Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Masterarbeit Informatik

Ein Eclipse-Plugin für Scheme 48

Sebastian Rheinnecker

10. Januar 2012

Gutachter

Prof. Dr. Herbert Klaeren
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Betreuer

Dipl.-Inform. Marcus Crestani
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Sand 13, 72076 Tübingen
Eberhard Karls Universität Tübingen

Rheinnecker, Sebastian:
Ein Eclipse-Plugin für Scheme 48
Masterarbeit Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 18. April 2011 - 10. Januar 2012

Zusammenfassung

Diese Masterarbeit beschreibt die Entwicklung und Implementierung einer modernen Entwicklungsumgebung für die Scheme-Implementierung Scheme 48 in Eclipse. Die *Scheme 48 Development Tools* (SDT) weisen neben den bekannten Funktionen einer modernen Entwicklungsumgebung, wie ein Editor mit Syntaxhervorhebung und einer Outline, auch viele speziell auf Scheme 48 ausgelegte Funktionen, wie syntaktische und semantische Fehlerannotationen, Formatierungshilfen, intelligente Suche, Code Templates, Content Assist und Wizards auf. Außerdem bietet die Entwicklungsumgebung eine Anbindung an den Scheme-48-Interpreter zur Verbesserung der semantischen Unterstützung und zur einfachen Ausführung von Programmen.

Danksagung

Als Erstes möchte ich Prof. Dr. Herbert Klaeren für die ausgereifte und wissenschaftlich fundierte Ausbildung im Studium danken und für die Ermöglichung dieser Masterarbeit. Der Studiengang Informatik profitiert enorm von Professoren, die sich um eine gute Ausbildung der angehenden Informatiker kümmern.

Als nächstes danke ich meinem Betreuer, Marcus Crestani, der mir bei der Erstellung der Masterarbeit immer hilfreich und kritisch zur Seite stand.

Auch danke ich Dominique Boucher und Matthias Köster, den Entwicklern von SchemeWay respektive Clojure IDE, für die Bereitstellung von Quelltexten und für die Hilfe mit Eclipse und DLTK, wenn ich feststeckte.

Dann möchte ich mich bei meinen Freunden bedanken, ohne die mein Studium mit diesem Erfolg wohl nicht möglich gewesen wäre und dafür, dass mein Studium durch sie zur Freude wurde.

Besonderen Dank geht natürlich auch an meine Familie, die mir das alles erst ermöglich hat und mir in allen Situationen des Lebens zur Seite steht. Ganz besonders danke ich meiner Schwester Laura für ihre gestalterische Hilfe bei der Benutzeroberfläche von SDT.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

IDE	Integrated Development Environment (integrierte Entwicklungsumgebung)
JDT	Java Development Tools
CDT	C/C++ Development Tooling
PDE	Plug-in Development Environment
SDT	Scheme 48 Development Tools
DLTK	Dynamic Languages Toolkit
AST	Abstract Syntax Tree (abstrakter Syntaxbaum)
ANTLR	ANother Tool for Language Recognition
CL	Configuration Language
UI	User Interface (Benutzeroberfläche)
DSL	Domain-specific Language (domänenspezifische Sprache)

Kapitel 1

Einleitung

1.1 Motivation

Nahezu alle Softwareprojekte werden heutzutage in spezialisierten Entwicklungsumgebungen (*Integrated Development Environment*, kurz IDE) angefertigt, die dem Programmierer oder dem Team von Programmierern möglichst viel Arbeit abnehmen, indem sie Informationen über das zu schreibende Programm darstellen. Das ermöglicht es dem Programmierer, sich auf den Entwurf der Software zu konzentrieren und nicht durch idiosynkratische Elemente der Sprache behindert zu werden. Moderne Entwicklungsumgebungen sind dabei gerade für größere Softwareprojekte eine enorme Hilfe, da sie den mit der Größe eines Projekts proportional steigenden Grad an Komplexität einer Software verkleinern und somit selbst komplexe und große Zusammenhänge dem Programmierer übersichtlich darstellen. Daher gibt es für fast jede Sprache eine eigene Entwicklungsumgebung, welche die Semantik der Programme versteht und für den Programmierer sowohl im Hintergrund als auch sichtbar mitdenkt, beispielsweise in Form von Syntaxhervohebungen, Programmumrissen, Fehlerannotationen und kontextualer Hilfe.

Aktuell gibt es eine große Auswahl verschiedener kommerzieller sowie frei verfügbarer Entwicklungsumgebungen. Zwei Magisterarbeiten an der Technischen Universität Wien [28][24] vergleichen einige davon in Hinblick auf die Sprachen Java und C++/C#. Demnach gibt es generell nur wenige Vorteile von kommerziellen IDEs gegenüber nicht-kommerziellen, und die meisten frei verfügbaren IDEs bieten selbst für professionelle Entwickler hinreichende Unterstützung.

Aus der Masse an verfügbaren Entwicklungsumgebungen sticht besonders Eclipse [4] hervor. Es bietet eine Plattform, die sich besonders durch ihre Erweiterbarkeit in Form von Plugins auszeichnet. Es versteht sich als programmiersprachenagnostisches Framework zur Konstruktion von IDEs und besitzt mit offiziellen JDT-Plugin eine der besten und populärsten Java-

Entwicklungsumgebungen überhaupt. Mit Eclipse ist die Entwicklung einer IDE stark vereinfacht, da häufig gebrauchte Komponenten durch das Framework bereits angeboten werden. Die Entwicklungszeit einer eigenständigen Entwicklungsumgebung ist daher gegenüber eines Eclipse-Plugins um ein vielfaches länger. Aus diesem Grund gibt es mittlerweile über eine Million Plugins, welche die Funktionalität von Eclipse erweitern und es auf die individuellen Bedürfnisse eines Entwicklers anpassen.

Verglichen mit dem Alter der Sprache gibt es nur relativ wenige moderne Entwicklungsumgebungen für Scheme und dessen Dialekte. Zu den bekanntesten Entwicklungsumgebungen von Scheme zählen DrRacket und GNU Emacs. Während DrRacket speziell auf die Bedürfnisse der Anfängerausbildung zugeschnitten und daher für professionelle Programmierer eher uninteressant ist, stellt GNU Emacs eine IDE dar, die aufgrund des schwierigen Einstiegs und im Hinblick auf andere moderne Entwicklungsumgebungen ungewöhnlich tastatur- und textlastigen Aufbau von vielen Programmierern gemieden wird.

Scheme erlangt mitunter mangels populärer und ansprechender IDEs kein großes Publikum. Hinzu kommt, dass es eine Fülle an verschiedenene Implementierungen der Sprache gibt, von denen sich manche nur sehr geringfügig unterscheiden, was für Scheme-Anfänger abschreckend sein kann und bei der Auswahl einer geeigneten Implementierung hinderlich ist. Scheme 48 ist eine Scheme-Implementierung, die sich vor allem wegen seines Modulsystems und Scheme-Debuggers auszeichnet. Trotzdem gibt es mit GNU Emacs nur eine Entwicklungsumgebungen für Scheme 48. In GNU Emacs exitiert lediglich ein Modus, der grundlegende Befehle verfügbar macht, um mit Scheme 48 arbeiten zu können. Moderne Entwicklungsumgebungen mit fortgeschrittener Funktionalität wie Syntaxhervorhebung, Outline, Fehlerannotationen, intelligenter Suche, Content Assist und Wizards fehlen. Diese Masterarbeit behandelt die Entwicklung und Implementierung einer auf Eclipse basierenden Entwicklungsumgebung für Scheme 48.

1.2 Aufbau der Arbeit

In dem ersten Teil dieser Masterarbeit entwickle ich zunächst ein allgemeines Konzept einer modernen Entwicklungsumgebung und erstelle einen Katalog von Komponenten, die in IDEs typischerweise enthalten sind und beschreibe, welche Vorteile diese dem Programmierer bieten.

Im zweiten Kapitel erläutere ich die Eigenschaften von Scheme und Scheme 48 und stelle vor, wie sich diese in einer IDE wiederspiegeln sollten.

Anschließend beschreibe ich den Aufbau von Eclipse und erkläre, wie Funktionalität in Form von Plugins hinzugefügt werden kann. Darauf aufbauend beschreibe ich die benötigten und typischen Komponenten eines Plugins für

Eclipse.

Im Hauptteil der Arbeit stelle ich das von mir entwickelte Eclipse-IDE-Plugin *Scheme 48 Development Tools* (SDT) vor und beschreibe dessen Entwurf und Implementierung.

Abschließend fasse ich das Erreichte zusammen und gebe eine Bewertung im Hinblick auf die gestellten Ziele. Dabei gebe ich auch einen Ausblick und beschreibe, welche Verbesserungsmöglichkeiten es gibt. Des Weiteren schlage ich verschiedene Wege vor, diese Verbesserungen umzusetzen.

Kapitel 2

Skizzierung einer modernen IDE

Moderne Entwicklungsumgebungen, sog. *Integrated Development Environments* (IDE), sind aus heutiger Sicht aus dem Prozess der Softwareentwicklung nicht mehr wegzudenken. Durch eine Vereinfachung des Entwicklungsprozesses und der Verfügbarkeit immer besserer Werkzeuge, die den Entwickler sowohl visuell als auch logisch unterstützen, sinkt theoretisch die Lernkurve für eine Sprache und die erforderliche Entwicklungszeit für alle Arten von Softwareprojekten.

2.1 Geschichtlicher Umriss

Mit der Entwicklung des Computers in der Mitte des 20. Jahrhunderts kam die Notwendigkeit eines Mediums für die Programmeingabe und Datenspeicherung auf. Damals boten sich Lochkarten an, die schon seit dem 18. Jahrhundert etabliert waren. Die Codierung von Programmcode erfolgte über die Position von Löchern auf einer Karte, die von Lochkartenlesern abgetastet werden konnten. Erfahrene Programmierer konnten zwar allein durch die Löcher den codierten Code interpretieren und in späteren Versionen waren auf den Karten ebenfalls die Klartexte abgedruckt, jedoch gestaltete sich der Entwicklungsprozess mehr als schwierig. Um ein Programm zu testen, musste der Code auf der Lochkarte gestanzt und dann vom Lochkartenleser wieder decodiert werden. Selbst kleinste Fehler auf den Lochkarten führten zu erheblichen Zeitverlusten beim Programmieren, zumal die Zeit für die meist wenig vorhandenen Lochkartenleser begrenzt war.
In den 70er Jahren wurden die Lochkarten durch Magnetbänder und Magnetplatten ersetzt. Diese ermöglichten eine einfachere Speicherung von Daten und wurden zusammen mit neuartigen Bildschirmgeräten dafür benutzt, eine digitale Textverarbeitung zu ermöglichen. Diese war jedoch äußerste träge und wenig intuitiv, da die Reaktion auf eine Eingabe erst nach dem

Ausfüllen einer Seite erschien. Mit Maestri 1 (ursprünglich *Programm-Entwicklungs-Terminal-System* PET) wurde 1975 von der Münchner Firma Softlab die weltweit erste IDE vorgestellt[1]. Diese löste das Problem der früheren Textverarbeitungsprogramme und machte Eingaben sofort sichtbar. In den darauffolgenden Jahren wurden immer mehr IDEs für Sprachen wie Turbo Pascal [13] und später Natural [23] veröffentlicht. Diese enthielten schon fortschrittliche Komponenten moderner IDEs wie Text- und Maskeneditor, Compiler/Interpreter, Debugger, Quelltextformatierungsfunktionen und GUI Designer.
Die größten und bekanntesten kommerziellen Hersteller von integrierten Entwicklungsumgebungen sind seit Mitte der 90er Jahre vor allem im Bereich C/C++ und C# Borland mit dem Borland Developer Studio (BDS) [1] und Microsoft mit Visual Studio [8]. Jedoch konnte Microsoft in den letzten Jahren immer mehr die Pendants von Borland verdrängen, wodurch Borland im Februar 2006[2] bekanntgab, die IDE-Sparte zu verkaufen und sich auf den Bereich des *Application Lifecycle Management* (ALM) zu konzentrieren. Das Borland Developer Studio, das seit 2008 der Firma Embarcadero Technologies gehört[3], vereinigte in der letzten Version (2006) C/C++, C# und Delphi, die zuvor getrennt vermarktet wurden. Microsofts Visual Studio ist eine Entwicklungsumgebung für das .NET Framework und unterstützt als solche Visual Basic .NET, C/C++, C++/CLI, C# und F#. Kurzzeitig wurde auch J# unterstützt (von 2002 bis 2008), das Java-Entwicklern den Umstieg auf das .NET Framework erleichtern sollte. Neben den kommerziellen IDEs gibt es eine Vielzahl an Open-Source-Projekten; die bekanntesten davon sind im Bereich C/C++ und C# Sharp Develop[4] und Eclipse mit CDT Plugin [2] und im Bereich Java wiederum Eclipse mit JDT Plugin und NetBeans [6].

2.2 Aspekte einer modernen IDE

Die Hauptaufgabe einer modernen IDE ist die Vereinfachung des Entwicklungsprozesses für den Programmierer. Aus der Erfahrung, die man während der letzten Jahrzehnte in der Softwareentwicklung gesammelt hat, lassen sich einige Aspekte zusammenfassen, die eine effektive Unterstützung für den Programmierer ermöglichen:

- Das Feedback für den geschriebenen Code muss möglichst schnell, am besten sofort, in einer möglichst akkuraten Form dem Entwickler präsen-

[1] http://www.computerwoche.de/heftarchiv/1975/47/1205421/
[2] http://www.eweek.com/c/a/Application-Development/Borland-to-Divest-Dev-Tools-with-Segue-Buyout/
[3] http://www.silicontap.com/embarcadero_technologies_buys_codegear/s-0015183.html
[4] http://www.icsharpcode.net/OpenSource/SD/

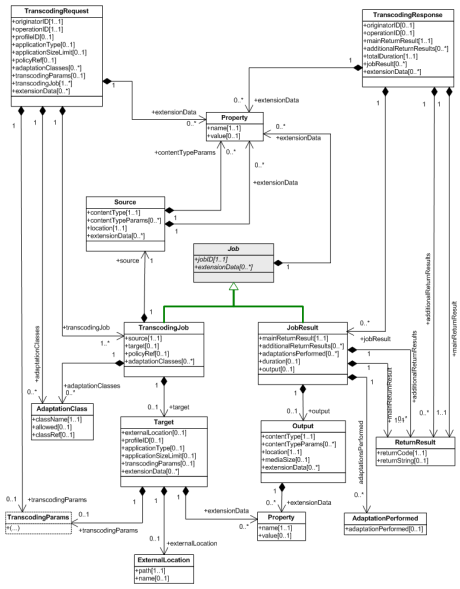


Abbildung 2.1: UML Diagramm eines kleinen, objektorientierten Programms. Beim Editieren einer Klasse ist diese Information nicht sichtbar und zu komplex, um erinnert zu werden.

tiert werden. Vor allen Dingen Fehler, die der Compiler erkennt, sollten ohne den Kompilierprozess angezeigt werden, um Zeit zu sparen. Solche Fehler sind meist anhand der Sprachdefinition leicht zu erkennen.

- Der Editor soll die logische Struktur eines Programms verdeutlichen. Software ist in ihrer Natur ein aus zusammenhängenden Teilen bestehendes System, was mitunter auch ein Grund ist, warum Informatik heutzutage zusammen mit der Mathematik als eine *Strukturwissenschaft* bezeichnet wird. Mit der Größe und Komplexität eines Programms steigt auch die Schwierigkeit für den Entwickler, die Struktur als Ganzes zu erfassen, weshalb eine IDE die innere Struktur eines Programms auf verschiedene abstrakte Weisen präsentieren sollte. Abbildung 2.1 zeigt eine Art dieser abstrakten Darstellung in Form eines UML-Diagramms. Diese versuchen in objektorientierten Sprachen die Abhängigkeiten der Klassen untereinander zu symbolisieren. Allerdings sind diese bereits für kleine Programme sehr groß und können so nur schwer erinnert werden.

- Die Semantik eines Programms sollte von der IDE erfassbar sein, um dem Entwickler Vorschläge und Hinweise geben zu können. Aufgrund der bereits erwähnten logischen Struktur eines Programms gibt es an verschiedenen Stellen auch verschiedene Kontexte. Wenn der Programmierer eine Stelle im Programm bearbeitet, befindet er sich in einem bestimmten Kontext, in dem für die Semantik verschiedene Regeln gelten

(bspw. Variablensichtbarkeit). Das Erkennen und Merken dieses Kontexts sind Aufgaben, die eine IDE dem Entwickler abnehmen und bei Bedarf anzeigen kann.

- Da zum Erstellen von Software verschiedene Teile notwendig sind (Editor, Compiler, Debugger etc.) und diese auch unter Umständen aufeinander abgestimmt werden müssen, ist es von Vorteil, wenn eine IDE diese Teile bereits integriert hat. Dadurch kann Planungs- und Einarbeitungszeit gespart werden.

Zur Umsetzung dieser Aspekte haben sich im Laufe der Jahre diverse Komponenten durchgesetzt. Viele IDEs unterscheiden sich nicht mehr darin, welche davon implementiert sind, sondern darin, wie intelligent diese umgesetzt sind. Im Folgenden werden die wichtigsten Komponenten erläutert.

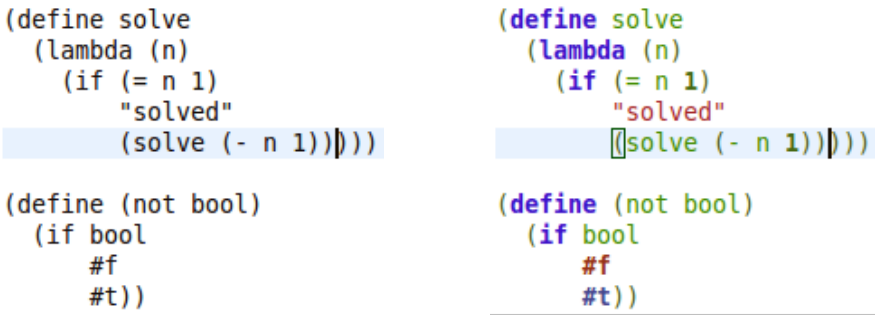
2.3 Konkrete Komponenten moderner IDEs

2.3.1 Dateiverwaltung

Nur wenige Programme bestehen aus einer oder wenigen Dateien. Die Auslagerung von Funktionalität in verschiedene Dateien trennt logisch unabhängige Programmabschnitte und fördert einen modularen Aufbau. Doch mit steigender Komplexität und gerade bei objektorientierten Sprachen nimmt die Zahl der benötigten Dateien stetig zu, weshalb eine hierarchische Ordnerstruktur verwendet wird, um die Dateien zu ordnen. Je nach der Tiefe ist dann das Wiederfinden oder Zuordnen von Funktionalität schwierig. Bei objektorientierten Sprachen wie Java oder C# wird meist für jede Klasse eine eigene Datei eingeführt, wodurch selbst bei kleineren Programmen die Anzahl der Dateien schnell ansteigt (jedenfalls wenn das objektorientierte Paradigma richtig umgesetzt wird). Dadurch ist eine passable Verwaltung und Darstellung der hierarchischen Ordner- und Dateistruktur notwendig, die in der IDE integriert ist.

2.3.2 Syntaxhervorhebung

Die Syntaxhervorhebung (engl. *Syntax Highlighting*) ist eine der grundlegensten Funktionen einer IDE. Dabei werden verschiedene Sprachelemente im Text durch unterschiedliche Farben, Schriftarten und -stilen verbessert markiert und differenziert. Dies steigert zunächst erheblich die Lesbarkeit des Textes, da die Struktur eines Quellcodes so intuitiv sichtbar wird. Kommentare werden sofort als solche erkannt und vom bedeutungsvollen Quellcode getrennt, Schlüsselwörter grenzen sich von anderen Symbolen ab, sprachliche Begrenzungssymbole werden hervorgehoben und man kann Datentypen unterscheiden



(a) Ohne Syntaxhervorhebung

(b) Mit Syntaxhervorhebung

Abbildung 2.2: Darstellung eines einfachen Programms ohne und mit Syntaxhervorhebung. Die Struktur ist im linken Bild nur duch die Formatierung zu erkennen, im rechten Bild jedoch offensichtlich.

(bspw. Strings, Character und Zahlen). Neben der erhöhten (Quer-)Lesbarkeit des Textes werden auch beim Schreiben Fehler schneller erkannt, da diese zu einer veränderten Darstellung führen (bspw. bei Tippfehlern).
In Abbildung 2.2(a) erkennt man ein Scheme-Programm ohne Syntaxhervorhebung. Die Struktur des Programms ist intuitiv nur durch die Formatierung erkennbar. Abbildung 2.2(b) zeigt hingegen das gleiche Programm mit Syntaxhervorhebung: Schlüsselwörter sind fett gedruckt und heben sich sofort von anderen Symbolen ab, ebenso wie Zeichenketten und Zahlen. Zu einer schließenden Klammer wird außerdem die dazugehörige öffnende Klammer angezeigt.
Syntaxhervorhebung ist ein fester Bestandteil vieler, wenn nicht aller IDEs und sogar vieler moderner Textverarbeitungsprogramme wie Notepad++[5] oder gedit[6].

