden konnten, im Mittel mit drei Punkten beantwortet (siehe Abbildung 4.1).

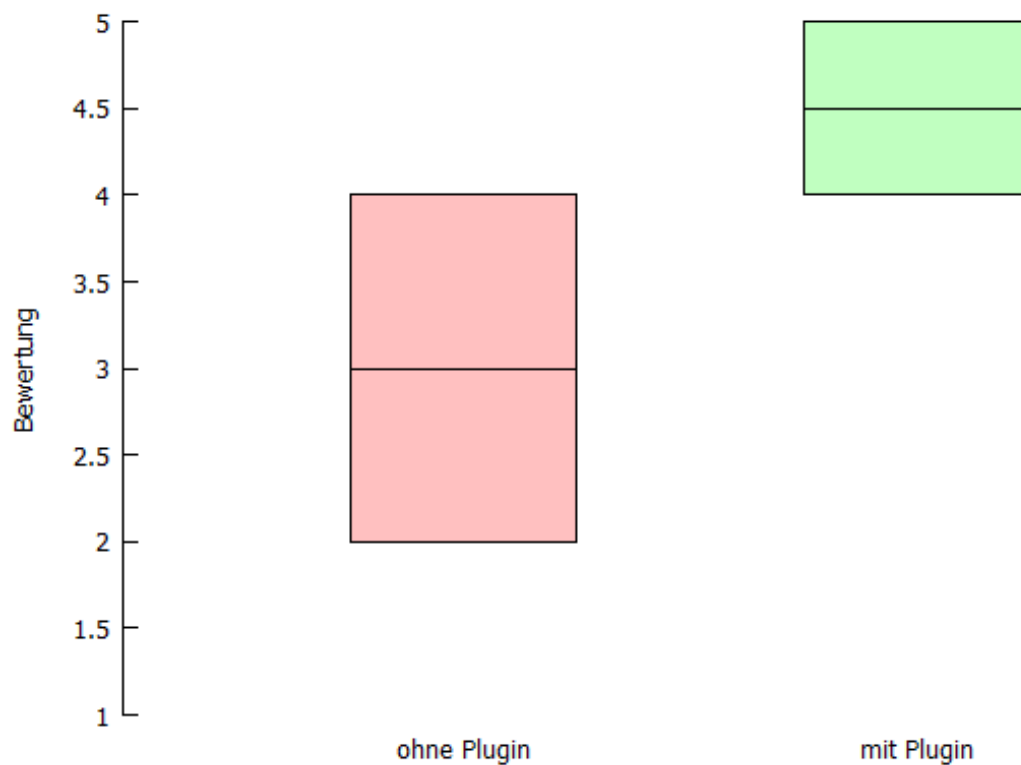


Abbildung 4.1: Zeigt einen Vergleich der Bewertungen wie einfach die erste Aufgabe gelöst werden konnte. Rechts die Gruppe A, links die Gruppe B.

Im direkten Vergleich der Ergebnisse beider Gruppen fällt deutlich auf, dass sich die eingesetzte Software positiv auf die Bearbeitungszeit (siehe Abbildung 4.2) und wie in Abbildung 4.7 zu sehen ist auch positiv auf die Fehlerrate auswirkte. Des Weiteren wurden Unsicherheiten im Erkennen von Fehlern in der Grammatik schneller beseitigt und dadurch das Lösen der Aufgabe erleichtert. Darüber hinaus bot die Software eine gute Hilfe, um eine angemessene Übersicht über den aktuellen Stand der Aufgabe zu erlangen. Dies wurde auch dadurch bestätigt, dass kein Teilnehmer der Gruppe A während der Bearbeitung dahingehend Fragen stellte. Im Gegensatz dazu stellten die Teilnehmer der Gruppe B durchschnittlich ein bis zwei Fragen.

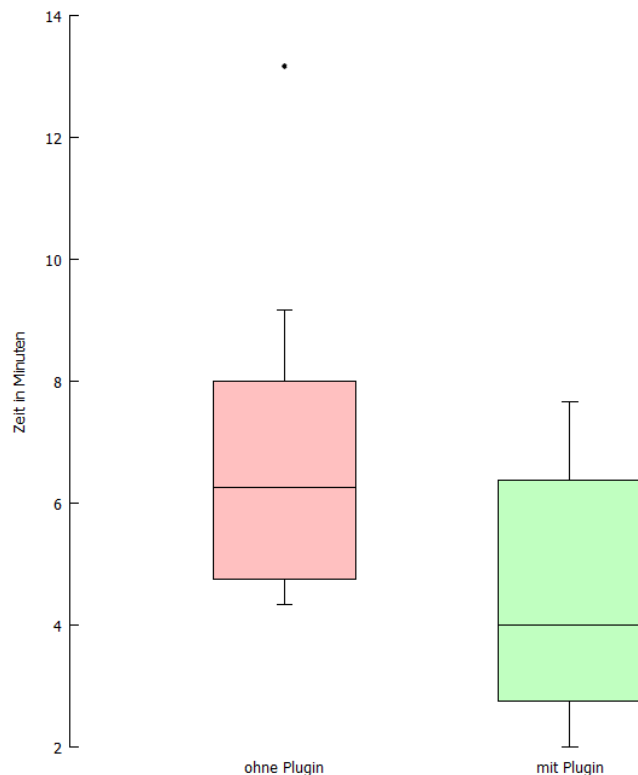


Abbildung 4.2: Zeigt einen Vergleich der Bearbeitungszeiten der ersten Aufgabe, in der es darum ging, eine fehlerhafte Grammatik zu korrigieren. Rechts die Gruppe A, links die Gruppe B.

4.3.2 Erstellen einer ISR-Grammatik

In der zweiten Aufgabe sollte eine Grammatik erstellt werden, die einen vorgegebenen Pool von Sätzen abbildet.

Zunächst werden die Ergebnisse der Gruppe A ausgewertet. Allen Teilnehmern fiel es sehr leicht, die Aufgabe zu lösen, wie die durchschnittliche Punktevergabe von 4,5 mit einer Varianz von 0,5 zeigt (siehe Abbildung 4.3). Dies ist unter anderem auf das Syntaxhighlighting, aber auch auf die zur Verfügung stehende Fehlermarkierung zurückzuführen. Da sich die Teilnehmer auch mit weiteren Funktionalitäten des Editors vertraut machten, wie zum Beispiel dem Content Assistant, gab es anfänglich Zeitverzögerungen. Dies ist wahrscheinlich für die relativ lange Bearbeitungszeit von durchschnittlich 6,4 Minuten verantwortlich. Positiv war zu beobachten, dass die Teilnehmer die Aufgabe fehlerfrei lösen konnten.