2.3.3 Outline

Geschieht die Auslagerung von Funktionalität nicht über verschiedene Dateien oder wird ein logisch zusammenhängender Quelltext sehr lang, kann der Überblick innerhalb einer Datei verloren gehen. Besonders das Auffinden von Definitionen oder Deklarationen kann sich als mühselige Arbeit erweisen. Eine Outline (dt. Umriss, sinngemäß auch Gliederung) bietet eine abstrahierte Darstellung über den Inhalt und skizziert einzelne Strukturen innerhalb einer Datei wie etwa Variablen- oder Klassendeklarationen (vgl. Abbildung 2.3). Die Darstellung mittels einer Baumstruktur entspricht meist der hierarchischen Struktur des Programms und man kann über die einzelnen Knoten direkt zur

[5] http://notepad-plus-plus.org/
[6] http://projects.gnome.org/gedit/

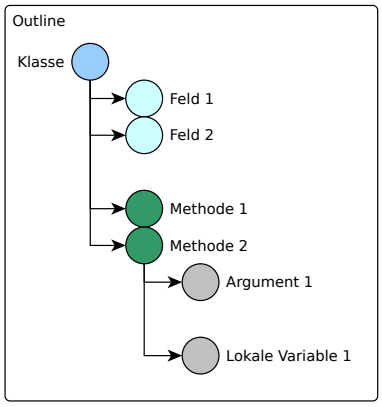


Abbildung 2.3: Schema einer Outline für eine Klasse mit zwei Feldern und Methoden. Die hierarchische Darstellungsweise der Klasse vermittelt den Aufbau.

Deklaration springen. Dies erlaubt eine schnelle Navigation innerhalb der Datei und verschafft eine Gesamtübersicht über die bereits geschriebenen Zeilen.

2.3.4 Fehleranzeige

Eine der wichtigsten Funktionen einer IDE ist die Anzeige von Fehlern im Quellcode noch vor der Übersetzung durch einen Compiler. Man unterscheidet hier zwischen *syntakischen* Fehlern, die durch eine falsche Struktur des Programms entstehen, d. h. der Sprachdefinitionen widersprechen, und *semantischen* Fehlern, die einem logischen Fehler im Programm entsprechen, etwa einer illegalen Zuweisungsoperation oder einem Prozeduraufruf mit der falschen Anzahl an Argumenten. Beide Fehlerarten können prinzipiell von einer IDE erkannt werden, bevor das Programm ausgeführt wird. Bei syntaktischen Fehlern ist die Fehlererkennung einfach, da meist der verwendete Parser die Ausdrücke einfach nicht erkennt und dabei eine Fehlermeldung generiert. Semantische Fehler hingegen sind wiederum aufzuteilen in Fehler, die zur Übersetzungszeit bereits bekannt sind und solche, die erst zur Laufzeit bekannt werden. Fehler der ersten Art können von IDEs mit etwas Aufwand zuverlässig ermittelt werden. Meist wird dazu eine Art interner Interpreter benutzt (beispielsweise ein Algorithmus, der den abstrakten Syntaxbaum des Programms traversiert). Solche Fehler sind etwa nicht-deklarierte Variablen, nicht-initialisierte Felder oder Typfehler bei Sprachen mit einer statischen Typisierung. Laufzeitfehler hingegen sind nur schwer von der IDE erkennbar, da solche Fehler meist von der Auswertungsreihenfolge abhängen, die entweder nicht bekannt oder nur schwer nachvollziehbar ist. Solche Fehler sind zum Beispiel Referenzen auf nicht vorhandene Objekte. Eine moderne IDE markiert die Fehler meist farblich in dem Editor und weist so den Programmierer unmittelbar darauf hin.

2.3.5 Formatierungshilfen

Zu jeder Programmiersprache existieren auch Konventionen über die Formatierung des Quellcodes, etwa der Einrückung, der Position der syntaktischen Begrenzungssymbole oder der Benennung der Variablen. Diese dienen der besseren Lesbarkeit des Textes sowie als Verständnishilfe für andere Programmierer. Entspricht der Quellcode der Konvention, erscheint er vertrauter und kann leichter verstanden werden. Da diese Konventionen immer Regeln entsprechen, können diese leicht durch die IDE automatisiert werden. Neben den Konventionen über die Formatierung sind auch häufig gebrauchte Textoperationen je nach Programmiersprache verschieden. Das Ziel ist es, automatisch eine bessere Lesbarkeit des Textes zu gewährleisten, als auch das Schreibprozedere an sich schneller und flüssiger zu gestalten. Man unterscheidet grob drei Strategien:

1. **Einrückung**
   Beim Zeilenumbruch wird die neue Zeile so eingerückt, dass sie der Konvention entspricht. Eine weitere Option ist die automatische Einrückung einer ganzen Region im Text.

2. **Selektierung**
   Häufig werden logisch zusammenhängende Regionen im Text markiert, die dann auf unterschiedliche Weise verarbeitet werden. In Java ist es zum Beispiel sinnvoll, bei einem Doppelklick auf einen Methodenaufruf nicht nur das gesamte Wort bis zu den Begrenzungssymbolen hin zu selektieren, sondern auch die abschließenden Klammern und eventuell auch die Variable, von der die Methode stammt (d. h. vor dem Punkt). Bei Scheme hingegen hat man geklammerte Ausdrücke (S-Expressions, Klauseln) und es wäre sinnvoll, bei einem Doppelklick innerhalb einer Klausel die gesamte zu selektieren. Je nach Programmiersprache ist also eine andere Strategie bei einem Doppelklick auf eine Stelle im Quellcode notwendig.

3. **Einfügen/Löschen**
   Da die Syntax einer Sprache einer festen Definition folgt, sind viele eingefügte Symbole redundant. Beispielsweise muss in fast allen Sprachen auf jede öffnende Klammer auch eine schließende folgen, sonst liegt ein syntaktischer Fehler vor. Diese redundanten Symbole können nach Eingabe des entsprechenden Pendants automatisch eingefügt werden. Auch bei Löschoperationen kann die selbe Taktik angewendet werden: Liegt beispielsweise eine öffnende und schließende Klammer ohne Inhalt vor und löscht man die öffnende Klammer, könnte die Syntax durch die nun ungeöffnete schließende Klammer falsch sein und sowohl die Syntaxhervorhebung als auch der Parser unnötige Fehler liefern. Entfernt

man beide gleichzeitig, wird die Syntax des Programms nicht verändert. Eine Automatisierung dieser Operationen vereinfacht also die Syntax und verhindert unnötige Fehlermeldungen und unvollständigen Code.

2.3.6 Intelligente Suche

Wie bereits erwähnt wird getrennte Funktionalität häufig in verschiedenen Dateien ausgelagert. Dadurch ist zwar zum einen die Modularität der Software gegeben, andererseits gerät diese auch aus den Augen und somit aus dem Sinn. Häufig kommt es vor, dass man die ausgelagerte Funktionalität nochmal nachschlagen möchte. Dazu ist eine Suche notwendig, die nicht nur eine Volltextsuche nach einem String durchführt, sondern zwischen Deklarationen und Referenzen von Variablen unterscheiden kann. Möchte man nach einer Deklaration einer Funktion suchen, kann man direkt zu dieser Deklaration hinspringen. Andererseits sind auch häufig Szenarien gegeben, in denen die Verwendung eines deklarierten Sprachelements gesucht wird. Dazu ist es dann nötig, die Deklaration selbst zu ignorieren. Jedoch dürfen nur Referenzen auf genau diese Deklaration betrachtet werden, wodurch ein genaues Indizieren und Verfolgen der Referenzen im Modell notwendig sind.

2.3.7 Content Assist

Content Assist beschreibt die kontextuale Hilfe, die eine IDE dem Programmierer in dem Abschnitt, an dem er gerade arbeitet, zur Verfügung stellt. Dieses Feature ist in einer IDE mit Abstand am schwierigsten zu implementieren, da zu einer akkuraten Kontextbestimmung die Struktur des Programms bis zu dem Punkt bekannt sein muss, an dem der Programmierer gerade arbeitet. Allerdings ist die Struktur meistens während der Eingabe unvollständig oder fehlerhaft, sodass diese nicht genau abgebildet werden kann, wodurch der Kontext unscharf wird. Weiterhin muss der aktuelle Zustand des Programms evaluiert werden, wodurch ein interner Interpreter notwendig wird.
Die kontextualen Hilfen der meisten IDEs stellen aktuell sichtbare Variablen dar und ahnen das nächste Sprachelement (etwa Schlüsselwörter) voraus. Meist wird hierzu der abstrakte Syntaxbaum des Quelltextes benutzt, um den zu der Stelle im Quelltext zugehörigen Teilbaum zu ermitteln und darauf basierend eine Entscheidung zu treffen, wie der Kontext dieses Teilbaums aussieht und was als nächstes folgen könnte. Die Schwierigkeit hierbei liegt darin, einen AST zu konstruieren, obwohl der Code unvollständig oder fehlerhaft ist. In dem zugrunde liegenden Parser muss dafür ein aufwendiger Wiederanlauf im Fehlerfall durchgeführt werden (engl. *error recovery*), welches die Haupt-Intelligenz eines solchen Algorithmus darstellt. Ein intelligentes Content Assist setzt also eine intelligente Rückkehr im Fehlerfall voraus.

2.3.8 Automatisierung häufig verwendeter Arbeitsschritte

Aufgrund der Struktur von Programmiersprachen und daraus gebauten Programmen kehren bestimmte Arbeitsschritte für den Programmierer immer wieder. Beispiele hierfür sind etwa das Erstellen von Klassen oder die Implementierung von Interface-Definitionen. Da in diesen Operationen nur wenig bis gar keine Logik vorhanden ist sondern meistens einem Muster folgen, können diese Schritte durch die IDE gekürzt werden bzw. auf das Essentielle beschränkt werden. Dazu gibt es im Allgemeinen drei Verfahren:

- **Wizards**
  Wizards sind meistens dafür da, neue Elemente in der jeweiligen Sprache zu erstellen. Dabei wird dem Benutzer die Syntax abgenommen und die Interaktion beschränkt sich auf die Auswahl in einem Formular der für die Erstellung notwendigen Informationen. Dazu werden in Dialogen die verfügbaren Informationen geordnet präsentiert (zum Beispiel mit Listen) und können dann ausgewählt oder bearbeitet werden. Die meisten IDEs für objektorientierte Sprachen verfügen über einen Wizard, der eine neue Klasse erstellt. Der Programmierer braucht nur Informationen wie den Namen, die Superklasse oder das Interface anzugeben, woraufhin der Wizard automatisch eine neue Datei mit einer entsprechenden Klassendefinition erstellt. Da in manchen Sprachen sehr viele Schlüsselwörter für solche Definitionen gebraucht werden, spart diese Vorgehensweise Schreibarbeit und vereinfacht den Erstellungsprozess. Die Anwendung von Wizards beschränkt sich nicht nur auf das Erstellen von neuen Elementen, sondern sie eignen sich zum Beispiel auch zum Konfigurieren von bereits vorhandenen.

- **Code Templates**
  Eine Vereinfachung der Wizards stellen Code Templates dar. Diese sind vorgefertigte Schablonen für Code-Blöcke, in denen nur bestimmte Namen ausgetauscht werden müssen. Dadurch ist es möglich dem Benutzer die Definition eigener Templates zu ermöglichen. Der Nachteil hierbei ist jedoch, dass nur wenige oder keine kontextuale Informationen geboten und dem Benutzer nicht wie in einem Wizard aufbereitet präsentiert werden können.

- **Refactoring**
  Oft kommt es vor, dass bereits geschriebener Code syntaktisch geändert werden muss ohne die Semantik zu ändern, um die Lesbarkeit des Quellcodes sicherzustellen und programmiertechnische Konventionen einzuhalten. Solche Operationen wären beispielsweise die Umbenennung einer

lokalen Variable oder die Extraktion eines Code-Blocks zu einer eigenen Methode.

2.3.9 Debugging

Durch die bereits erwähnte Unfähigkeit, Laufzeitfehler vor der Ausführung zu erkennen, ist es hilfreich, Debugging-Tools anzubieten. Diese ermöglichen die Beobachtung von Zuständen des Programms während der Ausführung und damit die Identifzierung der zur Laufzeit auftretenden Fehler, wie etwa Logik-Fehler oder Exceptions.
Gängige Debugger erlauben zwei Verfahren: Zum einen das Anhalten des Programms an einer bestimmten Stelle und das Einsehen des Programmzustands mit einer schrittweisen Auswertungsfortführung und zum anderen das Abfangen bestimmter Fehler mit Einsicht der Auswertungsreihenfolge, mit der das Programm in diesen Fehlerzustand gekommen ist. Beim ersten Verfahren kann bei entsprechender Unterstützung parallel zur Auswertung die korrespondierende Code-Zeile angezeigt werden und bietet so eine Einsicht in die Auswertung des Programms. Durch diese Methode können schwer erkennbare Fehler identifiziert und behoben werden.

2.3.10 Ausführung

Um die geschriebenen Programme direkt ausführen zu können, ist eine Integration des Compilers oder Interpreters in die IDE selbst von Vorteil. Es erlaubt die zügige Übersetzung und Ausführung des Programms ohne die Benutzung von externen Programmen. Außerdem reduziert es die notwendige Zahl an geöffneten Programmen und dient so der Übersicht. Durch diese Integration lassen sich auch häufig verwendete Ausführungskonfigurationen speichern und wiederverwenden, ohne dass der Benutzer lange Befehle in eine Kommandozeile tippen muss.

2.3.11 Konsole / REPL

Wird ein Programm ausgeführt, so gibt dieses in den meisten Fällen ein Feedback an den Programmierer oder Benutzer über das, was in dem Programm geschieht. Dies erfolgt in den meisten Fällen über eine Konsole, in der sowohl Eingabe als auch Ausgabe von Informationen erscheint. Eine andere Form der Interaktion ist eine grafische Benutzeroberfläche (*Graphical User Interface*, GUI). Eine besonders enge Interaktion erfolgt bei interpretierten Sprachen, hier ist eine REPL (*Read-Eval-Print-Loop*) gebräuchlich, eine interaktive Benutzerumgebung, mit der Kommandos in der Programmiersprache sofort interpretiert werden. In ihr lässt sich die Ausführung durch sofortiges Feedback

besser verfolgen. Wieder gilt: Ist eine Konsole oder REPL bereits in der IDE integriert, müssen weniger externe Programme verwendet werden. Dies erleichtert die Benutzung beider Instrumente, da nicht erst zwischen verschiedenen Formaten gewechselt werden muss. Die IDE kann so die Konsole oder REPL automatisch für eine bequemere Benutzung konfigurieren.

Kapitel 3

Scheme 48

3.1 Scheme 48

Scheme 48 ist eine von Richard Kelsey und Jonathan Rees im August 1986 in Scheme [26] geschriebene Scheme-Implementierung [20] und wird bis heute gepflegt. Die Besonderheiten von Scheme 48 gegenüber anderen Scheme-Implementierungen stellen das umfangreiche Modulsystem sowie der Scheme-Debugger dar. Im Folgenden stelle ich zunächst Scheme kurz vor, bevor ich weiter auf Scheme 48 eingehe.

3.1.1 Scheme

Die 1975 von Guy L. Steele und Gerald Jay Sussman am *MIT Computer Science and Artificial Intelligence Laboratory* entwickelte Programmiersprache ist neben dem fast zehn Jahre später entwickelten Common Lisp[1] eine der beiden populären Lisp-Dialekte. Scheme ist eine an sich simple Sprache, die auf *S-expressions* und dem Lambda-Kalkül basiert. Es existieren zwei gebräuchliche Sprachstandards: Die aktuelle Sprachdefinition und der laut IEEE offizielle Standard ist die R^{6}RS [25], wobei die ältere R^{5}RS [19] meistens als *Scheme Standard* bezeichnet wird[2] und der häufigste implementierte Sprachstandard ist. Die Einführung von R^{6}RS wurde von den Scheme-Entwicklern unterschiedlich aufgenommen[3] und wurde nur mit einer Mehrheit von 65% des offiziellen Komitees befürwortet[4]. Einige der wichtigsten Eigenschaften der Sprache umfassen:

- Lambda-Kalkül als Grundlage

[1] http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html
[2] http://community.schemewiki.org/?scheme-faq-standards
[3] http://www.r6rs.org/ratification/additional.html
[4] http://www.r6rs.org/ratification/results.html

- funktionales, prozeduales Paradigma – unterstützt jedoch auch Elemente anderer Paradigmas, wie z. B. imperative Programmierung
- lexikalische Bindung
- starke, dynamische Typisierung
- rekursive Funktionsbeschreibung und *proper tail recursion*
- hygienische Makros
- Continuations als Objekte erster Klasse

Als eine Programmiersprache, die viele dieser Eigenschaften zum ersten Mal implementierte, hatte Scheme großen Einfluß auf kommende Generationen von Sprachen. Mittlerweile wird Scheme aufgrund seiner Einfachheit und im Vergleich zu modernen Hochsprachen flachen Lernkurve in vielen Universitäten und Schulen in der Anfängerausbildung als Lehrsprache verwendet[5].

Scheme als Lehrsprache

Anders als klassische Einsteigerprogrammiersprachen wie Pascal oder C++ erlaubt es Scheme, die Zeit die für die Erklärung der Programmiersprache selbst verwendet wird, drastisch zu reduzieren, da nur sehr wenige syntaktische Elemente vorhanden sind. Diese Zeit können die Dozenten nutzen, um andere Inhalte zu vermitteln. Seit 1999 wird Scheme an den Universitäten Tübingen und Freiburg dazu genutzt, um ein Vorlesungskonzept für die Anfängervorlesungen in Informatik zu entwickeln [12]. Dabei wurde festgestellt, dass bei Programmsprachen in der traditionellen Programmierausbildung, die nur einen geringen Abstraktionsgrad sowie viele sprachspezifische Elemente in Form von idiosynkratischen Konstrukten und komplexer Syntax aufweisen, viel Zeit und Platz selbst für einfache Programmierbeispiele beansprucht wurde. Bei Scheme hingegen ist der relevante Sprachstandard kleiner, die Programme sind kürzer und durch die höheren Abstraktionsmöglichkeiten lassen sich fortgeschrittene Programmiertechniken behandeln. Es wurde sogar mit DrRacket eine auf die Anfängerausbildung zugeschnittene IDE entwickelt, die sich besonders durch verschiedene wählbare Sprachdefinitionen auszeichnet, die auf dem jeweiligen Entwicklungsstand des Auszubildenen zugeschnitten sind. Im Folgenden stelle ich neben DrRacket auch andere IDEs für Scheme und Scheme 48 näher vor.

[5]Eine komplette Liste mit Universitäten und Schulen, die Scheme einsetzen, findet sich unter http://www.schemers.com/schools.html

Entwicklungsumgebungen für Scheme

Obwohl es sehr viele verschiedene Implementierungen[6] von Scheme gibt, mangelt es an modernen Entwicklungsumgebungen. Die bekanntesten IDEs für Scheme derzeit sind GNU Emacs und DrRacket. Emacs ist ein Texteditor, der sich vor allem durch seine Erweiterbarkeit auszeichnet. Es wurde 1976 am MIT entwickelt und über die Jahre mit unzähligen Erweiterungen ausgestattet, die Emacs teilweise zu einer vollständigen IDE mit Code Browsing, Content Assist und Debugging erhoben haben[7]. Besonders die Bedienung von Emacs ist für das Editieren von Dateien sehr komfortabel, da der Entwickler dank der vielen Tastenkombinationen ausschließlich die Tastatur verwenden kann und ohne eine Maus auskommt. Die Vielzahl von Tastenkombinationen ist allerdings für Einsteiger abschreckend.

DrRacket [17], früher DrScheme, ist die IDE der Scheme-Implementierung Racket und bietet viele Komponenten von modernen Entwicklungsumgebungen, wie eine eingebaute REPL, Code-Outline und viele Debugging-Tools. Es zeichnet sich vor allem durch seine Erweiterbarkeit durch Teachpacks und genaue Fehlermeldungen aus, wodurch es gerne in der Anfängerausbildung eingesetzt wird [12]. Die Ausrichtung auf die Anfängerausbildung beschränkt allerdings DrRacket als IDE für fortgeschrittene Prorgammierer. Zwar ist ein Modulsystem vorhanden, allerdings fehlt ein integriertes Dateiverwaltungssystem. Weiterhin fehlt die Unterstützung von vielen, bereits vorhandenen Scheme-Programmen, da der Parser, anders als in allen gängigen Implementierungen, keine Vorwärtsreferenzen auf Variablen erkennt und somit die Syntax der Programme als falsch interpretiert. Kontextuale Hilfe, etwa in Form von Code-Vervollständigung, fehlt ebenfalls.