Die Benutzung des Standardeditors führte bei fast allen zu Problemen, sodass nur 12,5% , wie in Abbildung 4.7 zu sehen ist, die Aufgabe fehlerfrei lösten. Neben Syntaxfehlern innerhalb der Grammatik wurden des Öfteren Worte vergessen, so dass bestimmte Sätze nicht von der Grammatik abgebildet werden konnten. Diese oberflächliche Bearbeitung der Aufgabe ist der Grund, weshalb die Teilnehmer der Gruppe B zum Lösen im Mittel nur etwa 5,5 Minuten benötigten. Dass auch hier während der Bearbeitung eine Unsicherheit vorhanden war, lässt sich daran erkennen, dass nur zwei der Befragten die Aufgabe sehr gut lösen konnten und der Mittelwert trotzdem nur bei 3,4 Punkten liegt, wie in Abbildung 4.3 ersichtlich.

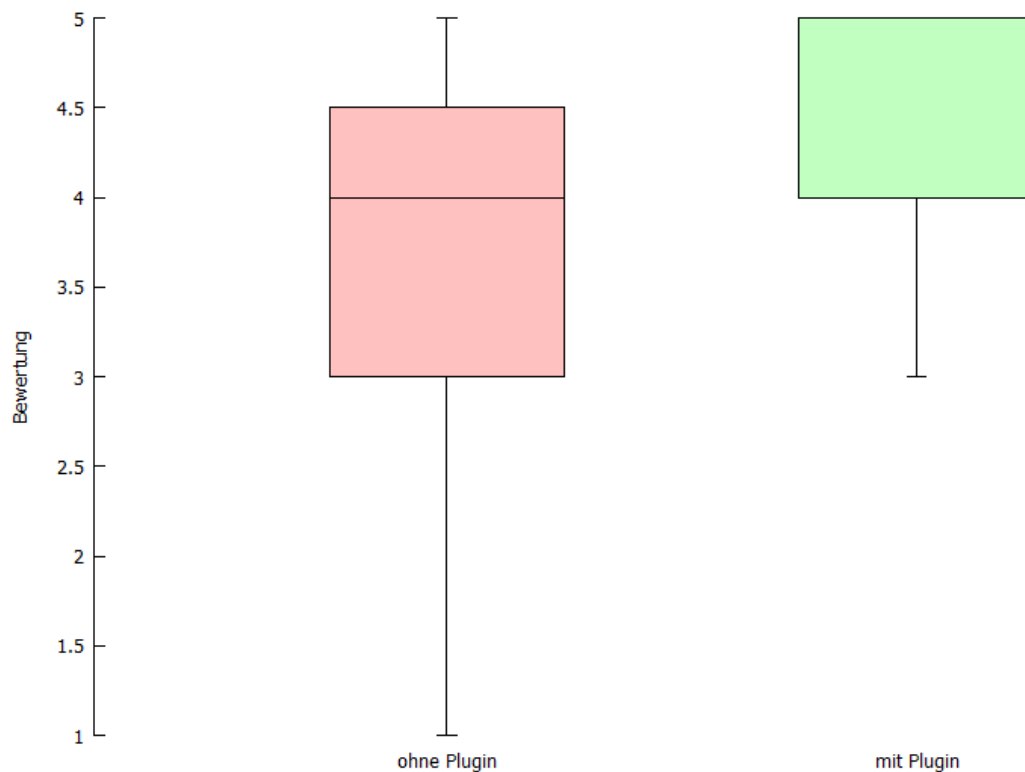


Abbildung 4.3: Zeigt einen Vergleich der Bewertungen wie einfach die zweiten Aufgabe gelöst werden konnte. Rechts die Gruppe A, links die Gruppe B.

In der Auswertung zeigte sich die zusätzliche Auffälligkeit, dass sich durch Benutzung der Software eine längere Zeit zum Lösen der Aufgabe ergab (Abbildung 4.4). Die Ursachen liegen vermutlich darin, dass die Teilnehmer zum einen das er-

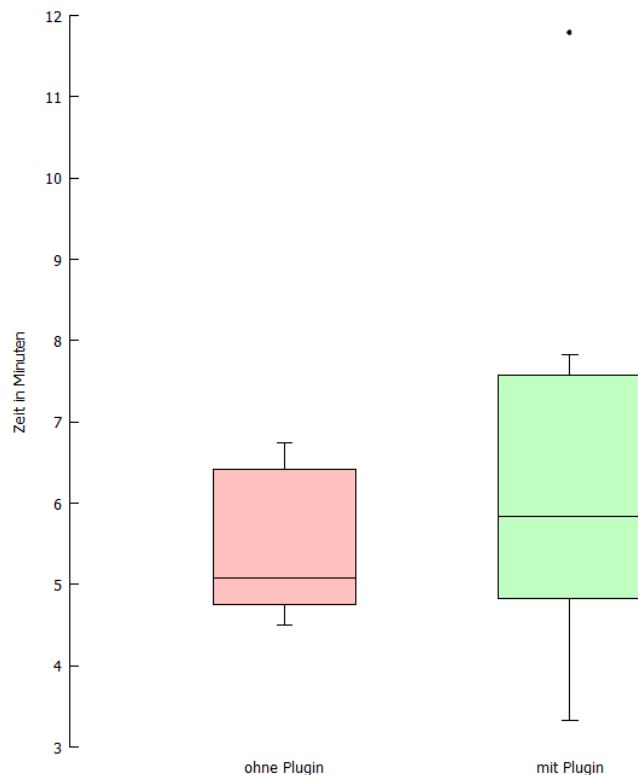


Abbildung 4.4: Zeigt einen Vergleich der Bearbeitungszeiten der zweiten Aufgabe, in der es darum ging, eine Grammatik für bestimmte Sätze zu bilden. Rechts die Gruppe A, links die Gruppe B.

ste Mal mit dem Editor praktisch arbeiten mussten und sich in der Funktionsweise nicht sicher auskannten. Zum anderen ließen sich die Teilnehmer häufig durch die angebotenen Funktionalitäten des Editors ablenken. Dies würde nach einer längeren Benutzungsphase abflachen. Eine wichtigere Rolle als der Zeitfaktor spielte jedoch die Anzahl der auftretenden Fehler, welche die Teilnehmer während der Bearbeitung machten. Dass auch bei dieser Aufgabe durch die Benutzung des Plugins im Schnitt null Fehler gemacht wurden, zeigt deutlich, dass der Editor auch zur Erstellung einer ISR-Grammatik einen geeigneten Assistenten darstellt und dem Anwender Sicherheit gibt.

4.3.3 Lösen des Wortproblems und Erstellen von Syntaxbäumen

Die letzte der drei Aufgaben befasste sich mit dem Lösen des Wortproblems. Die Teilnehmer bekamen erneut einen Pool von Sätzen und sollten anhand einer ebenfalls vorgegebenen Grammatik prüfen, welche Sätze durch die Grammatik erstellt werden können. Einer der gefundenen Sätze sollte zusätzlich als Syntaxbaum dargestellt werden.

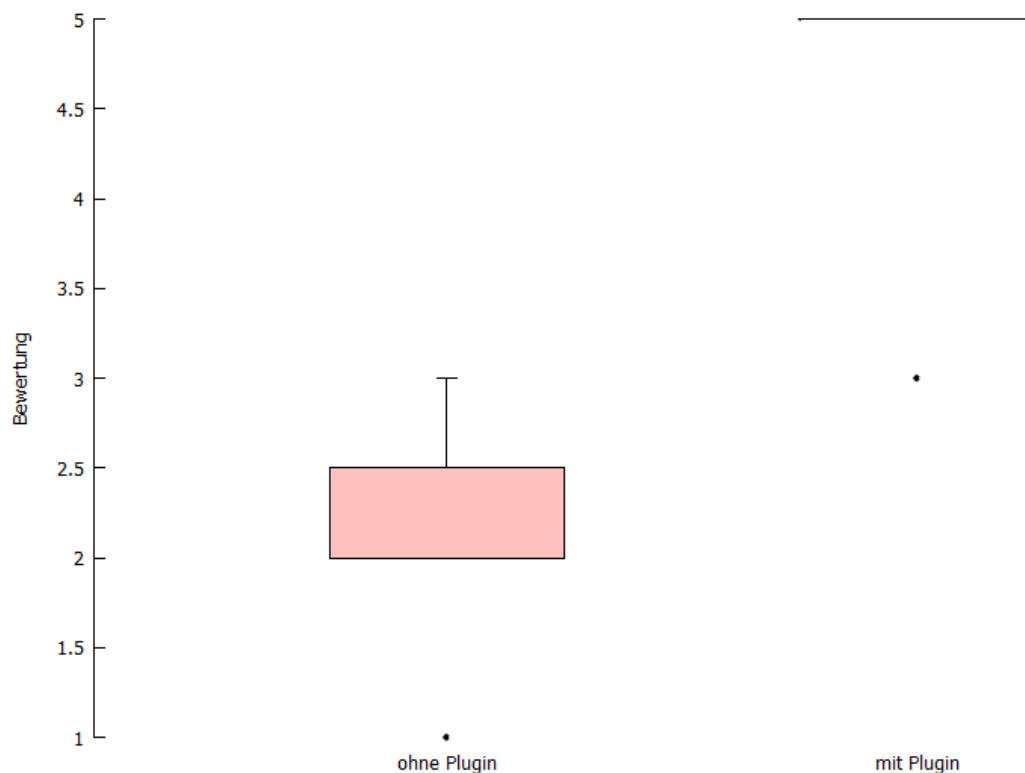


Abbildung 4.5: Zeigt einen Vergleich der Bewertungen wie einfach die dritten Aufgabe gelöst werden konnte. Rechts die Gruppe A, links die Gruppe B.

Für die Teilnehmer, die das Plugin zur Verfügung hatten, war diese Aufgabe schnell gelöst. Im Durchschnitt benötigten sie 4,6 Minuten, mit einer Varianz von 0,4 Minuten. Die sehr geringe Varianz ist dadurch bedingt, dass für Gruppe A diese Aufgabe lediglich darin bestand, die Sätze in ein Dialogfenster einzugeben

und vom Plugin prüfen zu lassen. Dies ist auch der Grund dafür, dass während dieser Aufgabe im Durchschnitt null Fehler gemacht wurden. Dementsprechend wurde die Frage hinsichtlich der leichten Lösbarkeit der Aufgabe von 87,5% der Gruppe A mit fünf Punkten bewertet. Die Abbildung 4.5 verdeutlicht den Vergleich in einem Boxplot-Diagramm.

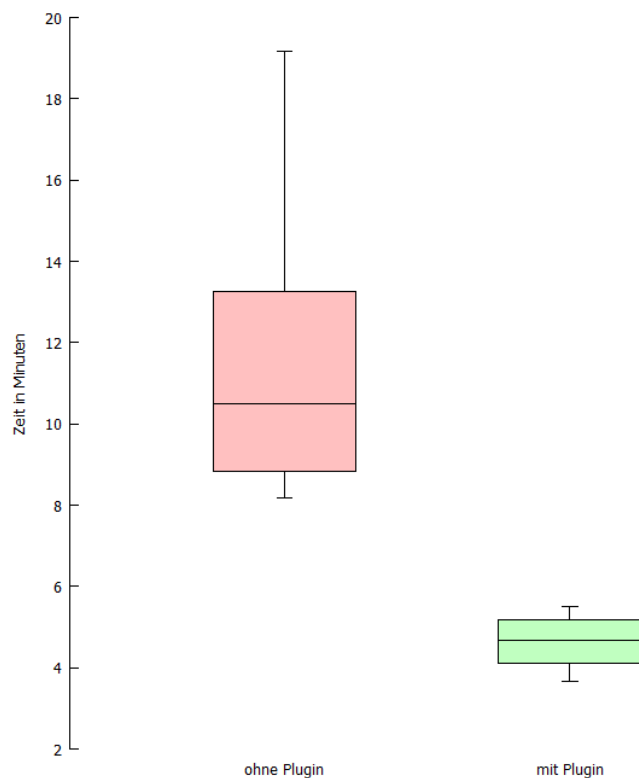


Abbildung 4.6: Zeigt einen Vergleich der Bearbeitungszeiten der dritten Aufgabe, in der es drum ging, das Wortproblem zu lösen. Rechts die Gruppe A, links die Gruppe B.

Diese Aufgabe zeigt starke Unterschiede in den Ergebnissen der Gruppen. Die Teilnehmer der Gruppe B hatten im Mittel einen knapp 2,5 mal höheren Zeitbedarf als die Teilnehmer der Gruppe A (siehe Abbildung 4.6). Gleichzeitig wurde von keinem der Probanden der Gruppe B - wie in der ersten Aufgabe - die Aufgabe mit null Fehlern bearbeitet. Im Schnitt hatte jeder Teilnehmer aus Gruppe B 25% der Fragen falsch beantwortet. Die Ergebnisse zeigen deutlich, dass auch in Hinsicht der Fehlerreduzierung das Plugin einen Vorteil geschaffen hat.