SchemeWay [14] ist ein Plugin für Eclipse, das eine IDE für verschiedene Scheme-Implementierungen darstellt. Es unterstützt vor allem Kawa [5], ein in Java geschriebenes Framework zur Erstellung von dynamischen Sprachen, und SISC [7], eine Implementierung des R^{5}RS-Standards auf Basis der Java-Plattform. Allerdings unterstützt SchemeWay auch generische Scheme-Interpreter. In diesem Eclipse-Plugin sind bereits viele der Standard-Komponenten moderner IDEs integriert: Syntaxhervorhebung, Outline, Autoformatierung, Content Assist und eine intelligente Suche. Des Weiteren gibt es ein Modul zum grafischen Programmieren mit Kawa und SchemeWay unterstützt die entsprechenden Bibliotheken von den beiden Implementierungen. Die Komponenten der IDE sind also sehr spezifisch auf die Kawa und SISC zugeschnitten und die Unterstützung von den Besonderheiten anderer Scheme-Implementierungen fehlt. Eine Anpassung auf andere Scheme-Implementierungen ist ebenfalls nur sehr beschränkt möglich, etwa durch Angabe der Schlüsselwörter und die Festlegung der Einrückung.

[6] http://community.schemewiki.org/?scheme-faq-standards#implementations
[7] http://cedet.sourceforge.net

3.1.2 Das Modulsystem von Scheme 48 und die Configuration Language

Scheme 48 ist eine Implementierung des R^{5}RS und zeichnet sich vor allem durch sein Modulsystem aus. Zur Definition von Modulen und Schnittstellen stellt Scheme 48 eine eigene Sprachdefinition zur Verfügung, die *Configuration Language* (im Folgenden auch mit *CL* abgekürzt). Oberste Einheit ist das *Modul*, das einen isolierten Namensraum darstellt, das die Sichtbarkeit von Bindungen durch Strukturen (*Structures*) und Schnittstellen (*Interfaces*) kontrolliert. Ein Paket (*Package*) ist ein instaziiertes Modul und stellt eine Umgebung dar, in der Quelltext evaluiert werden kann. Strukturen sind im Modul definierte Sichten (*Views*), auf darunterliegende Pakete und exportieren so dessen Bindungen. Eine solche View ist dabei eine Menge von Bindungen des Pakets, welche über Schnittstellen definiert wird.

Module bestehen also aus drei Elementen:

1. Eine Menge von Strukturen, deren Bindungen im Modul sichtbar sind
2. Eine Menge von exportierten Bindungen, definiert als Schnittstellen
3. Code (z. B. Scheme), der im Modulkontext evaluiert wird (meist in eigenen Dateien ausgelagert)

Folgender CL-Code definiert beispielsweise das Modul `foo`:

```
(define-structure foo (export make-foo foo?)
  (open scheme)
  (files foo))
```

Der `define-structure`-Ausdruck bindet den Namen `foo` an das Paket, in dem die Bindungen von dem Paket `scheme` sichtbar sind und das die Datei `foo.scm` lädt, sowie eine anonyme Schnittstelle, welche die beiden Bindungen `make-foo` und `foo?` enthält. Das Paket `scheme` enthält alle im R^{5}RS definierten Sprachkonstrukte und Standard-Bindungen wie `+` oder `list`, welche `foo` aber nicht exportiert. Hier zeigt sich auch eine Besonderheit des Modulsystems: Scheme 48 stellt keine Annahmen über die Sprache der ihr unterliegenden Pakete, stattdessen geben die importierten Module diese gesondert an (in diesem Beispiel durch das Öffnen das Scheme-Pakets).

Andere Module können das Paket `foo` nun öffnen und die darin exportierten Bindungen in ihrer Implementierung verwenden.

```
(define-interface bar-interface (export make-bar))
(define-structure bar bar-interface
  (open scheme foo)
```

```
(begin
  (define (make-bar object)
    (if (foo? object)
        (make-foo 5)
        5))))
```

Dieses Beispiel zeigt die Definition der Schnittstelle `bar-interface` und des Moduls `bar`, welches die Schnittstelle `bar-interface` verwendet. Die Definition der Schnittstelle bindet den Namen `bar-interface` an eine Menge von Variablen, in diesem Fall an die Variable `make-bar`. Das Modul importiert `scheme` und `foo`, wodurch in dem eingebetteten Scheme-Code die Variablen `foo?` und `make-foo` sowie alle Sprachelemente des R^{5}RS sichtbar sind. Die Implementierungen selbst wissen von dieser Aufteilung nichts, das bedeutet, die Datei `foo.scm` weiß nicht, dass sie in der Struktur `foo` geladen wird.

Die Unterstützung der Configuration Language steht bei einer IDE für Scheme 48 im Vordergrund. Im Folgenden beziehe ich mich auf Scheme-Dateien oder *Implementierungen*, wenn Dateien Scheme-Code enthalten, und CL-Dateien oder *Moduldefinitionen*, wenn diese Code in der Configuration Language von Scheme 48 enthalten.

3.1.3 Eine Entwicklungsumgebung für Scheme 48

Für die Entwicklung mit Scheme 48 wird laut dem offiziellen Manual [20] GNU Emacs oder XEmacs empfohlen. Das *Command Package* für Scheme 48 von Emacs erlaubt die Anbindung der IDE an den Scheme-48-Prozess und die Interaktion mit diesem. Da Scheme 48 Quellcode mit spezifischen Dateien assoziiert, erweitert es die Funktionalität des Scheme-Modus, um die an den Prozess gesandten Daten mit der richtigen Datei zu assoziieren. So evaluiert der Scheme-48-Prozess den Code automatisch im Kontext des richtigen Moduls für die Datei. Neben den obligatorischen Komponenten wie Syntaxhervorhebung, automatische Formatierung und der Hervorhebung von zusammenhängenden Klammern bietet Emacs viele Emacs-typische Tastenkombinationen für die Scheme-Entwicklung[8]. Außerdem ist die einfache Erweiterbarkeit von Emacs durch Emacs Lisp hervorzuheben, mit der die Definition von eigenen Makros möglich ist. Leider fehlen in Emacs viele fortgeschrittene Komponenten moderner Entwicklungsumgebungen. Es gibt keine kontextuale Hilfe, die dem Entwickler Vorschläge macht oder kontext-spezifische Informationen anbietet. Eine Outline zur Übersicht der logischen Struktur der gerade bearbeiteten Datei fehlt ebenfalls. Der Editor markiert nicht Fehler im Programm an der entsprechenden Stelle, erst beim Evaluieren erkennt der Interpreter diese.

[8] Eine Einführung in Emacs als Scheme IDE findet sich unter http://community.schemewiki.org/?emacs-tutorial

Tatsächlich gibt es zur Zeit keine andere Entwicklungsumgebung für Scheme 48. DrRacket unterstützt weder die Sprachdefinition von Scheme 48 noch die Interaktionen mit dessen Prozess; SchemeWay fehlt die spezifische Unterstützung von dessen typischen Elementen. Besonders für Entwickler, die das textbasierte Arbeiten mit Emacs nicht gewöhnt sind, sondern eher mit grafischen IDEs wie Eclipse, Visual Studio oder NetBeans vertraut sind, kann der Einstieg in Emacs eine Hürde sein, die ein Gegenargument für die Benutzung von Scheme 48 darstellt. An dieser Stelle wird deutlich, dass eine moderne IDE für Scheme 48 notwendig ist.

Eine solche IDE sollte daher folgende Elemente haben:

- Unterstützung zur Entwicklung von Scheme-Code als auch von Moduldefinitionen, also der Configuration Language von Scheme 48.
- Ein System zur komfortablen Dateiverwaltung und Wizards zum Erstellen von neuen Dateien und Modulen. Wenn der Benutzer ein neues Modul anlegt, kann der Wizard bereits definierte oder dem Interpreter bekannte Module und Schnittstellen anzeigen und automatisch einbinden sowie eine Relation zu einer Implementierungsdatei angelegen.
- Syntaxhervorhebung zum intuitiven Verständnis der logischen Struktur eines Programms, etwa fettgedruckte Schlüsselwörter und die Hervorhebung von zusammen gehörenden Klammern.
- Eine Outline, die den strukturellen Aufbau eines Modules und einer Scheme-Datei wiederspiegelt. So können etwa Struktur- und Schnittstellendefinitionen sowie deren Elemente oder top-level Definitionen auf einem Blick sichtbar gemacht werden.
- Erkennung logischer Fehler im Programm, wie ungebundene Variablen oder Syntaxfehler und deren Markierung direkt beim Editieren.
- Umfangreiche Formatierungshilfen, die ein flüssiges Programmieren ermöglichen. Die Einrückung nach einem Schlüsselwort etwa ist anders als nach einer Prozedur, und bei einem Doppelklick innerhalb einer *S-expression* wird automatisch der gesamte Ausdruck selektiert.
- Eine intelligente Suche, die Deklarationen in verschiedenen Dateien finden kann.
- Kontextuale Hilfe wie die Anzeige von sichtbaren und importierten Variablen und Code-Vervollständigung.
- Definierbare Code-Templates, welche häufig gebrauchte Sprachkonstrukte wie `let`-Konstruke oder Schnittstellen-Definitionen abkürzen.

- Ein Refactoring, dass die Struktur eines Programms automatisch verbessert, ohne die Semantik zu ändern. Dazu zählt etwa die Bindung eines Ausdrucks an eine lokale Variable oder das Hinzufügen einer top-level Definition zu der Menge der exportierten Bindungen einer Struktur.
- Verschiedene Ausführungsmöglichkeiten für Dateien und Code, etwa innerhalb verschiedener, auswählbarer Modulkontexte.
- Die Integration des Scheme-48-Prozesses und dessen REPL.

Die Verwirklichung dieser Anforderungen stehen im Vordergrund beim Design einer Entwicklungsumgebung für Scheme 48. Im folgenden Kapitel wird beschrieben, wie ich diese Anforderungen umgesetzt habe.

Kapitel 4

Eclipse und das Plugin-System

4.1 Eclipse

Eclipse ist ein ursprünglich von IBM entwickeltes Framework für die Entwicklung und Bereitstellung von Entwicklungsumgebungen, das aus der IDE VisualAge[1] hervorgegangen ist. Seit 2001 ist der Quellcode frei verfügbar[2] und es wird seit 2004 von der Eclipse Foundation[3] betreut. Abbildung 4.1 zeigt den schematischen Aufbau des Eclipse-Projekts, bestehend aus einer Plattform und einigen offiziellen Plugins. Die Plattform ist die Umgebung, auf der Plugins geladen, integriert und ausgeführt werden können (die *Runtime*) und stellt das Grundgerüst und die Dienste wie etwa eine Grafikbibliothek oder Erweiterungspunkte für Plugins bereit, auf denen Plugins basieren. Die Plattform ist agnostisch im Hinblick auf eine zu intergrierende Programmiersprache und enthält keine Funktionalität, sondern bietet nur das Gerüst, auf dem Plugins Funktionalität integrieren können. Beispielsweise existiert eine View für Texteditoren, d. h. ein Element einer grafischen Benutzeroberfläche, das Text darstellen kann. Die typischen Elemente eines Editors einer IDE wie Syntaxhervorhebung und ähnliches müssen jedoch komplett von den Plugins geliefert werden. Neben den offiziell zur Verfügung gestellten Plugins (etwa JDT zur Java- und CDT zur C/C++-Entwicklung), die standardmäßig zusammen mit Eclipse bezogen werden können, existieren derzeit über eine Million verfügbare Plugins[4]. Aus diesem Grund wird die Plattform auch als ,,eine IDE für alles und für nichts im speziellen" beworben[5]. An dieser Stelle erläutere ich den Aufbau von Eclipse und insbesondere des Plugin-Systems.

[1] http://wiki.eclipse.org/FAQ_Where_did_Eclipse_come_from%3F
[2] http://www.eclipse.org/org/#about
[3] http://www.eclipse.org/org/
[4] http://marketplace.eclipse.org/
[5] [11], Introduction

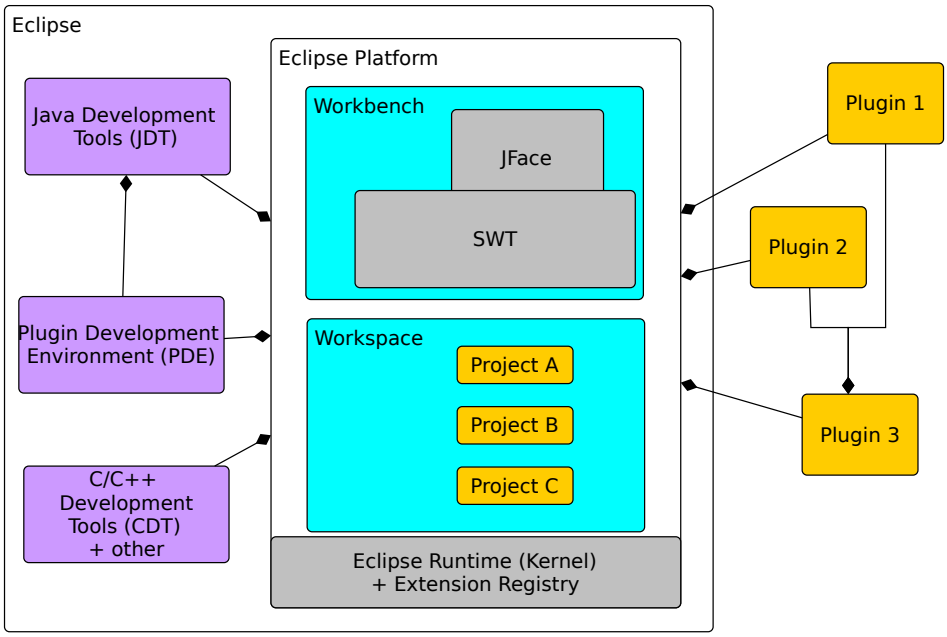


Abbildung 4.1: Aufbau des Eclipse-Projekts. Es besteht aus der Eclipse-Plattform und den offiziell bereitgestellten Plugins.

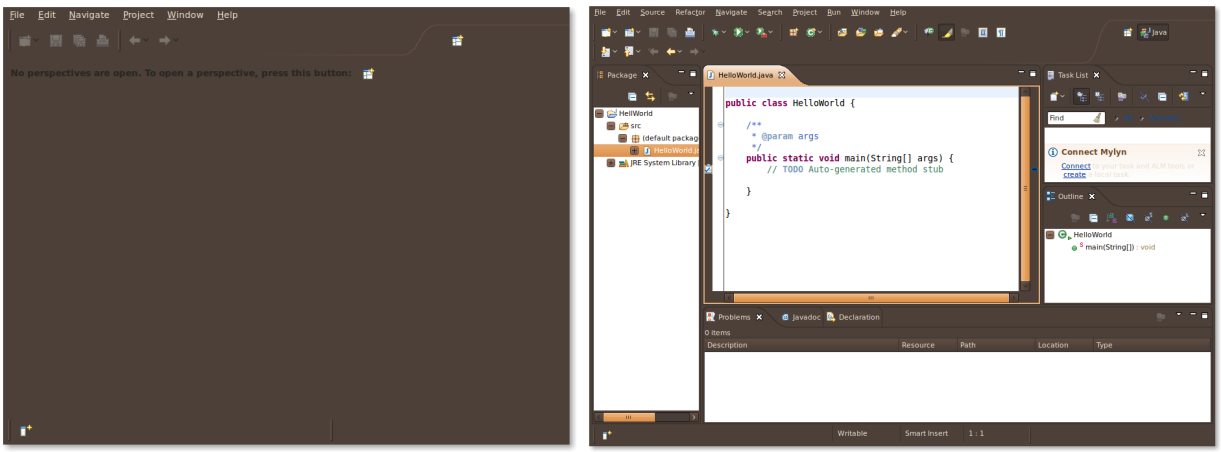
4.1.1 Aufbau der Eclipse-Plattform

In Abbildung 4.1 ist der Aufbau der Eclipse-Plattform dargestellt. Im Wesentlichen besteht sie aus drei Einheiten: Der zugrundeliegenden *Eclipse-Runtime*, der für die Interaktion mit dem Benutzer verantwortliche *Workbench* und dem *Workspace* zur Verwaltung von Dateien und Projekten.

Eclipse selbst erscheint dem Benutzer in Form der *Workbench* (vgl. Abbildung 4.2), die eine zentrale Rolle für Plugins spielt, da in ihr die Verwaltung aller Editoren, Sichten und Perspektiven stattfindet. Perspektiven sind von Plugins vordefinierte Auswahlen von Views und Editoren und bestimmen den Inhalt der Menüleiste sowie Beiträge zu bereits vorhandenen Views wie dem *Project Explorer*. Typischerweise fügen Plugins über diese wichtigen Einhängepunkte neue Editoren und Views hinzu und bieten mittels einer Perspektive dem Benutzer eine für die zu lösende Aufgabe sinnvolle Kollektion und Anordnung.

Die Benutzeroberfläche (*User Interface*, UI) von Eclipse besitzt typischerweise eine Menüleiste, eine Toolbar und diverse Sichten (*Views*). Die UI selbst bedient sich zweier Frameworks:

- Die zugrunde liegende Bibliothek SWT (Standard Widget Toolkit) und
- JFace, dass komplexere Komponenten auf Basis von SWT definiert.



(a) Ohne geöffnete Perspektiven.

(b) Mit geöffneter Perspektive des JDT-Plugins.

Abbildung 4.2: Die Eclipse-Workbench in ihrer Grundform ohne geöffnete Perspektiven und mit geöffneter Perspektive eines Plugins.

Aufgrund dessen sollten Plugins, die einen Beitrag zur UI von Eclipse liefern, ebenfalls SWT/JFace Komponenten erstellen. JFace bietet viele Klassen für häufig gebrauchte Elemente einer UI und unterstützt dabei automatisch spezifische semantische Bedingungen. Ein Beispiel hierfür sind Viewer, die bestimmte Datenstrukturen wie Listen oder Tabellen darstellen und automatisch mit den zugrundeliegenden Daten synchronisiert sind. Es steht dem Entwickler außerdem frei, die vorgefertigten Widgets von JFace je nach Bedarf mit SWT-Komponenten zu erweitern. Abbildung 4.1 zeigt die Relation der Komponenten: Die Workbench besteht aus SWT und JFace Elementen, wobei JFace auf SWT aufbaut.

Eclipse benutzt ein als *Workspace* bezeichnetes System zur Verwaltung von Dateien und Projekten. Der Workspace ist ein Ordner, dessen Unterordner *Projekte* darstellen, die diese Unterordner mit Meta-Informationen im XML-Format über deren Inhalt markieren (vgl. Abbildung 4.3). Wird zum Beispiel ein Java-Projekt angelegt, so entsteht im Workspace ein neue Unterordner, den Eclipse für die Entwicklung mit dem JDT Plugin markiert hat. Daher weiß Eclipse, welches Plugin für die Bearbeitung der dort enthaltenen Dateien verantwortlich ist. Der Workspace übernimmt auch andere Aufgaben der Dateiverwaltung, wie etwa die Konsistenzeinhaltung, indem es Änderungen an den Dateien protokolliert.

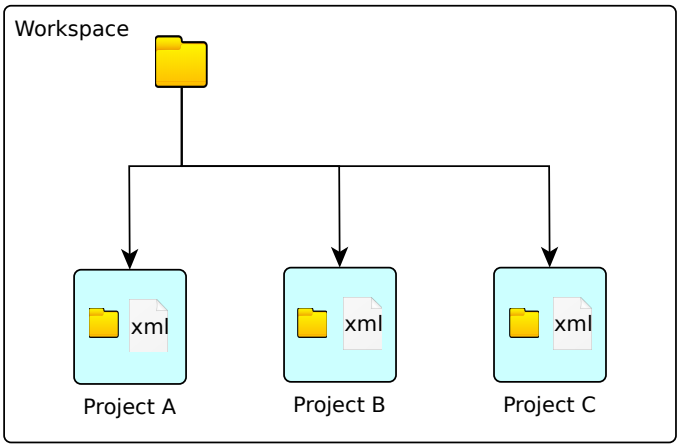


Abbildung 4.3: Eclipse definiert einen Workspace, in dem der Benutzer Projekte anlegt, die eine Abbildung von Informationen auf Unterordner darstellen.

4.1.2 Plugin-System

Eclipse selbst ist in Java implementiert, weswegen auch die Plugin-Entwicklung in der Regel in Java erfolgt. Eclipse unterscheidet die *Hostsprache*, in der Eclipse und die Plugins geschrieben sind, und die *Zielsprache*, für welche das Plugin eine Entwicklungsumgebungen bieten soll. Plugins sind die kleinste Einheit der Eclipse-Plattform, die Funktionalität separieren und in die Plattform integrieren können. Dabei interagieren installierte Plugins nicht nur mit der Plattform, sondern meistens auch mit anderen Eclipse-Plugins. Die Definition der Interaktionsmöglichkeiten erfolgt über *Extension Points* im XML-Format, die einen Austausch von Informationen in Form von Strings oder auch Klassen und Methoden ermöglichen. Beispielsweise kann ein Plugin eine Klasse, die über einen Extension Point von einem anderen Plugin eingehängt wurde, selbst instanzieren und verwenden.