Abbildung 4.7 zeigt noch einmal alle Fehlerwerte der Teilnehmer die ohne das Plugin arbeiteten. Die Werte sind in Prozent angegeben, da in jeder Aufgabe unterschiedlich viele Fehler gemacht werden konnten. Ein Vergleich mit denen der Gruppe A ist nicht nötig, da alle Aufgaben von allen Teilnehmern mit null Fehlern abgeschlossen wurden.

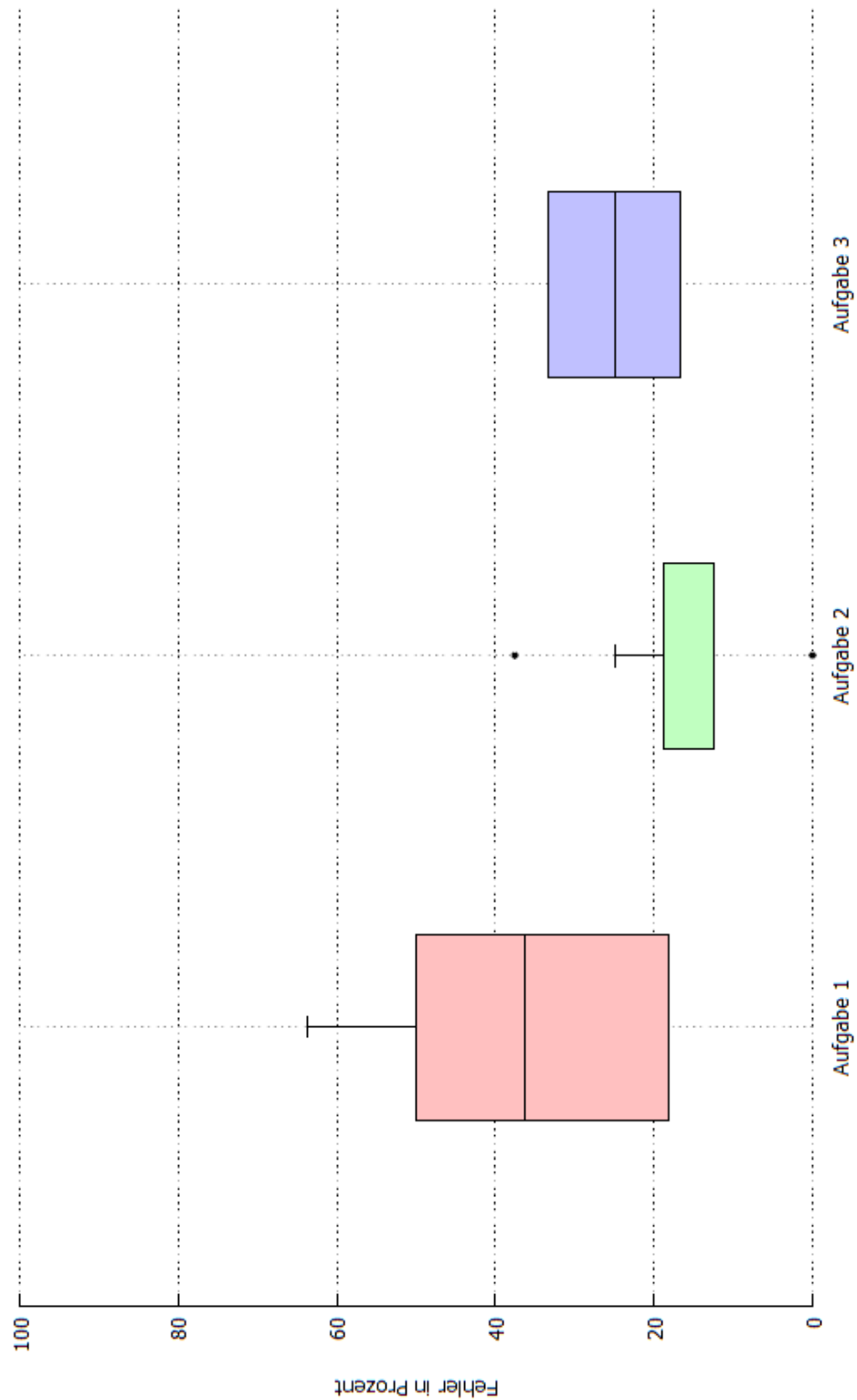


Abbildung 4.7: Zusehen sind die Fehlerwerte aller Aufgaben in Prozent. Die Werte beziehen sich auf die der Teilnehmer, die ohne das Plugin gearbeitet haben.

4.3.4 Funktionalitäten

Im Folgenden werden die Antworten des Fragebogens zu den implementierten Funktionalitäten ausgewertet und diskutiert. Dazu werden zuerst die Ergebnisse der Gruppe B (Teilnehmer ohne Plugin) vorgestellt und anschließend mit denen der Gruppe A (Teilnehmer mit Plugin) verglichen. Obwohl die Gruppe B die implementierten Funktionalitäten nicht zur Verfügung hatte, wurde ein Meinungsbild erstellt, ob ihnen der Funktionsumfang für die gegebenen Aufgaben praktisch erscheine. Die Reihenfolge, zuerst die Gruppe B und anschließend die Gruppe A zu betrachten, ist sinnvoll, da die Ergebnisse der Gruppe B als Vorab-Hypothese betrachtet werden können.

Fehlermarkierung Der Nutzen der Fehlermarkierung wurde von Gruppe B mit durchschnittlich fünf Punkten geschätzt. Das Ergebnis stimmt mit dem der Gruppe A mit 4,8 Punkten im Mittel fast überein. Auch die Varianzen beider Ergebnisse liegen mit null für Gruppe A und 0,4 für Gruppe B sehr nah beieinander; die Erwartungen wurden damit fast erfüllt, aber nicht vollständig. Obwohl die Abweichung nur minimal ist, ist es wegen der geringen Abweichung der Varianz sinnvoll sie zu interpretieren. Aus weiteren Anmerkungen der Probanden in den Fragebögen geht hervor, dass es Mängel in der Visualisierung der Fehler gab und deshalb keine fünf Punkte verteilt wurden. Welche dies waren, wurden nicht dem Fragebogen hinzugefügt.

Syntaxhighlighting Mit dem Syntaxhighlighting verhält es sich in der Bewertung gegensätzlich. Die Effizienz der Funktionalität wurde im Schnitt auf 4,6 Punkte geschätzt, bekam in der Anwendung aber fünf Punkte.

Content Assistant Auch der Content Assistant und seine Funktionalität wurden hinsichtlich der Nützlichkeit mit 4,1 zu 4,6 Punkten unterschätzt. Der Punktabzug, zur vollen Punktzahl, beziehungsweise die Kritik an dem implementierten Content Assistant bezog sich darauf, dass die Benutzung zu umständlich realisiert wurde. Bemängelt wurde auch, dass er bereits vorhandene Syntax zerstört, indem er „zu viel Text“ einfügt. Vermutlich liegt die Ursache hierfür darin, dass der implementierte Content Assistant mit dem des klassischen Java-Editors verglichen wurde.

Im Rahmen dieser Bachelorarbeit ist es jedoch aus Zeitgründen nicht möglich, einen vergleichbaren Content Assistant zu entwickeln.

Automatische Formatierer Die Nützlichkeit des Formatierers wurde ebenfalls unterschätzt. Sie wurde von Gruppe B mit durchschnittlich vier Punkten bewertet, bekam in der Anwendung von der Gruppe A jedoch 4,5 Punkte im Mittel. Obwohl die Teilnehmer keine weitere Kritik an dem Formatierer übten wurde seine Nützlichkeit dennoch nicht mit voller Punktzahl bewertet. Dies könnte daran gelegen haben, dass die gegebenen Grammatiken bereits formatiert waren. Zudem war die zu erstellende Grammatik in der zweiten Aufgabe nicht so komplex, dass eine stetige Formatierung nötig gewesen wäre: die Funktionalität blieb also überwiegend ungenutzt. Die Wichtigkeit des Formatierers wird vor allem bei großen und komplexen Grammatiken durch regelmäßige Anwendung erkennbar.

Hoverinformation und Hyperlinks Ebenso verhält es sich mit den folgenden zwei Funktionalitäten: die Hoverinformation und die Hyperlinks. Auch ihre Vorteile kommen erst in komplexen Grammatiken zum Tragen. Im Rahmen dieser Arbeit war ebenfalls es aus Zeitgründen nicht möglich die zwei Nützlichkeit Funktionalitäten zu validieren. Obwohl die Funktionalitäten nicht im vollen Umfang genutzt werden konnten, empfanden sie die Teilnehmer dennoch als hilfreich und bewerteten sie im Mittel mit 3,8 Punkten.

Visualisierung der Regeln Ein anderer Trend zeigte sich in der Visualisierung der Grammatik. Mit 3,5 Punkten wurde diese von Gruppe B als brauchbar bezeichnet. In Gruppe A fand sie hingegen in der Praxis wenig Verwendung und wurde dementsprechend mit nur 2,5 Punkten bewertet. Dies hat mehrere Gründe, zum einen war eine Visualisierung für die gestellten Aufgabentypen nicht unbedingt von Nöten. Zum anderen bietet die Visualisierung nicht jedem Nutzer eine Hilfe und ist eher als eine Alternative zur textuellen Betrachtung zu sehen. Das Ergebnis ist deshalb im unmittelbaren Zusammenhang mit der Auswahl der Probanden zu sehen.

Lösen des Wortproblems Die Bewertung der letzten Funktionalität, das automatisierte Lösen des Wortproblems, wich leicht vom erwarteten optimalen Ergebnis von fünf Punkten ab. Das automatisierte Lösen des Wortproblems und die Erstellung eines Syntaxbaumes bezogen sich direkt auf den Inhalt der dritten Aufgabe und wurde von Gruppe B mit 4,1 Punkten bewertet. Obwohl der Wert bereits weit oben liegt, sprechen die Bemerkungen der Probanden während des Lösens der Aufgabe, z.B. „*Muss ich das echt machen?*“ oder „*Das dauert ja ewig!*“ für eine bessere Einschätzung der Nützlichkeit dieser Funktionen und damit einer höheren Bewertung. Dieses Ziel wurde durch die praktische Anwendung teilweise erreicht. Mit durchschnittlich 4,4 Punkten der Teilnehmer, die das Plugin zur Verfügung hatten, nähert sich der Wert der vollen Punktzahl. Aus den Zusatzbemerkungen in den Fragebögen geht hervor, dass eine bessere Integration in Eclipse erwünscht ist. Wie diese aussehen könnte, wird im folgenden Kapitel 4.4 gezeigt.

Abbildung 4.8 und 4.9 fassen die aufgezählten Daten noch einmal in einem Diagramm zusammen. Abbildung 4.8 zeigt die Bewertung Funktionalitäten der Gruppe A, also diejenigen der Teilnehmer welche mit dem Plugin arbeiteten. Abbildung 4.9 zeigt die Schätzwerte der Gruppe B. Alle Bewertungen und Schätzungen sind über die Ergebnisse der jeweiligen Gruppe gemittelt.

Es lässt sich also zusammenfassen, dass die Funktionalitäten im Schnitt als sehr nützlich empfunden wurden und dem Anwender eine große Hilfe im Hinblick auf den Faktor Zeitersparnis bot. Zudem konnte die Fehlerrate deutlich reduziert werden. Des Weiteren, auch dies war ein Ziel der Studie, konnten Probleme aufgezeigt werden, die das Plugin nicht oder nur suboptimal unterstützten, und daher noch Potenzial für Verbesserungen bieten.

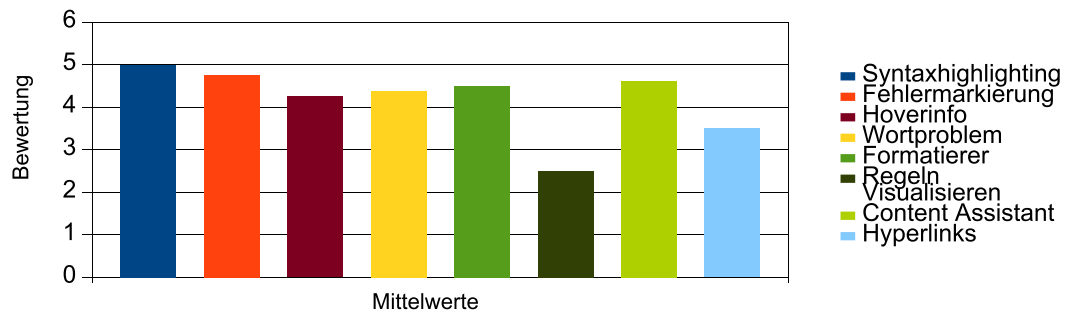


Abbildung 4.8: Zeigt die Bewertungen für jede Funktionalität der Gruppe A im Überblick.