Abbildung 4.4(a) zeigt die von einem Plugin bereitgestellte Erweiterung in Form eines eigenen Texteditors. Bei Rechtsklick auf einer Datei erscheint der Editor nun im Kontextmenü unter dem Punkt `Open with` (vgl. Abbildung 4.4(b)). Plugins können mittels der *XML Schema Definition* (XSD[6]), einer API für XML, eigene Extension Points definieren, um generische Funktionalität anzubieten. Beispielsweise bietet SchemeWay einen Einhängepunkt für eigene Interpreter-Klassen an, die das Plugin dann laden und benutzen kann:

```
...
<element name="extension">
    <complexType>
```

[6]http://www.eclipse.org/modeling/mdt/?project=xsd#xsd

```
      <sequence>
         <element ref="interpreter" minOccurs="1" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="point" type="string" use="required"/>
      <attribute name="id" type="string"/>
      <attribute name="name" type="string"/>
   </complexType>
</element>

<element name="interpreter">
   <complexType>
      <attribute name="class" type="string" use="required">
            <appinfo>
               <meta.attribute kind="java" basedOn=":sdt.interpreter.Interpreter"/>
            </appinfo>
         </annotation>
      </attribute>
      <attribute name="name" type="string" use="required"/>
      <attribute name="id" type="string" use="required"/>
   </complexType>
</element>
...
```

Das Feld `class` ist im obigen Beispiel als Java-Klasse markiert, die das Interface `Interpreter` implementieren muss. Diese XML-Schema-Definition eines Extension Points wird in der `plugin.xml` dazu benutzt, um in XML eine Extension zu definieren:

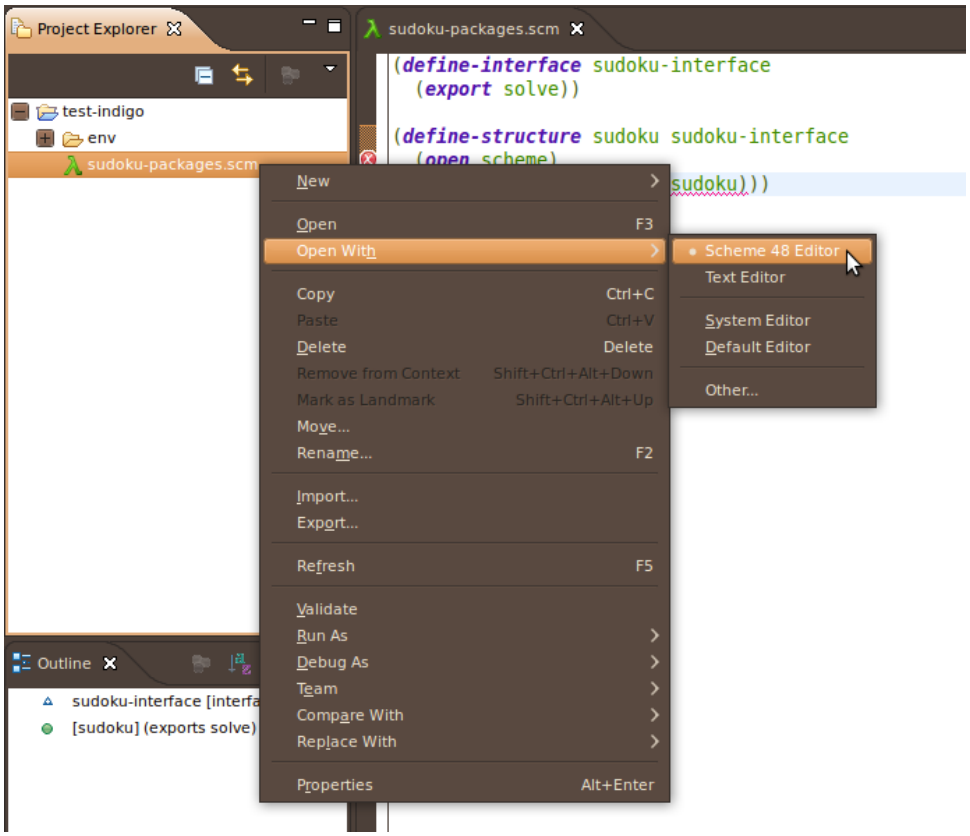
```
...
<extension point="sdt.interpreters">
      <interpreter
            class="sdt.interpreter.Scheme48Interpreter"
            id="scheme48"
            name="Scheme 48">
      </interpreter>
</extension>
...
```

Eclipse basiert auf der *Open-Services-Gateway-initiative*-Plattform (OSGi) *Equinox*[7], einer Modulsystem-Plattform für Java. OSGi implementiert ein dynamisches Komponentenmodell, das Anwendungen (Komponenten in Form von *Bundles*) integrieren und ausgeführen kann (in der ursprünglichen JVM ist das nicht möglich). Mit Equinox ist in Eclipse die *Extension Registry* realisiert.

[7] http://www.eclipse.org/equinox/

```
<extension point="org.eclipse.ui.editors">
    <editor
      id="sdt.editor.Scheme48Editor"
      class="sdt.editor.Scheme48Editor"
      icon="icons/lambda.png"
      name="Scheme 48 Editor">
    </editor>
 </extension>
```

(a) Einhängen des Editors



(b) Der eingehängte Editor in Eclipse

Abbildung 4.4: Plugins können mit dem XML-Schema Erweiterungen von Plugins in Eclipse einhängen und verfügbar machen.

Diese verwaltet alle definierten Einhängepunkte sowie alle dort eingehängten Erweiterungen. Plugins, die neue Einhängepunkte definieren, teilen diese der Extension Registry mit und können daraufhin von dieser alle Elemente beziehen, die von anderen Plugins dort eingehängt worden sind.

4.2 Entwicklung einer IDE auf Basis von Eclipse

Wie bereits erwähnt gibt die Eclipse-Plattform für die Elemente einer IDE wie Texteditor oder Outline keine konkreten Implementierungen vor. Dadurch hat man zwar auf der einen Seite den Vorteil der Gestaltungsfreiheit, auf der anderen jedoch sind die Konzepte von IDEs meist ähnlich und Entwickler müssen diese immer wieder neu implementieren. Beispielsweise muss ein Editor nicht nur Text entgegennehmen, sondern diesen auch grafisch aufbereiten und mit einem Parser verbunden sein, der den eingegebenen Text in ein logisch zusammenhängedes Konstrukt verwandelt. Dabei muss die IDE das Modell nach jeder neuen Eingabe aktualisieren.

Zur einfacheren Erstellung von IDEs existieren daher einige Frameworks, die diese häufig benötigten Schritte automatisieren oder bereits vorgefertigte und anpassbare Implementierungen bieten. Dadurch können Entwickler auf bereits bewährte Mechanismen zurückgreifen und Entwicklungszeit sparen. Der Nachteil daran ist, dass sprachspezifische Elemente durch die Abstraktion auf sprachagnostische Komponenten verlorengehen oder umständlich zu implementieren sind. Meist ist daher ein Trade-Off zwischen selbstimplementierter und frameworkspezifischer Funktionalität das Mittel der Wahl. Ich habe zwei solche Frameworks untersucht: Einerseits das *Dynamic Languages Toolkit* [3] und andererseits *Xtext*[9]. Im Folgenden erwäge ich die Vor- und Nachteile dieser Frameworks und bewerte diese im Hinblick auf die Verwendung für eine Scheme-48-IDE.

4.2.1 Dynamic Languages Toolkit

Das Dynamic Languages Toolkit, kurz DLTK, ist ein Framework zur vereinfachten und beschleunigten Erstellung von IDEs in Form von Eclipse-Plugins besonders für dynamische Programmiersprachen und wird seit 2007 stetig weiterentwickelt. Es ist quelloffen und enthält als Beispielimplementierungen vollständige IDEs für Python, Ruby, Tcl und JavaScript. Zwar wird angegeben, dass das Framework speziell dynamische Programmiersprachen unterstützt, allerdings sind dessen Bestandteile auch genausogut auf statische Programmiersprachen anwendbar. DLTK ist als eine Abstraktionsschicht zwischen Programmiersprachen (v. a. dem Parser) und der Implementierung von Views und Backend in Eclipse zu verstehen, die auf der Architektur von dem JDT-Plugin von Eclipse aufbaut. Abbildung 4.5 stellt die Beziehung zwischen Eclipse, DLTK und der zu entwickelnden IDE schematisch dar. Während die Eclipse-Plattform den groben Rahmen mittels Schnittstellen vorgibt, bietet DLTK konkrete Implementierungen, beispielsweise für einen Texteditor, einer Outline und einer intelligenten Suche. Außerdem unterstützt es die interne

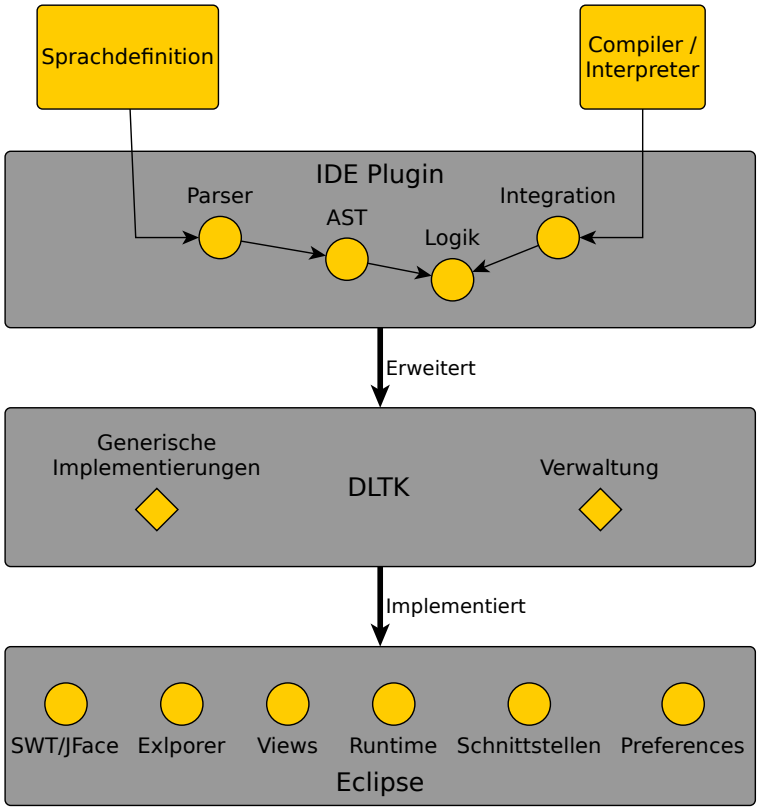


Abbildung 4.5: DLTK ist eine Abstraktionsschicht zwischen der Eclipse-Plattform, die Einhängepunkte und Schnittstellen definiert, und den sprachspezifischen Elementen einer IDE.

Handhabung des abstrakten Syntaxbaums eines von der IDE bereitgestellten Parsers, der automatisch mit dem Inhalt des Texteditors synchronisiert wird. Die IDE erweitert bestimmte Implementierungen von DLTK und fügt für die Programmiersprache spezifische Elemente ein.

Neben den enthaltenen IDEs existieren einige weitere für verschiedene Sprachen auf Basis von DLTK. Beispielsweise wurde eine IDE für die Programmiersprache D mittels DLTK erstellt [21]. Weiterhin gibt es auch ein Plugin für die Scriptsprache GAUSS[8] für die Bearbeitung und das Lösen von numerischen Problemen.

Für den Parsergenerator ANTLR [22] existiert das auf DLTK basierende Plugin ANTLR IDE[9], das mit einem stark erweiterten Editor aufwartet, in dem unter anderem ein Interpreter eingebunden ist. Auch hervorzuheben ist

[8] http://www.aptech.com/
[9] http://antlrv3ide.sourceforge.net/updates

eine IDE für die Lisp-Implementierung Clojure[10], da hiermit eine Lisp-Sprache umgesetzt wurde.

4.2.2 Xtext

Xtext ist ebenfalls ein Framework für die Entwicklung von vollständigen Programmiersprachen als auch für domänenspezifischen Sprachen (DSL) [16]. Das Grundprinzip von Xtext ist in Abbildung 4.6 dargestellt: Das Eclipse-Projekt stellt dabei die Grammatik einer Sprache in den Vordergrund und generiert selbstständig daraus mittels eines ANTLR-Parsergenerators sowohl Parser, als auch alle IDE Elemente wie ein speziell auf die Sprache ausgerichteter Editor mit Syntaxhervorhebung, Content Assist, Code-Analyse, Navigation und Code-Folding. Außerdem wird eine Outline-View generiert. Dazu wird für die Grammatik nicht nur durch den ANTLR-Parsergenerator ein Parser erstellt, sondern von Xtext auch ein Klassenmodell für den abstrakten Syntaxbaum der Sprache generiert. Aufgrund der relativen Neuheit von Xtext sind bisher nur wenige konkrete Implementierungen von IDEs auf Basis des Frameworks bekannt, so zum Beispiel eine IDE zum Erstellen von Testspezifikationen mittels domänenspezifischer Sprachen [18].

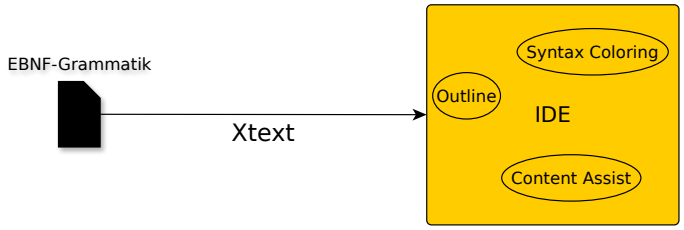


Abbildung 4.6: Das Grundprinzip von Xtext.

4.2.3 Konklusion: DLTK oder Xtext?

Der größte Unterschied zwischen DLTK und Xtext ist das zugrundeliegende Prinzip: Xtext nimmt als Eingabe eine Grammatik für die Sprache und erstellt daraus einen fertigen Texteditor mit Features wie Syntaxhervorhebung, Outline und Content Assist. DLTK hingegen fordert eine eigene Implementierung des Parsers und beschränkt sich auf eine Anbindung an dessen produzierten abstrakten Syntaxbaum. Generell haben Parser, die automatisch auf Basis einer EBNF-Grammatik generiert wurden den Vorteil, ein sehr exaktes Modell für die Verifizierung von einem Ausdruck in der Sprache zu liefern, zumal die Entwicklungszeit erheblich geringer ist als die von einem eigenen Parser. Es fehlen an dieser Stelle aber die Möglichkeiten, den Parser so zu modifizieren,

[10] http://www.slideshare.net/matthiaskoester/architektur-einer-eclipse-dltk-ide-fr-clojure

dass er Informationen über den Text in einer Weise sammelt, wie sie der Sprache gerecht wird. Dazu zählt etwa das Anlegen einer Symboltabelle. DLTK hingegen überlässt die Erstellung des Parsers dem Entwickler. Dieser kann also auf andere Parsergeneratoren wie ANTLR zurückgreifen, die eine gezielte Modifzierung des Parsers erlauben, oder einen eigenen Parser entwickeln. Ein weiterer Aspekt ist, dass Parser auf Basis von EBNF-Grammatiken zwar einfach einen Ausdruck in der Sprache verifizieren können, im Allgemeinen jedoch schlecht aus Fehlerzuständen zurückkehren können. Im Prozess des Schreibens eines Quelltextes befindet sich der Text jedoch nur selten in einem Zustand der Richtigkeit, sondern die meiste Zeit über ist er unvollständig. Der Parser muss diese Unvollständigkeit erkennen können, indem er aus dem Fehlerzustand zurückkehrt und versucht, nach dem Fehler kommende gültige Ausdrücke zu erkennen. Diese Error Recovery ist von einem automatisch erstellten Parser nur sehr schwer zu bewerkstelligen und geeignete Strategien müssen von Hand implementiert werden. Deswegen sind von Hand erstellte Parser den automatisch generierten überlegen, allerdings ist die Entwicklungszeit eines Parsers auch entsprechend hoch. Weiterhin gewährt Xtext weniger Freiheiten für eigene Implementierungen von Elementen wie Outline, Content Assist o. ä., während DLTK die Auswahl von benötigten Komponenten erlaubt.

Für eine Scheme-48-IDE ist es, wie oben erwähnt, wichtig, sowohl eine Unterstützung von Scheme als auch von Scheme 48 und deren spezifischer Sprachelemente zu gewährleisten. Aufgrund der Fixierung auf eine Grammatik ist Xtext dafür ungeeignet. DLTK hingegen bietet Funktionalität genauso wie die Freiheit eigener Komponenten und daher verwende ich es für die Implementierung einer Scheme-48-IDE.

4.3 Aufbau eines Eclipse-IDE-Plugins

Der Aufbau der meisten Eclipse-IDEs ist unabhängig von der zu implementierenden Programmiersprache. Bei größeren Projekten ist es üblich, die einzelnen Komponenten aufgrund ihrer Größe auch in verschiedenen Plugins mit eigenen Schnittstellen abzuspeichern. Gewöhnlicherweise bestehen Plugins zumindest aus zwei Teilen: Dem Kern und der Benutzeroberfläche. Andere Komponenten wie ein Debug-System, ein Ausführungsumgebung oder eine Dokumentation der Programmiersprache sind ebenfalls häufig in Eclipse-IDEs vorzufinden.

4.3.1 Kern

Der Kern der IDE ist für die Beschaffung und Verarbeitung von Informationen sowie für die Steuerung der UI zuständig. In einem Model-View-Controller-Modell würde der Kern den Controller darstellen. Zugrunde liegt die Definition der Zielsprache, anhand der ein Parser Quellcode einliest. Der Parser

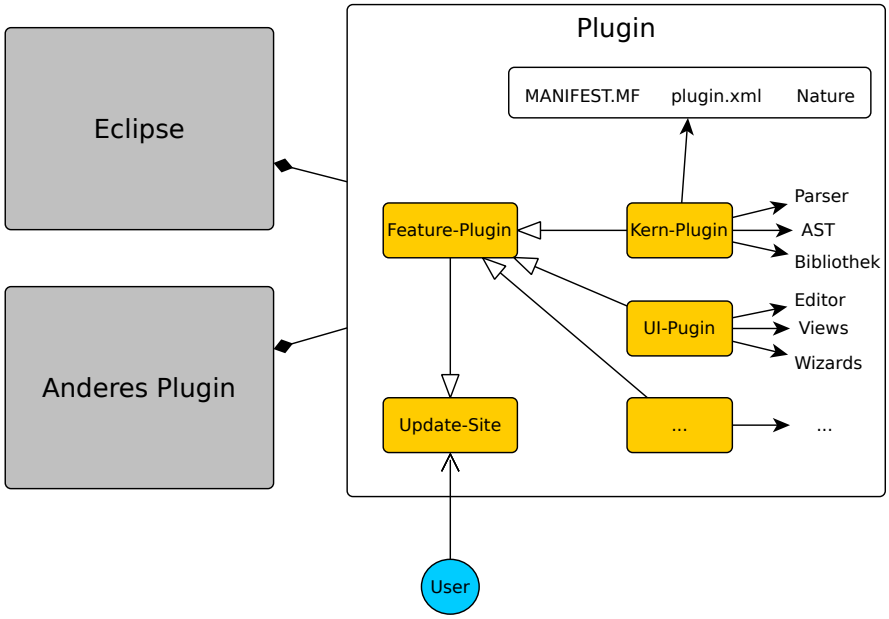


Abbildung 4.7: Die Architektur eines Plugins für Eclipse.

verifiziert den Code (d. h. erkennt Ausdrücke in der Zielsprache) nicht nur, sondern generiert auch einen abstrakten Syntaxbaum (AST), der die Syntax des Quellcodes abstrahiert darstellt. Abbildung 4.8 zeigt exemplarisch einen AST für den simplen Ausdruck `(+ 2 5)`. Die Wurzel enthält den Operator und dessen Kindknoten die Argumente. Die IDE kann den AST dazu verwenden, die Struktur der Sprache zu erfassen und sogar den Code zu interpretieren. Der Kern erstellt beispielsweise auf Basis des AST die Outline und kann gezielt nach Deklarationen suchen. Ein ausführlicher AST, der zwischen verschiedenen Sprachelementen differenziert, ist ebenfalls für eine gute Content Assist notwendig.

Der Parser kann dabei nicht nur einen AST erstellen, sondern auch eine Modellierung von Sprachelementen in Java, also der Übertragung von Konstrukten der Sprache in ein Java-Klassenmodell, vornehmen, die im Gegensatz zu einem AST nur spezifische Informationen speichert und keine Abbildung der Syntax vornimmt. Die IDE kann mit dieser Modellierung die wichtigsten Elemente und Informationen speichern und für diverse Elemente wie die Verifizierung des Programms, der Outline oder dem Content Assist benutzen. Dazu beinhaltet der Kern auch eine Bibliothek zur Verwaltung dieser Modellierungen. Um mit den Dateien synchron zu bleiben, deren Struktur sie repräsentieren, protokolliert Eclipse Veränderungen an Dateien und benach-

richtigt alle relevanten Komponenten, wenn Veränderungen auftreten. Im Falle der Bibliotheken aktualisieren diese bei einem Lösch- oder Verschiebevorgang ihren internen Bestand. Des Weiteren ist der Kern für die Bereitstellung und Steuerung aller UI-Komponenten verantwortlich.

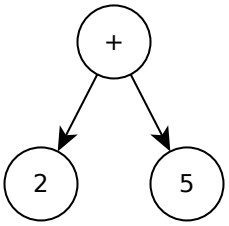


Abbildung 4.8: Der abstrakte Syntaxbaum des Ausdrucks `(+ 2 5)`.

4.3.2 Benutzeroberfläche

Die UI ist zuständig für die Interaktion mit dem Benutzer und die Darstellung von Informationen wie Dateien und Quellcode. Auch die Integration mit der Eclipse-UI wie der Workbench wird durch die UI des Plugins gesteuert.

Abbildung 4.9 zeigt die verschiedenen Komponenten der Eclipse-Benutzeroberfläche farbig markiert. Die Workbench selbst besteht aus einer oder mehreren *Workbench Windows* (rot markiert), die wiederum eine *Workbench Page* (lila markiert) beinhalten. Das Workbench Window stellt die Widgets bereit, während die Page den Inhalt bestimmt. In der Praxis existieren meist nur eine Window und Page, daher können beide als Einheit betrachtet werden. In dem Editor (blau markiert) manipuliert der Benutzer den Inhalt von Dateien. Der Editor stellt den Text verschiedenfarbig und formatiert so dar, wie der Benutzer es in den Einstellungen (*Preferences*) festlegt. Weiterhin unterstreicht er Fehler an entsprechender Stelle rot, und bei Eingabe des Textes präsentiert der Editor dem Benutzer Vorschläge. Weiterhin wird durch die Eingabe im Editor ein System zum Abgleichen des ASTs mit dem Inhalt des Editors gestartet (*Reconciler*). Dies geschieht nicht nach jedem Tastendruck, sondern wenn der Benutzer eine Pause von einer bestimmten Länge einlegt (meist ein halbe bis eine Sekunde).

Views (in der Abbildung grün markiert) sind Fenster (Widgets) innerhalb der Eclipse Workbench. Sie stellen Sichten auf Datenstrukturen dar und erlauben die Interaktion mit diesen. Ein Beispiel hierfür ist die Outline: Diese View stellt den AST des Programms in einer Baumstruktur dar. Durch einen Doppelklick auf einen Knoten springt der Editor an die Stelle der Deklaration.

Actions aus der JFace API repräsentieren Aktionen des Benutzers wie *Run* oder *Copy*. Die IDE kann dabei jede Aktion in verschiedene UI-Komponenten wie Toolbars oder Popup-Menüs über Einhängepunkte integrieren.

Weitere Komponenten der Eclipse-UI sind *Dialoge* und *Wizards*. Dialoge sind alle Popup-Fenster, in denen die IDE vom Benutzer Informationen abfragt. Wizards sind erweiterte Dialoge, die meist über mehrere aufeinanderfolgende Seiten verfügen. Diese fragen nicht nur Informationen ab, sondern führen mit diesen Informationen komplexere Operationen aus.

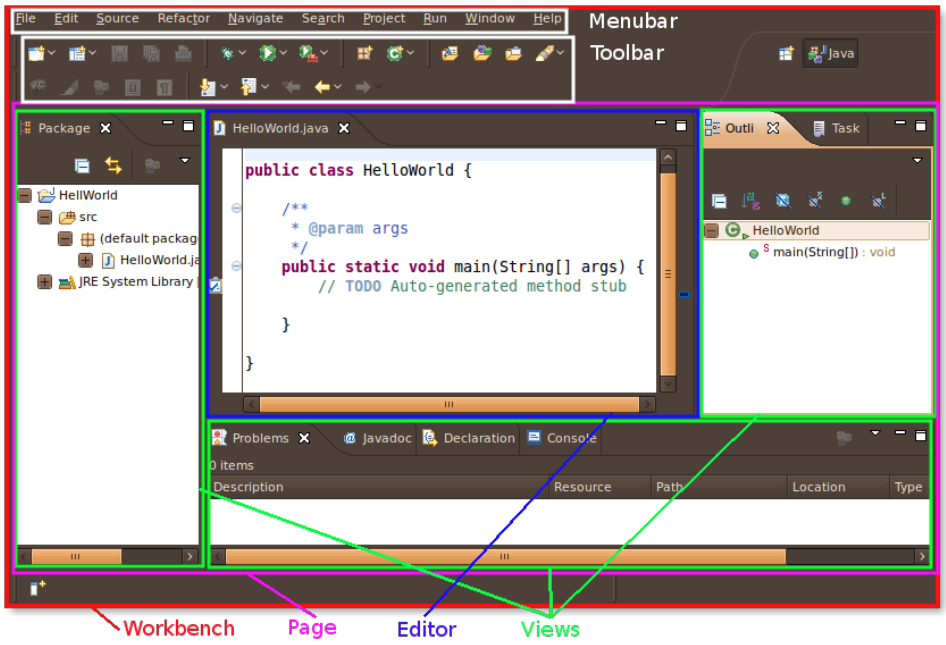


Abbildung 4.9: Die Komponenten der Eclipse-Benutzeroberfläche.

4.3.3 Sonstige Komponenten

Neben diesen Komponenten gibt es noch einige weitere wichtige Bestandteile, die notwendig oder typisch für Eclipse-Plugins sind.

- Die `plugin.xml`-Datei definiert das Plugin aus Sicht der Eclipse-Runtime. Es gibt die Beziehung zu anderen Plugins durch das Einhängen von eigenen Implementierungen in deren Einhängepunkte an und beschreibt, welche Einhängepunkte das Plugin zur Verfügung stellt.

- Die `MANIFEST.MF` stellt alle Informationen bereit, die die Eclipse-Runtime benötigt, um das Plugin erfolgreich auszuführen. Dazu gehören

Abhängigkeiten von anderen Plugins als auch Pfade zu extern eingebundenen Drittquellen wie jar-Dateien o. ä. In ihr deklariert das Plugin auch, welche Pakete es exportiert.

- Darüber hinaus muss ein Plugin eine Zeichenkette-Konstante als *Project Nature* definieren. Mit dieser Konstante kann Eclipse Projekte mit gewissen Plugins oder Werkzeugen assoziieren. Durch das Hinzufügen einer Nature zu einem Projekt weiß Eclipse, dass das Plugin, zu dem die Nature gehört, so konfiguriert ist, dass es mit dem Projekt umgehen kann. Dabei kann ein Projekt durchaus mehrere Natures gleichzeitig haben.

- Typerischerweise installiert der Benutzer Plugins in Eclipse über den *Update Manager* unter `Help -> Install New Software`. Um ein Plugin dort verfügbar zu machen, sind ein *Feature-Plugin* und eine *Update-Site* notwendig. Das Feature-Plugin gibt alle Plugins an, die zu dem auszuliefernden Plugin gehören. Außerdem spezifiziert es an dieser Stelle Meta-Informationen wie die Kategorie, die Beschreibung, das Copyright und die Lizenzrechte des Plugins. Der Wizard zeigt diese Informationen dem Benutzer beim Installieren an. Die Update-Site ist ein von einem Wizard der Eclipse-PDE generiertes Verzeichnis, dass alle benötigten Dateien des Plugins und eine `site.xml`-Datei mit Meta-Informationen enthält. Der Benutzer gibt das Verzeichnis, das entweder auf der lokalen Festplatte oder auf einem Webserver liegt, in dem Update-Manager an und bezieht darüber das Plugin.

Kapitel 5

Scheme 48 Development Tools (SDT)

In diesem Kapitel beschreibe ich die Implementierung einer IDE zur Entwicklung von Programmen mit Scheme 48: *Scheme 48 Development Tools*, kurz SDT. Der Name lehnt sich an offizielle Eclipse-Plugins wie Java Development Tools (JDT) und C/C++ Development Tooling (CDT) an. Die Komponenten des Plugins sind:

- Parser
- Abstrakter Syntaxbaum (AST)
- Editor
- Verarbeitung der Semantik von Programmen
- Modellierung der Sprache in Java
- Bibliothekssystem
- Anbindung des Scheme-48-Prozesses
- UI-Komponenten

Die einzelnen Bestandteile erläutere ich in diesem Kapitel. Abbildung 5.1 zeigt die Beziehung von SDT zu der Eclipse-Plattform und DLTK an einigen beispielhaften Elementen. Verschiedene Elemente wie der Editor oder andere UI-Komponenten sind direkte Extensions von Komponenten der Eclipse-Plattform, während andere Elemente wie der Parser oder die Content Assist eine Erweiterung von Komponenten von DLTK darstellen. Dabei spiegelt die Erweiterung nicht die Abhängigkeit oder Klassenbeziehung wieder, denn viele Elemente, die direkte Extensions von Eclipse sind, erweitern Klassen von

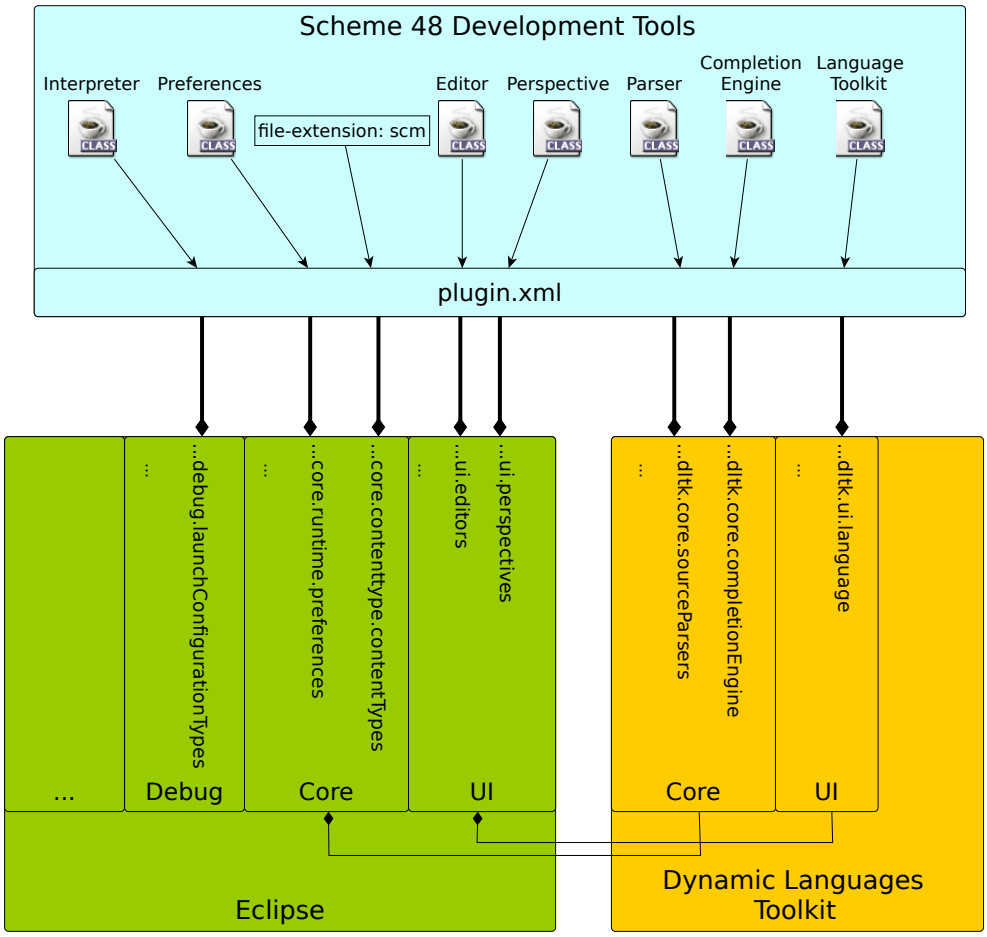


Abbildung 5.1: Die Architektur von SDT und dessen Relation zu der Eclipse-Plattform und dem Dynamic Languages Toolkit.

DLTK. Die Extension des Editors von SDT an Eclipse zum Beispiel teilt der Plattform die konkrete Editor-Klasse mit, die Implementierung der Klasse selbst ist aber eine Erweiterung der Editor-Klasse von DLTK.

5.1 Parser

Einer der wichtigsten und grundlegensten Elemente einer IDE ist der Parser. Dieser ist dafür zuständig, die Ausdrücke der zu implementierenden Sprache zu erkennen, ohne dass der Quelltext kompiliert oder interpretiert wird. Außerdem liefert der Parser alle Informationen, die für die IDE notwendig sind, um die Semantik des Programms zu verstehen und dadurch die vorher beschriebene Unterstützung des Programmierers durch die IDE zu ermöglichen. Für die Erstellung des Parsers benutze ich ANTLR.

5.2 ANTLR

ANTLR steht für *Another Tool for Language Recognition* und ist ein Parsergenerator von Terrence Parr, der seit 1989 an der Universität von San Francisco entwickelt wird. In der aktuellen dritten Version ist der Parsergenerator in der Lage, aus LL(k)-Grammatiken Lexer, Parser und TreeParser in vielen verschiedenen Hostsprachen (Java, Python, C/C++, C# etc.) zu erzeugen. Abbildung 5.2 stellt die Übersetzung von einer Grammatik in einen ausführbaren Parser schematisch dar: Die Grammatik definiert in einer EBNF-ähnlichen Form die Sprache, woraufhin der ANTLR-Compiler einen Parser in der Hostsprache (hier: Java) erstellt. Durch einen Java-Compiler (z. B. von Eclipse) kann dieser Parser ausführbar gemacht werden. Der Übersetzungsprozess vom Ausdruck einer Zielsprache (im Falle von SDT Scheme und CL) zu einem AST ist in Abbildung 5.3 dargestellt. Der Lexer verwandelt einen Zeichenstrom in eine Folge aus Tokens. Der Parser liest diese Tokens ein und versucht anhand vordefinierter Regeln die Struktur des Ausdrucks zu erkennen und in einen abstrakten Syntaxbaum zu verwandeln. Dabei ist es möglich, in den Regeln der Grammatik Ausdrücke in der Hostsprache einzubetten, um aus den Tokens und erkannten Ausdrücken Informationen für die IDE zu beziehen und somit die Semantik der Zielsprache zu verstehen. Diese Einbettung von Anweisungen in einer ANTLR-Grammatik nennt man *Actions*. Folgendes Beispiel demonstriert den Aufbau einer ANTLR-Grammatik. Gegeben sei folgende EBNF-Grammatik, die eine Sprache beschreibt, die aus beliebig vielen aufeinanderfolgenden Ausdrücken der Form `(define ...)` besteht:

⟨`program`⟩ ⟶ ⟨`definition`⟩`*`
⟨`definition`⟩ ⟶ `( define` ⟨`variable`⟩`+ )`
⟨`variable`⟩ ⟶ ⟨`letter`⟩`*`
⟨`letter`⟩ ⟶ `a | b | .. | z | A | B | ... | Z`

Die folgende ANTLR-Grammatik erzeugt einen Parser, der Ausdrücke in dieser Sprache erkennt:

```
grammar define-example;

options {  language = Java; output = AST;}

program   :   (define {System.out.println(‘‘found a definition’’);} )* EOF;
define  :   ’(’ ’define’ VARIABLE ’)’;

LETTER  :   (’a’..’z’|’A’..’Z’);
VARIABLE : LETTER*;
SPACE : (’ ’ | ’\n’ | ’\t’ | ’\r’) {$channel = HIDDEN;};
```

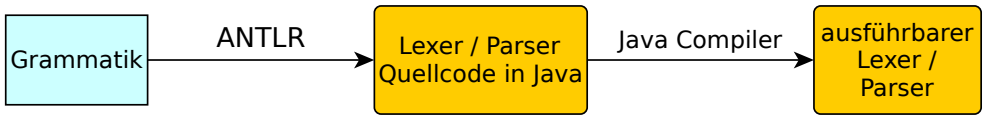


Abbildung 5.2: Der Übersetzungsprozess von einer ANTLR-Grammatik zu einem Parser.

Diese Grammatik definiert gleichzeitig die Regeln (Bezeichner gefolgt von einem Doppelpunkt) für den Parser als auch für den Lexer. Großbuchstaben markieren in einer ANTLR-Grammatik Lexer-Regeln, während Parser-Regeln klein geschrieben sind. Die ersten beiden Zeilen beschreiben Meta-Informationen für den Parser: `grammar` gibt den Typ der Grammatik an und `options` konfiguriert den Compiler von ANTLR. In diesem Fall liegt eine Lexer- und Parser-Grammatik vor und die Sprache, in der Lexer und Parser generiert werden sollen, ist Java. Die letzte Zeile der Grammatik gibt an, dass der Parser Leerzeichen und Zeilenumbrüche ignorieren soll. Geschweiften Klammern markieren Actions. In der Beispiel-Grammatik wird für jeden `define`-Ausdruck ein String auf der Konsole ausgegeben. Durch die Angabe in den `options` geben alle Parser-Regeln automatisch einen abstrakten Syntaxbaum zurück. Dadurch ist es möglich, diese an Variablen zu binden und innerhalb von Actions zu referenzieren. Wird die `program`-Regeln in obigem Beispiel folgendermaßen definiert:

```
program   :   (def=define
                 {Sytem.out.println(‘‘found a definition: ’’+$def.text);}
              )* EOF;
```

Dann gibt der Parser bei Eingabe von `(define alpha) (define beta)` auf der Konsole folgendes aus:

```
found a definition: (define alpha)
found a definition: (define beta)
```

`$def.text` referenziert das Feld `text` des Rückgabewerts der `define`-Regel, es ist allerdings auch möglich, eigene Felder zu definieren und dadurch entsprechende Werte zwischen den Regeln auszutauschen. Außerdem ermöglicht ANTLR die Definition eines *TreeWalker*, also eines Parsers, der AST-Knoten anstatt Tokens liest und den resultierenden AST des Parsers traversieren kann. Dadurch ist es möglich, auf den AST Algorithmen anzuwenden, die auf Bäumen operieren.

Die Grammatik für Scheme und CL habe ich mit der ANTLR IDE in Eclipse sowie mit einer speziellen IDE für ANTLR, ANTLRWorks [15] erstellt. Mit ANTLR IDE ist es möglich, die Grammatik in Eclipse zu bearbeiten und dem ANTLR-Compiler anzugeben, in welches Verzeichnis er die resultierende

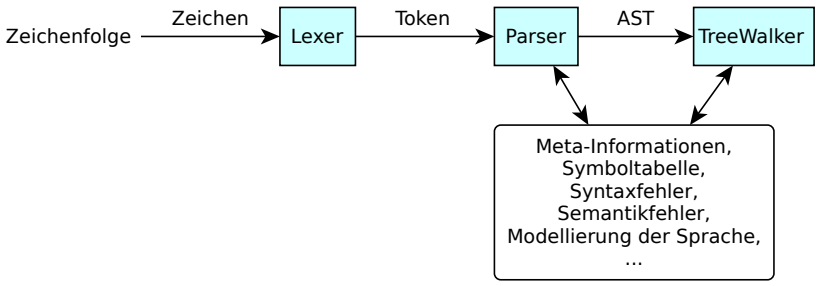


Abbildung 5.3: Der Übersetzungsprozess von einer Zeichenkette zu einem AST.

Java-Datei ablegen soll. Der Java-Compiler von Eclipse kompiliert dann automatisch die Java-Datei und untersucht die Syntax. Auf diese Weise ist ein schnelles Testen und Debuggen der Actions in der Grammatik möglich. Außerdem existieren Besonderheiten wie ein Interpreter, um gezielt Ausdrücke auf Regeln anzuwenden und der den vom Parser erzeugten AST grafisch veranschaulicht, sowie eine Railroad-View, eine Darstellung der Regeln selbst als Graphen.

ANTLRWorks ist ein eigenständige IDE zum Bau von ANTLR-Grammatiken und zeichnet sich vor allem durch seine Debugging-Möglichkeiten sowie vieler IDE-typischer Elemente wie Refactoring und Content Assist aus. Der Debugger ermöglicht es, einen Beispielausdruck schrittweise parsen zu lassen. Der Entwickler kann die Tokens schrittweise dem Parser übergeben und die IDE zeigt im Texteditor an, welche Regel der Parser gerade besucht und wie der AST dementsprechend aufgebaut wird. Außerdem werden Grammatik-Fehler im Syntaxdiagramm einer Regel grafisch veranschaulicht. ANTLRWorks eignet sich aufgrund dessen besser, die Grammatik selbst zu erstellen und zu testen, während mit ANTLR IDE die Einbettung der Actions besser vonstatten geht. Weder ANTLR IDE noch ANTLRWorks können dabei die Actions in der Hostsprache interpretieren oder ausführen.