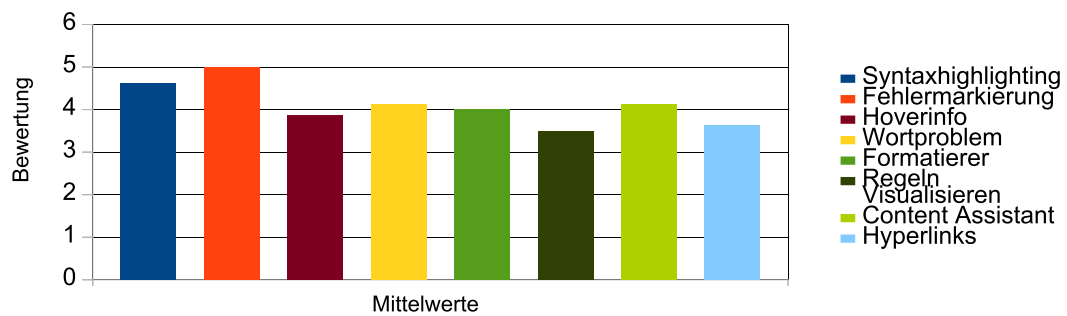


Abbildung 4.9: Zeigt die Schätzungen für jede Funktionalität der Gruppe B im Überblick.

4.4 Verbesserungsmöglichkeiten

Die Auswertung der Fragebögen, aber auch Vorüberlegungen, die während der Implementierung entstanden und nicht mehr berücksichtigt werden konnten, führten zu einer Liste von Verbesserungen und Erweiterungen, die das Plugin intuitiver und mächtiger gestalten sollen.

Es bestand Verbesserungsbedarf hinsichtlich der Fehlermarkierung. Es wurde angemerkt, dass die Fehlermeldungen zu ungenau seien. Eine mögliche Verbesserung wäre es, eine erweiterte Fehleranalyse anzubieten, die vor allem Anfängern einen großen Nutzen bringen würde. Ein weiterer Kritikpunkt bezog sich darauf, dass das Prüfen von Sätzen zu umständlich sei. Eine sinnvolle Konsequenz wäre, dass durch eine weitere Ansicht innerhalb der Eclipse Workbench ein Bereich geboten wird, um Sätze einzugeben, die simultan zum Erstellen der Grammatik geprüft werden. Dies hätte den Vorteil, dass die Sätze bei einem erneuten Start von Eclipse gespeichert blieben und nicht jedes Mal erneut eingegeben werden müssten. Durch das Anklicken eines Symbols könnten zu beliebigen Sätzen Syntaxbäume erstellt werden. Des Weiteren wurde angeregt, eine Übersicht mit allen Funktionalitäten und deren Tastenkombination zu erstellen. Dies wurde bereits umgesetzt und zum Plugin hinzugefügt. Ferner ist eine Verbesserung des Content Assistant vorgeschlagen worden, unter anderem mit Blick auf die Autovervollständigung und der automatischen Fehlerbehebung. Wie im Kapitel 3.1 Funktionalitäten beschrieben, gibt es zahlreiche und vielfältige Möglichkeiten, den Content Assistant intelligenter zu gestalten. Eine Erweiterung des Content Assistant in späteren Projekten ist also gut vorstellbar und sinnvoll.

5 Fazit und Ausblick

Im Rahmen dieser Bachelorarbeit ist auf Basis eines Eclipse-Plugin ein Editor zur optimierten Bearbeitung von ISR-Grammatiken erstellt worden.

Zunächst wurde durch das Erstellen von Anforderungen an die zu entwickelnde Software und eine anschließende Analyse der Anforderungen, eine Übersicht über die zu implementierenden Funktionalitäten erarbeitet. Ein detaillierter und realistischer Anwendungsfall unterstützte dabei die Auswahl und Entwicklung der Funktionalitäten. Aufgrund der konkreten Anforderungen war es sinnvoll, den Editor als Eclipse-Plugin zu entwickeln. Das ermöglichte, die zu implementierenden Funktionalitäten übersichtlich und erweiterbar einzubetten.

Die Auswahl und die anschließende Implementierung der Funktionalitäten stellen den Kern der Bachelorarbeit dar. nach der Implementierungsphase wurde eine Studie durchgeführt, in der belegt werden konnte, dass die gewählten Funktionalitäten sinnvoll sind und ihre Nutzung einen klaren Vorteil bei der Bearbeitung der ISR-Grammatik bietet. Aus der Studie ging ebenfalls hervor, dass an einigen Stellen des Plugin Verbesserungspotential gesehen wurde. Dazu wurden die Lösungsvorschläge der Probanden miteinbezogen und zum Teil nach der Studie dem Programm hinzugefügt. Dies lieferte den Beweis, dass die gewählte Architektur der Software es ermöglicht, Funktionalitäten nachträglich zu optimieren oder neue hinzuzufügen.

In weiteren Projekten könnte die Software dahingehend verbessert werden, selektive Funktionalitäten intelligenter oder effizienter zu programmieren. Es wäre zudem möglich, bestimmte Regeln, welche wiederkehrend in Grammatiken vorkommen, auszulagern. Dazu müsste ein eigenes Format entwickelt werden, welches solche Referenzen zulässt. Es würden redundante Informationen vermieden und gleichzeitig die Grammatiken konsistent gehalten. Eine zusätzliche Erweiterung

könnten weitere Metazeichen sein, die zum Beispiel zum Auskommentieren von Regeln definiert werden.

Zusammenfassend lässt sich sagen, dass der in dieser Bachelorarbeit entstandene Editor den Umgang mit der ISR-Grammatik sowohl für Anfänger als auch für Fortgeschrittene erleichtert. Darüber hinaus bietet die Erweiterbarkeit der Architektur die Möglichkeit, das Plugin zu optimieren oder um weitere Funktionalitäten zu ergänzen.