5.2.1 Implementierung der Scheme- und CL-Grammatik in ANTLR

Zur Implementierung der Grammatiken beider Sprachen habe ich die jeweilige EBNF-Sprachdefinition als Grundlage verwendet. Da Scheme 48 den Sprachstandard R^5RS implementiert und R^6RS noch nicht vollständig unterstützt, verwendet SDT ebenfalls den R^5RS Sprachstandard. Die EBNF-Grammatik ist im offiziellen Revised Report nachzulesen [19] und ist zu umfangreich, um sie an dieser Stelle komplett zu behandeln. Stattdessen erläutere ich die Implementierung der Grammatik beispielhaft an einigen wenigen Regeln.

Die EBNF-Regel für Definitionen und der Funktionsargumente sieht folgendermaßen aus:

```
⟨definition⟩ ⟶ ( define ⟨variable⟩ ⟨expression⟩ )
               | ( define ( ⟨variable⟩ ⟨def formals⟩ ) ⟨body⟩ )
               | ( begin ⟨definition⟩* )
⟨def formals⟩ ⟶ ⟨variable⟩*
                | ⟨variable⟩* . ⟨variable⟩
```

Zunächst habe ich die Regeln ,,wörtlich" in ANTLR übersetzt:

```
definition
    :    LPAREN DEFINE variable expression RPAREN
    |    LPAREN DEFINE LPAREN variable def_formals RPAREN body RPAREN
    |    LPAREN BEGIN definition* RPAREN;
def_formals
    :    variable*
    |    variable* DOT variable;
```

Da jedoch eine EBNF-Grammatik beispielsweise wegen Links-Rekursion nicht direkt in einen Parser umgesetzt werden kann, ist eine Umschreibung der Regeln nötig. Obwohl in diesem Fall die Grammatik funktioniert, ist es trotzdem sinnvoller die Regel umzuschreiben, damit die Handhabung mit Actions leichter und die Regel übersichtlicher ist:

```
definition
    :    LPAREN DEFINE ( variable expression RPAREN
                       | LPAREN variable def_formals? RPAREN body RPAREN )
    |    LPAREN BEGIN definition* RPAREN;
def_formals
    :    variable+ (DOT variable)?
    |    DOT variable;
```

Semantische Prädikate sind ein weiteres wichtiges Element von ANTLR-Grammatiken. Semantische Prädikate sind boolesche Ausdrücke in der Hostsprache, die in einer Regel die Wahl des Parsers beeinflussen. Beispielsweise sind in Scheme Definitionen innerhalb eines Ausdrucks erlaubt, der in einem anderen Ausdruck geschachtelt ist. Definitionen, die nicht innerhalb eines anderen Ausdrucks stehen, sind top-level Definitionen und erstellen globale Bindungen. In der EBNF-Grammatik wird dieser Unterschied nicht deutlich, dort ist nur definiert, wie eine Definition auszusehen hat, jedoch nicht, wie sie sich verhält. Der Parser muss dieses Verhalten jedoch erkennen und unterscheiden.

```
definition[boolean toplevel]
    :    {toplevel}?
        LPAREN DEFINE ( variable expression RPAREN
```

```
                    | LPAREN variable def_formals? RPAREN body RPAREN )
|    {!toplevel}?
    LPAREN DEFINE ( variable expression RPAREN
                    | LPAREN variable def_formals? RPAREN body RPAREN )
|    LPAREN BEGIN definition[false]* RPAREN;
```

In diesem Beispiel wird der Regel `definition` ein boolescher Parameter `toplevel` zugeordnet. Jeder Aufruf der Regel muss dadurch mit einem booleschen Wert erfolgen, wie in der letzten Zeile des Beispiels zu sehen ist. Semantische Prädikate sind Actions gefolgt von einem Fragezeichen und bewirken, dass der Parser nur dann der Option folgt, wenn das Prädikat in der Zielsprache zu *Wahr* auswertet. In diesem Beispiel ist das semantische Prädikat jeweils die Abfrage nach dem Parameter `toplevel`: Wird die Regel von einer anderen Regel aus mit dem Wert *Wahr* aufgerufen, folgt der Parser der Option eins, mit *Falsch* der Option zwei. Dadurch unterscheidet die Grammatik zwischen top-level und geschachtelten Definitionen und die unterschiedliche Behandlung in den Actions ist möglich.

Die Configuration Language [20] von Scheme 48 besteht aus top-level Formen zur Definition von Modulen und Schnittstellen und kennt keine Sprachelemente von R^5RS, daher ist eine separate Grammatik nötig. Top-level Definitionen sind in der Configuration Language beispielsweise wie folgt definiert:

```
⟨configuration⟩ ⟶ ⟨definition⟩*
⟨definition⟩ ⟶ (define-structure ⟨name⟩ ⟨interface⟩ ⟨clause⟩* )
               | (define-structures (( ⟨name⟩ ⟨interface⟩ )* ) ⟨clause⟩ * )
               | (define-interface ⟨name⟩ ⟨interface⟩ )
               | (define-syntax ⟨name⟩ ⟨transformer-spec⟩ )
```

Die entsprechenden Regeln in der ANTLR-Grammatik sind der EBNF-Form sehr ähnlich:

```
configuration : definition* EOF;
definition
    :    LPAREN ( DEFINITION_STRUCTURE variable s48_interface clause*
                | DEFINITION_STRUCTURES LPAREN
                    ( LPAREN variable s48_interface RPAREN )+ RPAREN clause*
                | DEFINITION_INTERFACE variable s48_interface
                | DEFINE_SYNTAX variable transformer_spec )
         RPAREN;
```

5.3 AST-Generierung

Analog zum vorhergehenden Kapitel habe ich die Grammatik zur Erstellung des ASTs in ANTLR implementiert. Die AST-Generierung erfolgt automatisch

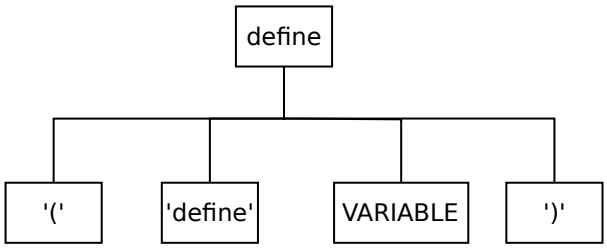


Abbildung 5.4: Der vom Parser auf Basis der Grammatik `define-example` automatisch generierte AST zur Eingabe `(define alpha)`.

durch den Parser. Jede Parser-Regel gibt standardmäßig ein Objekt der Klasse *CommonTree* zurück, in dem alle geparsten Elemente der Regel wie Tokens und Unterbäume anderer Regeln enthalten sind. Der AST ist dabei *homogen*, d. h. alle Knoten sind Objekte der selben Klasse. Der Parser der oben definierten Grammatik `define-example` liefert beispielsweise für den Ausdruck `(define alpha)` den in Abbildung 5.4 dargestellten AST. Die Wurzel des Baums ist das erste Element, in diesem Fall also das Token der runden Klammer. Die Wurzel hat drei Kinder: `'define'`, `VARIABLE` und `')'`. Oft kommt es jedoch vor, dass man den Inhalt des ASTs selbst bestimmen möchte, um etwa Informationen einzufügen oder unwichtige Dinge wegzulassen. Beispielsweise könnte es sein, dass die Tokens für runde Klammern des ASTs in Abbildung 5.4 überflüssig sind. Daher ist es möglich, den von einer Parser-Regel erzeugten Baum mittels `Rewrite-Regeln` selbst anzugeben. Folgende Modifizierung der `define-example`-Grammatik schreibt den resultierenden AST so um, dass die Klammer-Tokens nicht mehr im Baum enthalten sind:

```
grammar define-example;

options {  language = Java; output = AST;}

program   :   define* EOF;
define  :   '( ' DEF='define' VARIABLE ')' -> ^(DEF VARIABLE);

LETTER  :   ('a'..'z'|'A'..'Z');
VARIABLE : LETTER*;
SPACE : (' ' | '\n' | '\t' | '\r') {$channel = HIDDEN;};
```

Die Parser-Regel `define` der Grammatik liefert zu der Eingabe `(define alpha)` einen AST mit `DEF` als Wurzel und `VARIABLE` als einzigen Kindknoten.

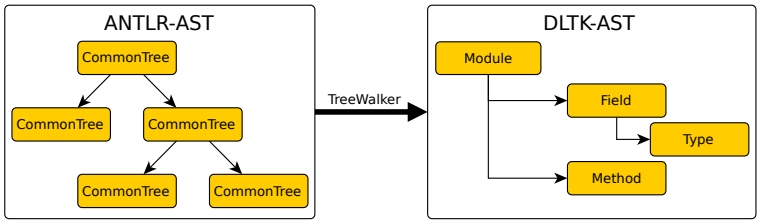


Abbildung 5.5: Der TreeWalker übersetzt einen homogenen ANTLR-AST in einen heterogenen DLTK-AST.

5.4 ANTLR-AST und DLTK-AST

ANTLR erlaubt die Angabe einer eigenen Implementierung von AST-Knoten als Resultat des Parsers, bietet aber auch eine standardmäßige Implementation in Form von eines homogenen ASTs aus `CommonTree`-Objekten. Ein homogener AST stellt alle Knoten durch gleiche Objekte dar, während ein heterogener AST verschiedene Knotenobjekte hat. Im Allgemeinen ist ein homogener AST zwar einfacher zu implementieren, allerdings ist die Gewinnung von Informationen zur semantischen Unterstützung aus einem heterogenen AST einfacher und umfangreicher. ANTLR gibt dabei keine konkrete Java-Implementierung der AST-Knoten vor, sondern verlangt stattdessen eine Adapter-Klasse `TreeAdaptor`, die sowohl eine Factory-Klasse für die selbst implementierten AST-Knoten ist, als auch Methoden zum Navigieren im Baum bereitstellt. Der ANTLR-Parser in SDT liefert einen AST aus `CommonTree`-Objekten.

DLTK verwendet zur Bereitstellung von vorimplementierter Funktionalität, beispielsweise für eine Outline oder die intelligente Suche, einen von JDT abgeleiteten AST mit Standardimplementierung für top-level Java-Elemente wie Module, Typen, Methoden, Felder etc. DLTK erzwingt zwar nicht die Be-

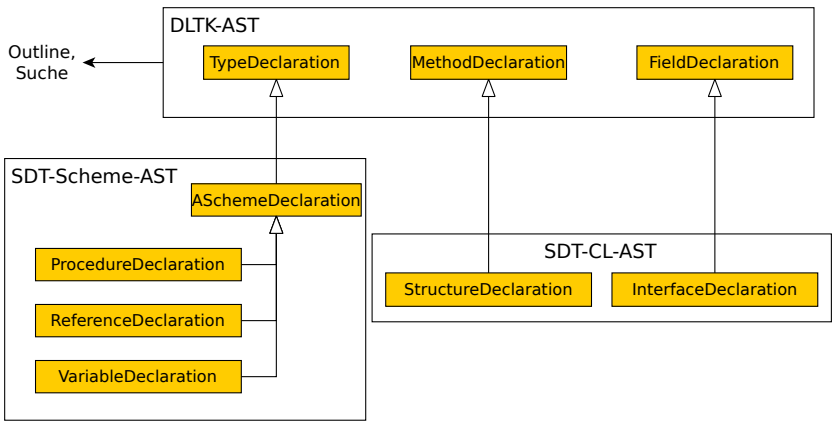
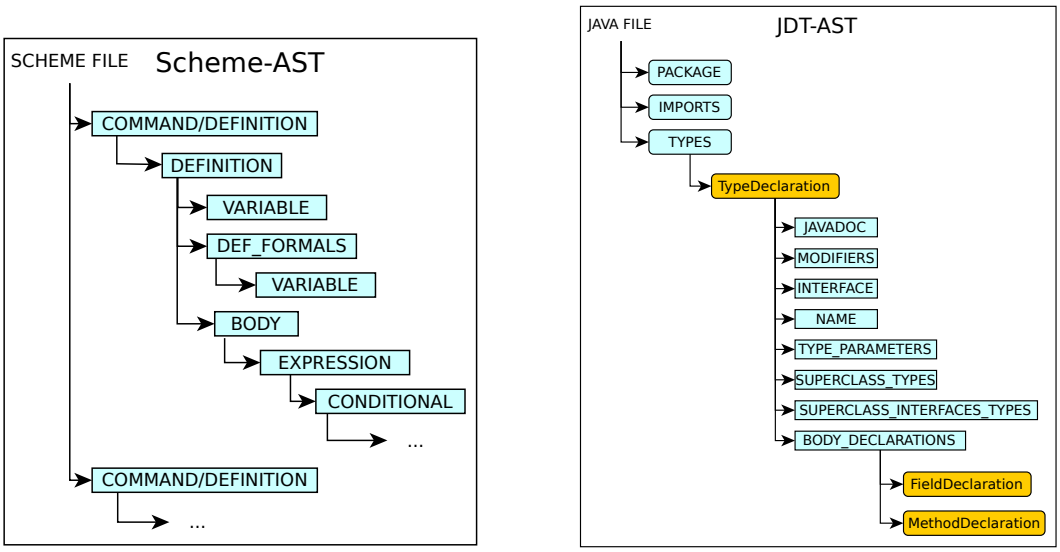


Abbildung 5.6: Abbildung des Scheme- und CL-ASTs auf den DLTK-AST.



(a) Der AST eines Scheme-Programms.

(b) Der AST eines Java Programms.

Abbildung 5.7: Der Scheme- und JDT-AST im Vergleich.

nutzung des DLTK-ASTs, aber ohne diesen ist die von DLTK bereitgestellte, bereits implementierte Funktionalität nicht nutzbar. Daher benutze ich für die Repräsentierung des Scheme- und CL-ASTs in SDT Klassen, die von DLTK-AST-Klassen ableiten (vgl. Abbildung 5.6).

Um den homogenen ANTLR-AST in einen DLTK-AST zu übersetzen, habe ich einen `TreeWalker` verwendet (vgl. Abbildung 5.5), der ebenfalls in ANTLR implementiert ist. Der `TreeWalker` traversiert den AST von ANTLR und generiert bei diversen Knoten entsprechende Scheme- oder CL-AST-Knoten, welche DLTK-AST-Klassen ableiten.

5.4.1 Der Scheme- und CL-AST in SDT

Abbildung 5.7 zeigt exemplarisch jeweils den AST eines einfachen Scheme- und Java-Programms. Die Unterschiede zwischen den Bäumen spiegeln die verschiedenen Programmierparadigmen wider: Java ist objektorientiert, daher sind Klassen (Typen) die zentralen Elemente. Im funktionalen Scheme hingegen gibt es kein vergleichbares Konstrukt und quasi alle Sprachelemente können auch in der top-level Ebene in beliebiger Reihenfolge vorkommen. In der Configuration Language hingegen gibt es nur wenige verschiedene top-level Elemente: Module und Schnittstellen. Aufgrund der Unterschiede in den syntaktischen Elementen von Java und Scheme ist eine äquivalente Implementierung des Scheme- und CL-ASTs mit dem DLTK-AST ohne Informationsverlust nicht möglich. Ziel war es primär, durch eine Abbildung die bereitge-

stellte Funktionalität von DLTK nutzbar zu machen. Dazu beschränkt sich die Implementierung des SDT-Scheme- und SDT-CL-ASTs auf top-level Definitionen und deren Unterscheidung. Abbildung 5.6 stellt die Implementierung der beiden Syntaxbäume grafisch dar. Alle top-level Definition in Scheme bilden auf die Klasse `TypeDeclaration` ab. Dazu wird zwischen drei Entitäten unterschieden: Prozeduren, Variablen und Referenzen auf andere Variablen. Prozeduren sind alle top-level Definitionen, die an einen Lambda-Ausdruck gebunden sind. Referenzen sind Umbenennungen von Variablen (z. B. ist in dem Ausdruck `(define a 3) (define b a)` die Variable `b` eine Referenz). Variablendeklarationen sind alle andere Formen der Bindung: Konstanten wie Strings oder Zahlen, aber auch komplexere Elemente wie let-Ausdrücke. In CL bilden hingegen Strukturen auf `MethodDeclaration` und Schnittstellen auf `FieldDeclaration` ab. Dadurch kann in der von DLTK bereitgestellten Suche gezielt nach der Deklaration von Strukturen und Schnittstellen in CL-Dateien und top-level Bindungen in Scheme-Dateien unterschieden werden.

Durch die Abstraktion des Scheme- und CL-ASTs auf top-level Bindungen gehen viele Informationen verloren. So spiegelt der AST nur die globalen Bindungen wieder, während er den Inhalt und Aufbau einer solchen Bindung bis auf den ungefähren Typ nicht darstellen kann. Eine Folge dieses Informationsverlusts ist zum Beispiel, dass die DLTK-Suche auf diese Weise nicht die Benutzung von Variablen finden kann, und fortgeschrittene Funktionen in der Content Assist wie die Anzeige von lokal gebundenen Variablen eingeschränkt sind.

5.4.2 Einbindung des Parsers in DLTK

Um den fertigen Parser in DLTK einzubinden, muss das Plugin zwei Klassen bereitstellen: eine Klasse, die das Factory-Pattern und das Eclipse-Interface `ISourceParserFactory` implementiert, und eine `SourceParser`-Klasse, die eine Erweiterung der abstrakten Klasse `AbstractSourceParser` ist. Die Factory-Klasse wird über den Einhängepunkt `org.eclipse.dltk.core.sourceParsers` DLTK mitgeteilt. Die `SourceParser`-Klasse hat als einzige Methode `parse`, welche als Argument ein `IModuleSource`-Objekt bekommt, das die zu parsende Datei darstellt, und eine Instanz von `IProblemReporter`, an die der Parser Fehler im Quelltext berichtet und später im Texteditor anzeigt. Die `parse`-Methode gibt ein Objekt der Klasse `IModuleDeclaration` zurück, die einen Wrapper für den AST darstellt.

5.5 Editor

Der Editor ist die zentrale Komponente der Benutzeroberfläche und der Ort mit der meisten Interaktion. Aus diesem Grund behandelt Eclipse Editoren

anders als Views: Eine Workbench besteht aus *ViewParts*, die Views darstellen, und einem *EditorPart*, der einen Editor darstellt. Anders als Views kann die Perspektive den EditorPart nicht entfernen, sondern nur verstecken. Der Editor selbst besteht aus einem Viewer und dem darzustellenden Dokument. Der Viewer des Editors besteht aus dem Textbereich und einer Leiste auf der linken Seite für Annotationen im gerade sichtbaren Bereich des Textes, dem `Vertical Ruler`, und einer Leiste auf der rechten Seite auf für alle Annotationen des Textes, dem `Overview Ruler`.

Der SDT-Texteditor ist abgeleitet von der Klasse `ScriptEditor`, der Standardimplementierung eines Editors von DLTK. Zu einem Texteditor in DLTK gehören folgende Komponenten:

- Editor

  Eine Klasse, die von `ScriptEditor` erbt und den eigentlichen Editor darstellt. Das Plugin stellt in einer Extension zu `org.eclipse.ui.editors` die Klasse direkt der Eclipse-Plattform zur Verfügung.

- `SourceViewerConfiguration`

  Der Editor benötigt zur Konfiguration seines Viewers eine Instanz von `SourceViewerConfiguration`. Diese Klasse ist für alle Darstellungen und besondere Funktionen des Editors zuständig und wird von dem Editor selbst über eine Factory-Klasse erstellt.

- `IDocumentSetupParticipant`

  Der `IDocumentSetupParticipant` ist dafür zuständig, ein Dokument für die Verarbeitung in dem Editor und der `SourceViewerConfiguration` vorzubereiten, indem er die Partitionierung des Dokuments festlegt. Die Mitteilung der Klasse an Eclipse erfolgt über den Einhängepunkt `org.eclipse.core.filebuffers.documentSetup`.

- Partitionierung

  Die Partitionierung unterteilt das darzustellende Dokument in verschiedene Bereiche (Partitionen) wie zum Beispiel Kommentare, Code oder Strings. Zum einen ist die Syntaxhervorhebung für verschiedene Partitionen festgelegt, zum anderen sind alle Funktionen wie Formatierungshilfen und Content Assist einer oder mehreren Partitionen zugewiesen. Beispielsweise ist die automatische Einrückung nach einer neuen Zeile innerhalb der Partition des Programmcodes anders als in einem Kommentar. Dazu ermittelt der Editor die Partition, die der Benutzer editiert hat und ruft dementsprechend verantwortliche Klassen auf. Die Partitionen sind als String-Konstanten festgelegt und in SDT in der Klasse `Scheme48Partitions` zu finden. SDT unterteilt Dokumente in Code (Standardpartition), Kommentare und Strings.

- `CodeScanner`

  Der `CodeScanner` ist dafür zuständig, das Dokument in verschiedene Partitionen zu unterteilen. Dafür bietet Eclipse eine Klasse `RuleBasedPartitionScanner` an, die Regeln in Form von JFace-Klassen für die Partitionen festgelegt. Diese Regeln produzieren für die Partitionen *Kommentare* und *Strings* JFace-`ITokens`. Für das Scannen der Standardpartition hingegen ist der ANTLR-Lexer zuständig. Der Nachteil davon ist, dass der Lexer eine Zeichenfolge in ANTLR-Tokens übersetzt und die IDE diese zuerst auf Tokens von JFace bzw. DLTK abgebilden muss. Dies nimmt der `ANTLRTokenScanner` vor, der ANTLR-Tokens auf die Formatierungsvorschrift abbildet, die in den Einstellungen festgelegt ist und auf dieser Basis JFace-Tokens erstellt.

- `PresentationReconciler` und `DamagerRepairer`

  Für die Abbildung von Partitionen auf Formatierungen ist der `PresentationReconciler` zuständig. Dieser legt für jede Partition einen `DamagerRepairer` an, der dafür zuständig ist, die Tokens in der Partition richtig zu formatieren. Editiert der Benutzer eine Partition, ruft der `PresentationReconciler` den zuständigen `DamagerRepairer` auf. Dieser ermittelt den editierten Bereich und aktualisiert die Darstellung. Der `PresentationReconciler` wird in der `SourceViewerConfiguration` angelegt und mit den DamagerRepairern und den zuständigen Partitionen konfiguriert.

- Einstellungen

  In den Einstellungen kann der Benutzer das Verhalten des Editors festlegen: Die Anzeige von Fehlern und Warnungen, die allgemeine Schriftgröße und das Aussehen des Editors und vor allem die Art der Darstellung von einzelnen Elementen der Sprache. Die Darstellung der Einstellungen ist in Eclipse über eine Baumstruktur gegeben. Die Knoten stellen Kategorien dar, die das Plugin festlegt. In der `plugin.xml` hängt das Plugin dann über den Einhängepunkt `org.eclipse.ui.preferencePages` Klassen ein, die den Aufbau einer Einstellungsseite beschreiben. DLTK bietet vorgefertigte Seiten an, die das Plugin ableiten und einhängen kann. Dazu sind zwei Klassen nötig: Eine, die `AbstractConfigurationBlockPreferencePage` erweitert und die Seite selbst sowie den grafischen Aufbau darstellt, und eine Ableitung der Klasse `AbstractConfigurationBlock`, welche den Inhalt der Einstellungsseite definiert. Weiterhin muss das Plugin eine Klasse definieren, die initiale Werte für die Einstellungen hält, und diese über den Einhängepunkt `org.eclipse.core.runtime.preferences` der Eclipse-Plattform mitteilen. Auf diese Weise erhält man von DLTK beispiels-

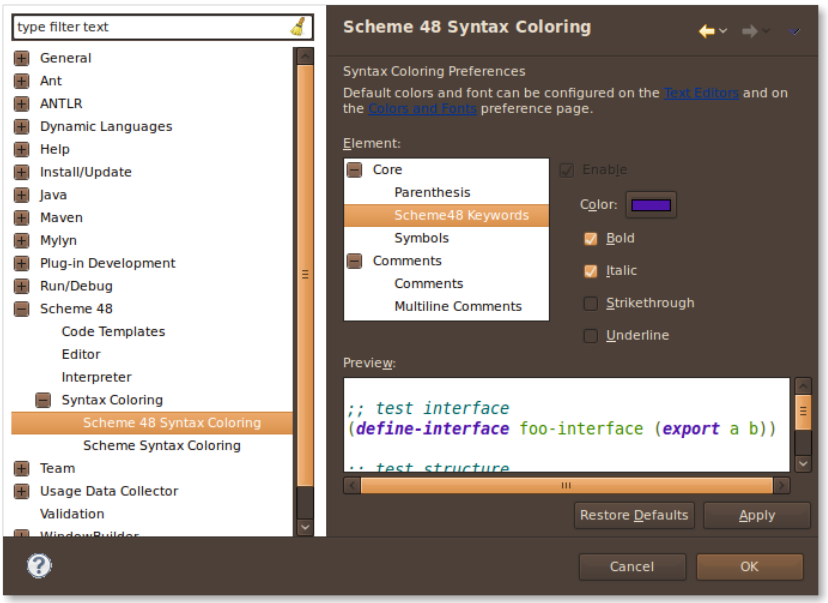


Abbildung 5.8: Die Einstellungsseite für Syntaxhervorhebung von SDT, bereitgestellt von DLTK.

weise Einstellungen für Syntaxhervorhebung (vgl. Abbildung 5.8) und Code-Templates.

- `EditorContributor`

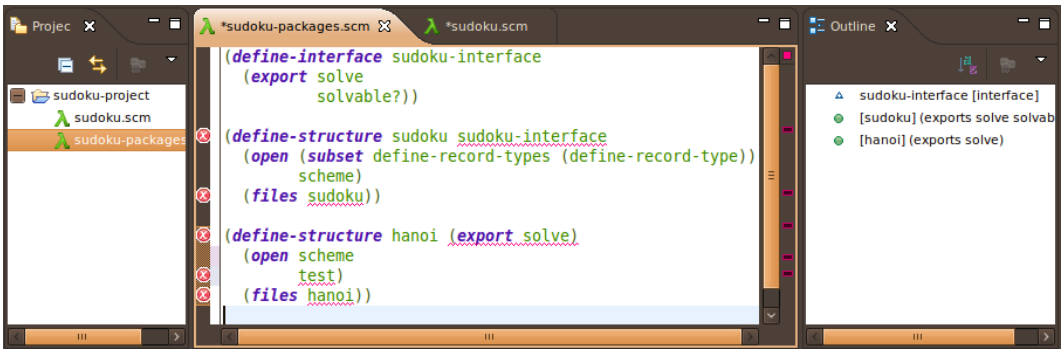
  Das Plugin kann zusätzlich eine `EditorContributor`-Klasse bereitstellen, die das `IEditorActionBarContributor`-Interface implementiert und Beiträge zu anderen UI-Komponenten wie der Toolbar leisten kann. Der EditorContributor legt beispielsweise fest, welche Elemente in der Toolbar erscheinen, wenn der Editor geöffnet ist und wie diese sich verhalten. Die Klasse wird über den selben Einhängepunkt wie der Editor selbst in die Eclipse-Plattform eingehängt.

Im Folgenden beschreibe ich die konkreten Fähigkeiten des Editors von SDT und erkläre deren Aufbau.

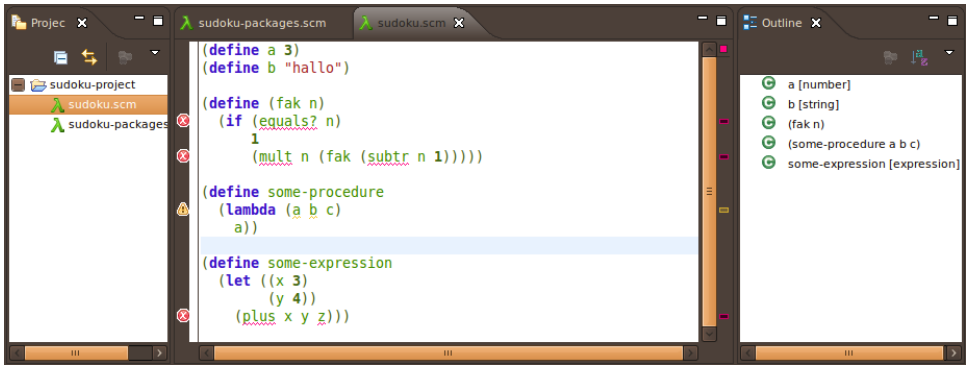
5.5.1 Besondere Fähigkeiten

Der Editor ist in Abbildung 5.9 dargestellt. Das obere Bild zeigt den Editor beim Bearbeiten einer Moduldefinition, das untere beim Bearbeiten einer Implementierung. In beiden Modi ist auf der rechten Seite die Outline zu sehen, die top-level Definitionen auflistet.

- **Outline**



(a) Der Editor im CL-Modus.



(b) Der Editor im Scheme-Modus.

Abbildung 5.9: Ausschnitte aus Eclipse mit dem SDT-Editor jeweils im Modus zur Bearbeitung von einer Moduldefinition und Implementierung.

Die Outline ist ein von DLTK bereitgestelltes Widget und stellt den Inhalt der von dem Parser generierten `ModuleDeclaration` dar. Dazu sind folgende Komponenten notwendig:

- Eine Klasse, die von `ScriptOutlinePage` erbt und somit die Standardimplementierung von DLTK zur Darstellung der Outline bereitstellt.
- Eine Klasse, die von `ScriptOutlineInformationControl` erbt und somit die Standardimplementierung von DLTK zur Synchronisierung und Aktualisierung der Outline bereitstellt.
- Überschreiben der Methode `doCreateOutlinePage()` in der Editor-Klasse, welche die OutlinePage instanziiert.
- Erstellen eines `TreeWalkers`, der den AST in der `ModuleDeclaration` traversiert und bei jedem Knoten die Informationen sammelt, welche die Outline darstellen soll.

Die Outline des Editors im CL-Modus ist auf Abbildung 5.9(a) zu se-

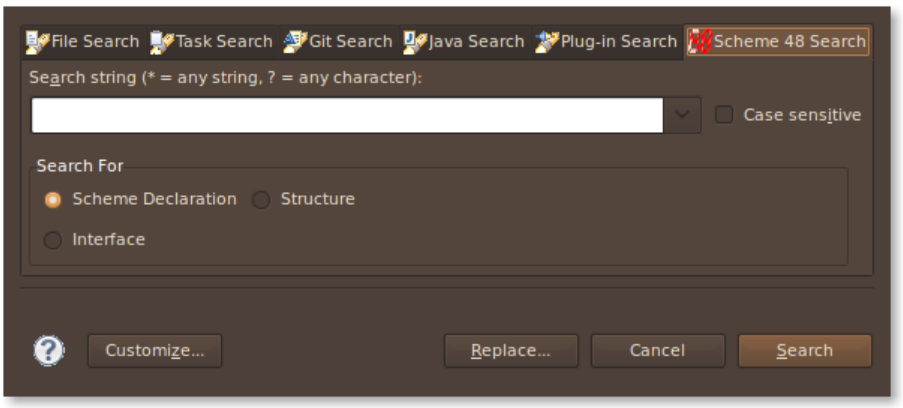


Abbildung 5.10: Die intelligente Suche in SDT.

hen. Die Outline stellt Module und Schnittstellen als eigene Knoten dar und führt zusätzlich dazu die von den Strukturen exportierten Variablen auf. Im Scheme-Modus (Abbildung 5.9(b)) werden alle top-level Bindungen als eigene Knoten aufgeführt. Die Outline führt Prozeduren, also Bindungen an Lambda-Ausdrücke, zusammen mit ihren Argumenten in S-expression-Form auf (beispielsweise ist die Darstellung der Prozedur `some-procedure` mit ihren drei Argumenten so, wie ein Aufruf aussehen würde: `(some-procedure a b c)`). Alle anderen Bindungen sind Variablen und die Knoten entsprechen den Variablennamen mit den Typen des gebundenen Werts: `a` → `number`, `b` → `string` und `some-expression` → `expression`.

- **Intelligente Suche**

  Zur Nutzung der intelligenten Suche ist die Standardimplementierung von DLTK nutzbar. Das Plugin leitet dazu die Klassen `ScriptSearchPage`, `MatchLocatorParser` und `AbstractSearchFactory` ab. `ScriptSearchPage` liefert die in Abbildung 5.10 dargestellte Benutzeroberfläche der Suche. Der `MatchLocatorParser` ist dafür verantwortlich, die SDT-AST-Klassen zu separieren und aufzurufen und wird vom der Factory bereitgestellt. Die Suche kann die Deklaration von top-level Bindungen in Scheme-Dateien und CL-Dateien finden.

- **Fehlerannotierungen**

  Wie auf den Abbildungen zu sehen ist, zeigt der Editor an manchen Stellen Annotierungen in Form von roten und gelben Markierungen an, die Fehler und Warnungen darstellen. Für die Darstellung dieser Annotationen ist ein `IProblemReporter` zuständig. Die `parse`-Methode des `SourceParser` erhält den `IProblemReporter` und reicht an diesen Fehler und Warnungen in Form von `DefaultProblems`, welche die Fehlermel-

dung und die Position im Text angeben. Um Syntaxfehler anzeigen zu lassen, muss der ANTLR-Parser die Fehler, die während der Übersetzung auftreten, in `DefaultProblems` übersetzen. Der `SourceParser`, der den eigentlichen Parser startet, sammelt nach dem Übersetzungsvorgang diverse Informationen vom Parser (wie bspw. eine Liste von ungebundenen Variablen) und berichtet diese an den `IProblemReporter`.

- **Formatierungshilfen**

  Die Hilfen, die der Editor zur Formatierung von geschriebenem Text anbietet, stellt die `SourceViewerConfiguration`. Dazu ist es nötig, die Methode `getAutoEditStrategies` zu überschreiben. Diese gibt ein Array von Klassen zurück, die das Interface `IAutoEditStrategy` implementieren. Dieses fordert die Methode `customizeDocumentCommand`, welche als Argument das Dokument in seinem aktuellen Zustand bekommt, und ein Objekt der Klasse `DocumentCommand`, welche die Änderung beschreibt, die der Benutzer an dem Dokument vornimmt. Meistens entsprechen diese Änderungen dem Hinzufügen oder Löschen eines Strings im Dokument wenn der Benutzer Text eintippt. Eine `IAutoEditStrategy` kann diese Veränderung am Dokument beliebig modifzieren. Der SDT-Editor bietet folgende automatische Formatierungshilfen an:

  - **Aus- und Einkommentieren per Knopfdruck**

    Eclipse-typisch kommentiert der Editor automatisch ganze Zeilen mit einem Semikolon aus und wieder ein, wenn der Benutzer die Tastenkombination `Ctrl + 7` drückt.

  - **Schließen von geöffneten Klammern**

    Tippt der Benutzer eine öffnende Klammer ein, überprüft das Plugin, ob diese Klammer eine bereits vorhandene passende schließende Klammer schließt. Ist das nicht der Fall, will der Benutzer einen neuen geklammerten Ausdruck erstellen, weshalb automatisch hinter der öffnenden Klammer eine schließende gesetzt wird. Damit soll es der Benutzer einfacher haben, die Klammern im Quelltext zu balancieren.

  - **Löschen von leeren geklammerten Ausdrücken**

    Befindet sich der Zeiger direkt nach einer öffnenden Klammern und ist der geklammerte Ausdruck leer, löscht das Plugin den gesamten Ausdruck, falls der Benutzer die öffnende Klammer löscht.

  - **Schließende Klammer überspringen**

    Oft kommt es vor, dass der Benutzer einen geklammerten Ausdruck von Hand wieder schließen will, obwohl das Programm automatisch die schließenden Klammern eingefügt hat. Somit würde oft die Balancierung der Klammern zerstört werden. Um das zu verhindern,

überspringt der Editor das Einfügen einer schließenden Klammer, falls der Ausdruck schon richtig geklammert ist.

- **Doppelklick selektiert ganzen geklammerten Ausdruck**
  Geklammerte Ausdrücke sind in Scheme logisch zusammenhängende syntaktische Teile. Daher kommt es oft vor, dass der Benutzer einen ganzen Ausdruck selektieren will, etwa um ihn zu kopieren. Ein Doppelklick innerhalb eines Ausdrucks selektiert daher den gesamten Ausdruck.
- **Einrückung neuer Zeilen**
  Eine der wichtigsten Formatierungshilfen ist die automatische Einrückung nach Eingabe eines Zeilenumbruchs. In Scheme entscheidet die Einrückung einer neuen Zeile die Art des Operators des Ausdrucks in der die Zeile umbrochen wird. Bei manchen Schlüsselwörtern wie `define`, `lambda` oder `begin` ist die Einrückung der neuen Zeile einfach zwei zusätzliche Leerzeichen, nach Prozeduren und einigen anderen Schlüsselwörter wie `cond` oder `if` hingegen soviele Leerzeichen, wie der Name lang ist, damit die Argumente auf der selben Höhe platziert werden. Ein Beispiel für ein korrekt eingerücktes Scheme-Programm ist folgendes:

  ```
  (define faculty
    (lambda (n)
      (if (= n 1)
          n
          (* n
             (faculty (- n 1))))))
  ```

  Daher findet das Plugin bei Eingabe eines Zeilenumbruchs den Typ des Operators des Ausdrucks heraus und entscheidet daraufhin, wie die Einrückung der nächsten Zeile aussehen soll.

Mit diesen Formatierungshilfen ermöglicht der Editor ein flüssiges Schreiben von Programmen mit Scheme-Syntax. Darüber hinaus zeigt der Editor wichtige Annotierungen im Quellcode an, die auf die Semantik des Programms Bezug nehmen. Im Folgenden beschreibe ich, welche Semantik Scheme- und CL-Programme beinhalten, die relevant für eine IDE sind und wie SDT diese unterstützt.

5.6 Semantische Unterstützung

Um dem Benutzer eine möglichst umfangreiche Unterstützung zu bieten, soll die IDE nicht nur die Darstellung des Programms verbessern, sondern auch ein Verständnis der Semantik des zu schreibenden Programms besitzen. Das

Ziel ist, dem Benutzer nicht nur syntaktische Hilfe über automatisierte Formatierungen und Anzeige von syntaktischen Fehlern zu geben, sondern auch semantische Unterstützung anzubieten. Zuerst ist daher eine Beschreibung der Semantik von Scheme- und CL-Programmen nötig.

5.6.1 Sichtbarkeitsbereich von Variablen

Fast jede Programmiersprache definiert Variablen und Sichtbarkeitsbereiche, in denen diese Variablen sicht- und nutzbar sind. Eine der grundlegenden Aufgaben einer IDE ist es daher, diese Sichtbarkeitsbereiche zu verstehen, dem Programmierer zu visualisieren und Fehler anzugeben, etwa bei der Benutzung von Variablen außerhalb ihres Sichtbarkeitsbereichs. In Scheme werden lokale Bindungen durch Lambda-Ausdrücke angelegt. Beispielsweise erzeugt der Ausdruck `(lambda (point) ...)` die Bindung einer Variable `point` innerhalb des Rumpfes [19]. Die Variable ist dabei nur innerhalb des Rumpfes des Lambda-Ausdrucks sichtbar, weshalb diese Bindungen auch *lokale Variablen* genannt werden. Die Bindung globaler Variablen erfolgt in Scheme mit dem Schlüsselwort `define`. Dabei ist zu beachten, dass wie in den meisten modernen Programmiersprachen Vorwärtsreferenzierung in Scheme 48 erlaubt ist, d. h. die Definition einer Variablen im globalen Sichtbarkeitsbereich muss nicht vor ihrer Benutzung stattfinden. Das hat zur Folge, dass ein einfacher Mechanismus wie das bloße Aufschreiben von geparsten Bindungen in eine Liste zum Notieren der gebunden Variablen nicht ausreicht, um die Sichtbarkeit von Variablen zu reflektieren. Es gibt zwei Arten, Sichtbarkeitsbereiche für Variablen mit der Möglichkeit der Vorwärtsreferenzierung zu realisieren:

1. **2-Pass**:

   Der Parser liest den Text zweimal: Beim ersten Mal schreibt er alle Bindungen auf, beim zweiten Mal überprüft er die Anwendung der Bindung. Diese Methode ist einfacher zu realisieren, aber bei großen Texten und häufiger Anwendung aufgrund von Performanzeinbußen unvorteilhaft.

2. **Notation gebundener und ungebundener Variablen**

   Der Parser liest den Text einmal und führt zwei Listen: Eine Liste speichert gebundene Variablen, die andere Vorkomnisse von ungebundenen Variablen. Liest der Parser eine Bindung, dann speichert er die Bindung der Variablen und löscht gleichzeitig aus der Liste der ungebundenen Variablen die mit dem selben Namen heraus. Am Ende bleiben so die Variablen übrig, für die der Parser keine Bindung gefunden hat. Diese Methode ist solange ebenfalls relativ einfach zu implementieren, wie die Auflösung der Referenzen nicht von anderen abhängt. Kann eine Bindung vom Parser beispielsweise nur dann aufgelöst werden, wenn der Ausdruck keine ungebundenen Variablen hat, muss der Parser auch notieren, wo

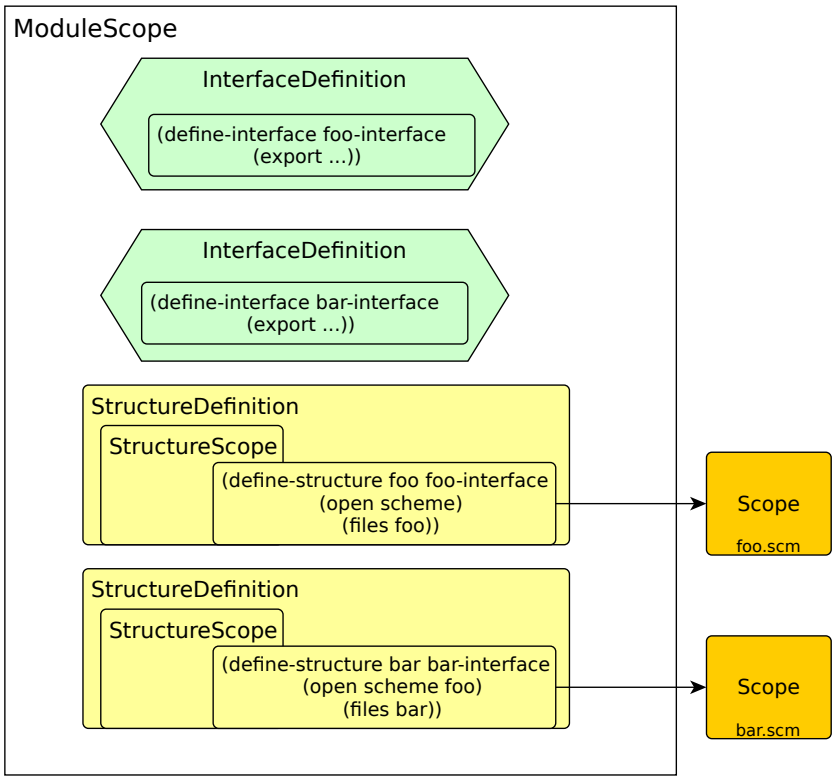


Abbildung 5.11: Aufbau des `ModuleScope` zum Verwalten der Bindungen in Moduldefinitionen.

diese Variablen auftreten und dementsprechend bei einer Bindung diese Referenzen ebenfalls auflösen.