Anhang

Anwenderstudie zum ISR-Grammatik-Editor (Eclipse-Plugin)

Ziel

Versuchsperson: _____
Mit Plugin: ☐ja ☐nein

Aus meiner Bachelorarbeit ist eine Software herausgegangen, welche den Umgang mit der ISR-Grammatik vereinfachen soll. Die Software beschreibt ein Plugin welches in der Programmierumgebung Eclipse integriert wird. Das Ziel dieser Studie ist es heraus zu finden, wie viel Nutzen die integrierten Funktionalitäten des Editors mit sich bringen. Um zusätzliche Informationen zu gewinnen wird die Zeit gestoppt, die sie zum Lösen der Aufgabe benötigen. Alle Aufgaben beziehen sich auf die ISR-Grammatik. Der Aufbau ihrer Syntax gibt die folgende Syntax-Spezifikation her:

Syntax-Spezifikation:

GRAMMAR = { (IGN_LIST | ["[" RULE_NUM "]"] RULE) } .
IGN_LIST = "%IGNORE" "=" { TERMINAL } ";" .
RULE_NUM = NUMBER .
RULE = NONTERMINAL ":" BODY { "|" BODY } ";" .
BODY = { SYMBOL } .
SYMBOL = NONTERMINAL | TERMINAL .

NUMBER = *eine beliebige Integer-Zahl.*
z.B. 4

TERMINAL = *eine Sequenz von Buchstaben aus dem Set [a-zA-Z_\\"0-9].*
Wobei der Anfang einer Sequenz gegeben ist durch [a-zA-Z_\"]
z.B. _hello-World

NONTERMINAL = *ein TERMINAL mit dem Vorzeichen \$ oder \$\$.*
z.B. \$Greetings

Für Fragen bezüglich der Aufgabenstellung stehe ich ihnen jederzeit zur Verfügung.

Aufgaben

1. Aufgabe

1. *Korrigieren sie die gegebene **ISR-Grammatik**, sodass sie **keine Syntax-Fehler** mehr enthält. Nehmen sie dabei **keine Änderungen** an der **syntaktischen Struktur** der Grammatik vor.*

*Benutzen sie zum Lösen der Aufgabe die **Syntax-Spezifikation**.*

2. Aufgabe

1. ***Erstellen** sie eine **ISR-Grammatik** nach der oben gegebenen **Syntax-Spezifikation**, welche **keine Syntaxfehler** enthält und mindestens folgende **Sätze abbildet**.*

*Achten sie darauf, die **Grammatik** so aufzubauen, dass sie möglichst **leicht zu erweitern** ist. Des weiteren sollen **ähnliche Wortgruppen** in **namensgebenden Nonterminale** ausgelagert werden.*

Abzubildenden Sätze:

- Hi Tobì
- Hi Robot
- Go to the kitchen
- Tobì go to the kitchen
- Tobì go to the bath_room
- Robot go to the bath_room
- Leave the apartment
- Robot leave the room

3. Aufgabe

1. ***Prüfen** sie welche der folgenden **Sätze** von der in **Aufgabe 1** korrigierten **ISR-Grammatik abgebildet** werden können und welche nicht.*
2. ***Kreuzen** sie alle **positiv-getesteten Sätze** an.*
3. ***Zeichen** sie zu einem beliebigen positiv-getesteten Satz einen **Syntaxbaum**.*

Zu prüfenden Sätze:

- Bring some food from the sofa
- Bring it from the kitchen_table to the side_bar
- Go to the kitchen_table get the glass and bring it to the sofa
- Move to the kitchen get some food and bring it to the kitchen_table
- Get the pringles and put it on the side_table
- Drive to the couch and get the cream

Enthalten

☐
☐
☐
☐
☐
☐

Syntaxbaum:

Fragen für die Teilnehmer mit Plugin:

1. Wie schätzen sie ihre Erfahrung im Umgang mit dem Computer ein?

Viel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Keine
------	--------------------------	--------------------------	--------------------------	--------------------------	-------

2. Wie gut können sie mit der Programmierumgebung Eclipse umgehen?

Gut	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Gar nicht
-----	--------------------------	--------------------------	--------------------------	--------------------------	-----------

3. Wie viel Erfahrung haben sie im Umgang mit kontextfreien Grammatiken?

Sehr Viel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Keine
-----------	--------------------------	--------------------------	--------------------------	--------------------------	-------

4. Wie viel Erfahrung haben sie im Umgang mit der ISR-Grammatik?

Viel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Keine
------	--------------------------	--------------------------	--------------------------	--------------------------	-------

5. Wie gut haben sie die Aufgabenstellung verstanden?

Sehr gut	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Gar nicht
----------	--------------------------	--------------------------	--------------------------	--------------------------	-----------

6. Wenn sie bereits Erfahrungen mit der Isr-Grammatik haben:
Wie schätzen sie die Aufgabenstellungen im Bezug zur Realität ein?

Realistisch	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Keinen Bezug
-------------	--------------------------	--------------------------	--------------------------	--------------------------	--------------

7. Wie einfach fiel es ihnen die Aufgaben zu lösen?

Aufgabe 1:	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Schwer
Aufgabe 2:	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Aufgabe 3:	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

8. Wie viel haben die einzelnen Funktionalitäten zum schnellen Lösen der Aufgabe beigetragen? **Bei Fragen bitte fragen!**

1. Syntax-Hervorhebung	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nützlich	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Ungebraucht
2. Fehler-Markierung	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>						
3. Automatisierte Syntaxbäume	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>						
4. Automatisierte Formatierung	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>						
5. Hover-Information	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>						
6. Hyperlinks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>						
7. Visualisierung	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>						
8. Content-Assistent	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>						

9. Wie nützlich erscheint Ihnen dieser Editor für die praktische Anwendung?

Sehr	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Gar nicht
------	--------------------------	--------------------------	--------------------------	--------------------------	-----------

10. Ich fände zusätzlich folgende Funktionalität sinnvoll:

11. Folgende Funktionalitäten waren meiner Meinung nach zu unständiglich realisiert:

12. Weitere Kommentare und Anmerkungen:

Zur Person:

Alter: ☐ < 25 ☐ 25-50 ☐ > 50

Geschlecht: ☐ m ☐ w

Studiengang/Tätigkeit: _____

Fragen für die Teilnehmer ohne Plugin:

1. Wie schätzen sie ihre Erfahrung im Umgang mit dem Computer ein?

Viel ☐ ☐ ☐ ☐ Keine ☐

2. Wie gut können sie mit dem benutzen Texteditor umgehen?

Gut ☐ ☐ ☐ ☐ Gar nicht ☐

3. Wie viel Erfahrung haben sie im Umgang mit kontextfreien Grammatiken?

Sehr Viel ☐ ☐ ☐ ☐ Keine ☐

4. Wie viel Erfahrung haben sie im Umgang mit der ISR-Grammatik?

Viel ☐ ☐ ☐ ☐ Keine ☐

5. Wie gut haben sie die Aufgabenstellung verstanden?

Sehr gut ☐ ☐ ☐ ☐ Gar nicht ☐

6. Wenn sie bereits Erfahrungen mit der Isr-Grammatik haben:
Wie schätzen sie die Aufgabenstellungen im Bezug zur Realität ein?

Realistisch ☐ ☐ ☐ ☐ Keinen Bezug ☐

7. Wie einfach fiel es ihnen die Aufgaben zu lösen?

Aufgabe 1: ☐ ☐ ☐ ☐ Schwer ☐
Aufgabe 2: ☐ ☐ ☐ ☐
Aufgabe 3: ☐ ☐ ☐ ☐

8. Wie viel Nutzen zum lösen der Aufgaben würden sie sich von den gegebenen Funktionalitäten versprechen? **Bei Fragen bitte fragen!**

	Nützlich	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Ungebraucht
1. Syntax-Hervorhebung	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2. Fehler-Markierung	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3. Automatisierte Syntaxbäume	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4. Automatisierte Formatierung	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5. Hover-Information	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
6. Hyperlinks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
7. Visualisierung	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
8. Content-Assistent	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

9. Wie nützlich erscheint Ihnen ein Editor für die praktische Anwendung?

Sehr ☐ ☐ ☐ ☐ Gar nicht ☐

10. Ich fände zusätzlich folgende Funktionalität sinnvoll:

11. Weitere Kommentare und Anmerkungen:

Zur Person:

Alter: ☐ < 25 ☐ 25-50 ☐ > 50
Geschlecht: ☐ m ☐ w

Studiengang/Tätigkeit: _____

Abbildungsverzeichnis

2.1	In dieser Abbildung ist ein Syntaxbaum dargestellt, indem die Herleitung des Satzes „ <i>Today at one oclock</i> “ zu sehen ist.	7
3.1	Syntaxhighlighting einer standardformatierten Grammatik im ISR-Grammatik-Editor.	16
3.2	Zwei verschiedene Arten von Fehlermarkierungen. Die gelben Markierungen stellen Warnungen dar, rote zeigen fatale Fehler an. Der Fehlertext wird als Hoverinformation dargestellt.	18
3.3	Content Assistant bei einem Aufruf zwischen zwei Regeln. Die fehlende Deklaration des Non-Terminals GERMAN als erste Auswahlmöglichkeit.	19
3.4	Visualisierung zweier Regeln einer Grammatik. Links im Bild ist die Selektion der Regeln, rechts die Darstellung.	21
3.5	Die Abbildung zeigt das automatische Lösen des Wortproblems. Oben die Eingabe, unten das Ergebnis, nach drücken des OK-Button	22
3.6	Die Abbildung zeigt die Plugin-Architektur. Links das neu erstellte Plugin, rechts die bereits vorhandene Plugins.	24
3.7	Die Editor-Architektur	27
3.8	Diese Abbildung zeigt die Anbindung einer Funktionalität der Klasse ExampleHandler über Extension Points.	30
3.9	Klassendiagramm des Hyperlinksystems	32
3.10	Diese Abbildung zeigt das Klassendiagramm des Systems der automatischen Formatierung.	36
4.1	Zeigt einen Vergleich der Bewertungen wie einfach die erste Aufgabe gelöst werden konnte. Rechts die Gruppe A, links die Gruppe B. . .	45

4.2	Zeigt einen Vergleich der Bearbeitungszeiten der ersten Aufgabe, in der es darum ging, eine fehlerhafte Grammatik zu korrigieren. Rechts die Gruppe A, links die Gruppe B.	46
4.3	Zeigt einen Vergleich der Bewertungen wie einfach die zweiten Aufgabe gelöst werden konnte. Rechts die Gruppe A, links die Gruppe B.	47
4.4	Zeigt einen Vergleich der Bearbeitungszeiten der zweiten Aufgabe, in der es darum ging, eine Grammatik für bestimmte Sätze zu bilden. Rechts die Gruppe A, links die Gruppe B.	48
4.5	Zeigt einen Vergleich der Bewertungen wie einfach die dritten Aufgabe gelöst werden konnte. Rechts die Gruppe A, links die Gruppe B.	49
4.6	Zeigt einen Vergleich der Bearbeitungszeiten der dritten Aufgabe, in der es drum ging, das Wortproblem zu lösen. Rechts die Gruppe A, links die Gruppe B.	50
4.7	Zusehen sind die Fehlerwerte aller Aufgaben in Prozent. Die Werte beziehen sich auf die der Teilnehmer, die ohne das Plugin gearbeitet haben.	52
4.8	Zeigt die Bewertungen für jede Funktionalität der Gruppe A im Überblick.	56
4.9	Zeigt die Schätzungen für jede Funktionalität der Gruppe B im Überblick.	56

Literatur

- [1] G. A. Fink, “Developing hmm-based recognizers with esmeralda,” in *Proceedings of the Second International Workshop on Text, Speech and Dialogue*, TSD '99, (London, UK, UK), pp. 229–234, Springer-Verlag, 1999.
- [2] O. O. K. K. C. IBM Software Group, 2670 Queensview Drive, “Eclipse: A platform for integrating development tools,” *IBM Systems Journal*, vol. 43, pp. 371 – 383, 2004.
- [3] D. P. Volker Wohlgemuth, Tobias Schnackenberg and R.-L. Barling, *Development of an Open Source Software Framework as a Basis for Implementing Plugin-Based Environmental Management Information Systems (EMIS)*. No. ISBN: 978-3-8322-7313-2, Shaker Verlag, 2008.
- [4] H. S. Manfred Henning, “Einführung in den extension point mechanismus von eclipse,” *JavaSPEKTRUM*, vol. 01, 2008.
- [5] I. Wegener, *Theoretische Informatik - eine algorithmenorientierte Einführung*. Teubner, 2008.
- [6] J. Earley, “An efficient context-free parsing algorithm,” *Commun. ACM*, vol. 13, pp. 94–102, Feb. 1970.
- [7] C. Burch and L. Ziegler, “Science of computing suite(socs) :resources for a breadth-first introduction,” *Technical Symposium on Computer Science Education (SIGCSE)*, 2004.

Versicherung

Versicherung gemäß §10a, Absatz 5 der Studien- und Prüfungsordnung für das Bachelorstudium (BPO) an der Technischen Fakultät der Universität Bielefeld vom 31. März 2009.

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

Eclipse-Plugin zur optimierten Erstellung und Wartung von Grammatiken für das Spracherkennungssystem ESMERALDA

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Bielefeld, den 5. November 2012

HENDRIK TER HORST