Die IDE muss ein Parsen des Textes immer dann vornehmen, wenn der Inhalt der Datei nicht mehr mit dem aktuellen Modell übereinstimmt. Das passiert meistens dann, wenn der Benutzer etwas eintippt. Die IDE muss also in den Tipp-Pausen des Programmierers das Modell aktualisieren und darf für diesen Prozess nicht zu viel Zeit beanspruchen, da sonst der Programmierfluss gestört wird. Aus diesem Grund habe ich die Sichtbarkeitsbereiche in SDT mit der zweiten Methode implementiert, welche ich im Folgenden näher beschreibe.

5.6.2 Implementierung der Sichtbarkeitsbereiche

Die Sichtbarkeitsbereiche (engl. *Scopes*) im Scheme-Modus des SDT-Editors habe ich als Stapel implementiert. Scopes besitzen einen umschließenden Scope sowie eine Liste der gebundenen und ungebundenen Variablen. Betritt der Parser einen neuen Sichtbarkeitsbereich (etwa den Rumpf eines Lambda-Ausdrucks) so schiebt er einen neuen Scope auf den Stapel. Ist der Sichtbarkeitsbereich zu Ende, entfernt er den oberste Scope vom Stapel und stellt den

umschließende Scope wieder her. Eine Variable gilt genau dann im aktuellen Scope gebunden, wenn eine Definition entweder im selben Scope oder in einem der umschließenden Scopes erfolgte. Damit der Parser mit diesem System Vorwärtsreferenzen auflösen kann, muss er beim Entfernen des obersten Scopes vom Stapel beim Verlassen eines Sichtbarkeitsbereichs die ungebundenen Variablen in den umschließenden Scope kopieren. Folgt die Bindung einer ungebundenen Variable später im umschließenden Scope, löscht der Parser das Vorkommen der ungebundenen Variable im früheren Scope, der nach dem Verlassen im umschließenden Scope nicht mehr zugänglich ist. Folgendes Beispiel veranschaulicht das:

```
(define (faculty n)
  (if (equals? n 1)
      1
     (mult n (faculty (subtr n 1)))))
(define equals? =)
(define mult *)
(define subtr -)
```

Das Beispiel zeigt die Definition einer top-level Prozedur `faculty`. Beim Lesen der Prozedur ist der einzige Scope auf dem Stapel der globale Scope. Der Parser schiebt für die Prozedur einen neuen Scope auf den Stapel, in der das Benutzen der ungebundenen Variablen `equals?`, `mult` und `subtr` notiert wird. Ist der Parser mit dem lesen der Prozedurdefinition fertig, entfert er den Scope für die Prozedur wieder vom Stapel. Um später die ungebundenen Variablen im Programm sehen zu können, muss der globale Scope alle ungebundene Variablen am Ende enthalten. Dazu muss der vom Stapel entfernte Scope die Menge der ungebundenen Variablen in den umschließenden Scope (in diesem Fall den globalen Scope) kopieren. Später liest der Parser die Bindungen von den drei Variablen und entfernt diese aus der Menge der ungebundenen Variablen im globalen Scope, womit der Parser die vorherigen Vorwärtsreferenzen auflöst.

Für die Configuration Language ist ein anderes Vorgehen notwendig. CL-Code besteht nur aus top-level Moduldefinitionen, weshalb auch nur ein Scope notwendig ist, um den Sichtbarkeitsbereich von gebundenen Bezeichnern zu verwalten. Abbildung 5.11 zeigt den Aufbau dieses `ModuleScope`: Liest der Parser eine Schnittstellendefinition, speichert er ein Objekt der Klasse `InterfaceDefinition`. Dieses enthält neben dem Bezeichner auch die Modellierung der Schnittstelle. Kann der Parser zu dem Zeitpunkt der Definition die Modellierung nicht vollständig ermitteln, etwa weil eine Komponente noch nicht gebunden ist, speichert er das Objekt als zwar gelesen, aber noch nicht definiert ab, zusätzlich mit dem Hinweis, welche Komponente gefehlt hat. Erfolgt später eine Bindung der fehlenden Komponente, überprüft der `ModuleScope`, in welchen unvollständigen Definitionen diese Komponente

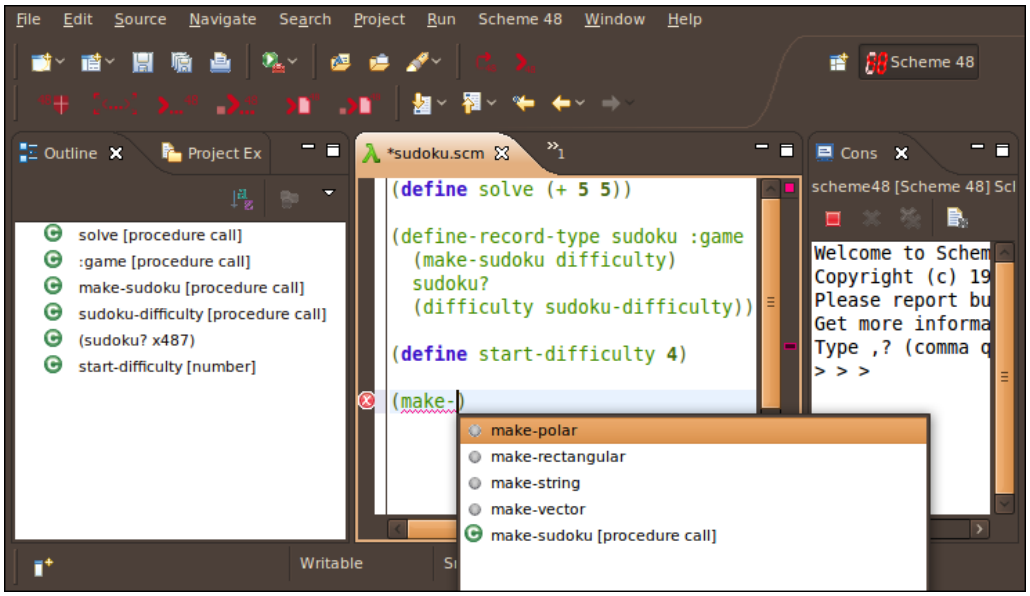


Abbildung 5.12: Die Bindungen des expandierten Makros sind in der Outline und Content Assist sichtbar.

vorkommt und versucht diese dann zu binden. Eine Moduldefinition, die eine Schnittstelle benutzt, deren Definition unbekannt ist, kann der Parser ebenfalls noch nicht binden und er verfährt damit auf die gleiche Weise.

5.6.3 Scheme-Makros

Eine Besonderheit von Scheme sind Makros, die Ausdrücke darstellen, die der Interpreter bei der Ausführung in anderen Programmcode expandiert. Dadurch kann der Programmierer neue Sprachkonstrukte anlegen, ohne die Sprachdefinition von Scheme selbst ändern zu müssen. Makros legen häufig Bindungen an, welche die IDE auch als Bindungen registrieren muss. Ansonsten kann die IDE viele Bindungen in Scheme-Programmen nicht auflösen und zeigt diese als ungebunden an. Um solche Bindungen zu erkennen, muss der Parser das Makro expandieren. Dazu ist die Anbindung an den Scheme-48-Prozess notwendig, welche die Bindung expandiert und damit den Code liefert, den der Parser nach Definitionen untersuchen kann.

Abbildung 5.12 zeigt die Benutzung eines Makros in SDT. `define-record-type` ist eine Prozedur von einem in Scheme 48 eingebautem Modul und der Interpreter hat bei der Übermittlung der Schnittstelle des Moduls die exportierte Variable mit `:syntax` markiert. Dieses Makro legt die in Scheme oft verwendeten *Records* an, welche zusammengesetzte Datentypen repräsentieren. Ein Record besteht aus einem Konstruktor, der das Record initialisiert, einem Typ, und Selektoren für Felder, welche die

in dem Record enthaltenen Daten zurückgeben. Der Parser erkennt an der Stelle `(define-record-type ... )`, dass der Aufruf eines Makros stattfindet und übergibt den gesamten Ausdruck zur Expansion an den Interpreter. Abbildung 5.13(a) zeigt den vom Interpreter zurückgebeben expandierten Ausdruck. Das Makro legt vier Bindungen an:

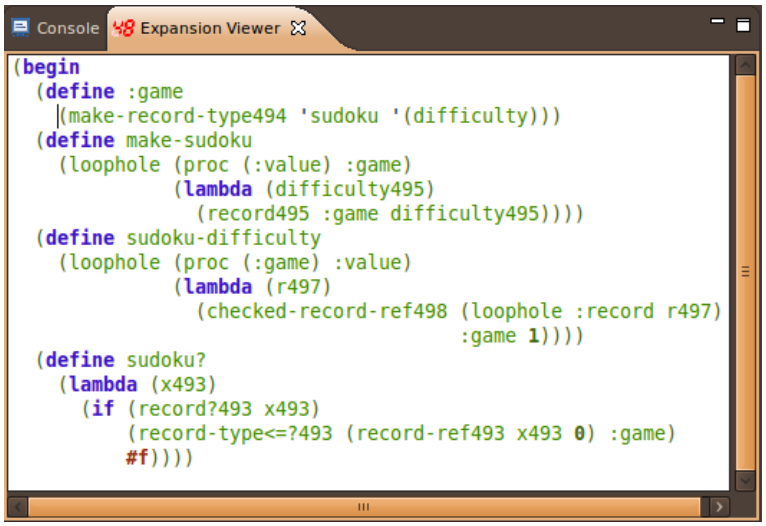
1. Den Typ des Records: `:game`
2. Den Konstruktor: `make-sudoku`
3. Das Prädikat: `sudoku?`
4. Einen Selektor: `sudoku-difficulty`

Diese Bindungen soll die IDE erkennen und in der Outline und Content Assist anzeigen. Dazu ruft der Parser, nachdem er das Makro erkannt und vom Interpreter hat expandieren lassen, eine neue Instanz des Parsers auf und übergibt diesem den expandierten Ausdruck. Die neue Instanz des Parsers liest den Ausdruck und gibt sowohl die gebundenen Variablen als auch einen AST zurück. Die alte Instanz bindet die gebundenen Variablen in den aktuellen Sichtbarkeitsbereich und fügt den zurückgegebenen AST des expandierten Makros anstelle des vom nicht-expandierten Makros erzeugten AST ein. In diesem neuen AST sind Knoten für die globalen Definitionen, welche die IDE dazu verwenden kann, die Bindungen in der Outline und Content Assist darzustellen.

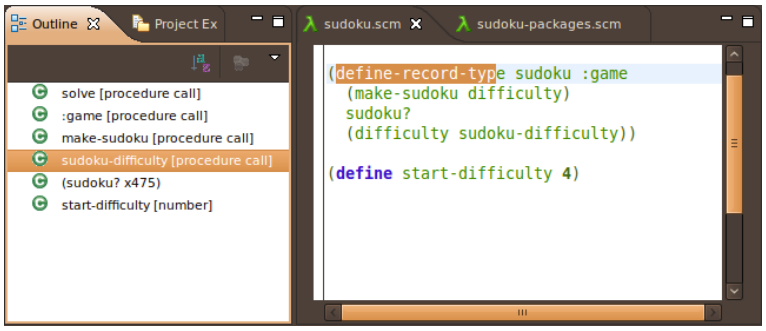
Damit der Interpreter den Ausdruck richtig expandieren kann, muss die Datei bereits samt Modul im Interpreter geladen sein. Daher muss der Benutzer zuerst die Datei über den Menü-Eintrag `Load file` oder dem entsprechenden Button in der Toolbar in den Interpreter laden. Anschließend kann der Interpreter der IDE die richtige Expansion des Makros mitteilen. Bei einem Doppelklick auf eine von dem Makro angelegeten Bindung in der Outline springt der Editor an die Stelle des Makros (Abbildung 5.13(b)). Die Bindungen des Records markiert die Outline als `procedure call`, weil die Bezeichner an Prozeduraufrufe binden (vgl. Abbildung 5.13(a)). Es ist der IDE hier nicht möglich, zu erkennen, dass eine Bindung einen Konstruktor oder ein Prädikat darstellt.

5.6.4 Bindungen in der Configuration Language

Jedes Modul in der Configuration Language verwaltet einen eigenen Sichtbarkeitsbereich, den `StructureScope`, der die importierten Bindungen aus anderen Strukturen und die gebundenen Variablen aus Implementierungen notiert. Bei der Ausführung eines Moduls durch den Interpreter muss dieser ebenfalls



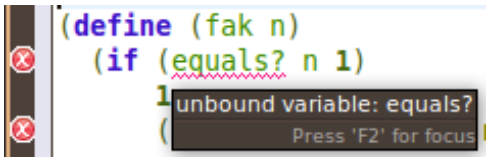
(a) Die Anzeige der expandierten Form im Expansion Viewer.



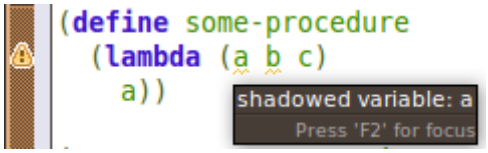
(b) Bei einem Doppelklick auf dem Knoten in der Outline springt der Editor an die Stelle des Makros.

Abbildung 5.13: Ein von der IDE expandiertes Makro in Scheme und dessen Einbindung in Expansion Viewer, Outline und Content Assist.

dessen Implementierung mit dem Modul als Umgebung ausführen. Das bedeutet, die importierten Bindungen aus anderen Strukturen im Modul sind auch in der Implementierung als top-level Definitionen sichtbar. Liest der CL-Parser im Modul eine Referenz auf eine Implementierungsdatei, startet dieser den Scheme-Parser auf diese Datei mit dem `StructureScope` als globalen Sichtbarkeitsbereich. Der Scheme-Parser liefert dem `StructureScope` Bindungen innerhalb der Implementierungsdatei, die er dazu benutzt, um die exportierten Bindungen in der Schnittstelle des Moduls zu überprüfen.



(a) Die Markierung einer ungebundenen Variable als Fehler.



(b) Die Markierung der Bindung einer bereits gebundenen Variablen als Warnung.

Abbildung 5.14: Annotationen im SDT-Editor im Scheme-Modus.

5.6.5 Anzeige der semantischen Unterstützung im SDT-Editor

Eclipse zeigt Annotationen im Quelltext durch direkte Unterstreichung der Stelle und Markierung der Zeile an den Seiten des Editors an. Abbildung 5.9 zeigt die Annotationen eines Programms im Scheme- und CL-Modus.

Die IDE weist den Benutzer auf folgende Fehler hin:

1. **Scheme-Modus**

   - **Ungebundene Variablen**
     Abbildung 5.14(a) zeigt die Annotation einer ungebundenen Variable als Fehler.
   - **Überschattete Variablen**
     Eine gelbe Warnung zeigt dem Benutzer an, dass eine Bindung eine bereits vorher gebundene Variable überschattet. In Abbildung 5.14(b) ist die Prozedur `some-procedure` aus dem Quelltext in Abbildung 5.9(b) dargestellt, welche die globale Variable `a` in ihrem Rumpf durch eine lokale Variable überschattet. Dadurch hat der Benutzer es einfacher, zwischen lokalen und globalen Variablen zu unterscheiden.

2. **CL-Modus**

   - **fehlende Bindungen exportierter Variablen**
     Jede Struktur deklariert in ihrer Schnittstelle eine Menge von Variablen, für die das Modul eine Implementierung bereitstellen muss. So

eine Implementierung kann einmal von anderen importierten Strukturen kommen, oder aber aus einer Bindung von einer Implementierungsdatei. Gibt es für eine exportierte Variable keine Implementierung, ist das ein Fehler, den Scheme 48 beim Evaluieren des Pakets bemerkt. SDT bemerkt diesen Fehler vor Auswertung des Moduls und stellt diesen grafisch dar (Abbildung 5.15(a)).

- **fehlende Module**

  Module können andere Module öffnen und damit deren exportierte Variablen innerhalb des Moduls sichtbar machen. Das gilt auch für die Implementierung des Moduls: Innerhalb der Scheme-Datei sind die importierten Variablen solange sichtbar, wie Auswertung der Datei im Kontext des Moduls erfolgt. Die geöffneten Module müssen dabei entweder in der selben Datei oder in einer bekannten Datei innerhalb des Projekts definiert sein, oder aber dem Interpreter von Scheme 48 bekannt sein. Öffnete ein Modul eine unbekannte Struktur, zeigt dies der SDT-Editor als Fehler an (Abbildung 5.15(b)).

- **Fehler in der Implementierungsdatei eines Moduls**

  Gibt ein Modul eine Implementierung an, startet die IDE den Parser für diese Datei im Kontext des Moduls, d. h. alle sichtbaren Variablen im Modul sind auch in der Scheme-Datei sichtbar. Enthält die Implementierung dennoch freie Variablen, zeigt der Editor einen Fehler an (Abbildung 5.15(c)).

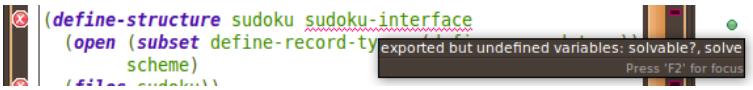
- **Fehlende Implementierungsdatei**

  Gibt ein Modul eine Implementierungsdatei an, ist diese aber nicht Dateisystem vorhanden, markiert der Editor die Stelle im Modul als Fehler (Abbildung 5.15(d)).

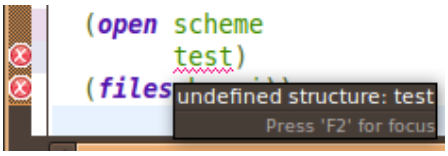
Damit die IDE die Semantik der Programme verstehen kann, ist ein Java-Modell der Scheme-48-Sprachelemente wie beispielsweise Strukturen und Schnittstellen notwendig. Im Folgenden beschreibe ich, welche Elemente ich in Java modelliert habe und welche Verwendung diese finden.

5.7 Modellierung von Scheme-48-Sprachelementen im Plugin

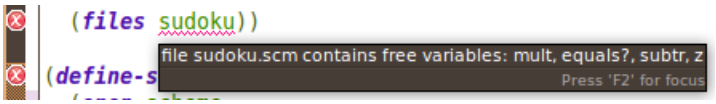
Um die Semantik eines Programms verstehen zu können, muss die IDE Teile des Codes interpretieren. Dies soll jedoch ohne Hilfe des Scheme-48-Prozesses geschehen, da sonst die Performanz sinken würde. Zur Interpretation erstellt der Parser dazu ein Klassenmodell in Java, dass die Struktur des Programmes



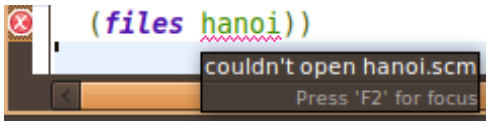
(a) Die Markierung einer nicht definierten, exportierten Variable als Fehler.



(b) Die Markierung einer ungebundenen Variablen als Fehler.



(c) Fehler in Scheme-Dateien werden ebenfalls in der Struktur als Fehler markiert.



(d) Markierung einer nicht gefundenen Datei als Fehler.

Abbildung 5.15: Annotationen im SDT-Editor im CL-Modus.

repräsentiert und die IDE weiter verarbeiten kann, um daraus Informationen zu beziehen und diese aufzubereiten, um dem Benutzer Unterstützung in Form von Fehlerannotationen oder Content Assist bereitzustellen. Folgende Komponenten modelliert die IDE:

5.7.1 Dateien

Das Bibliothekssystem braucht die Modellierung einer Datei und der darin enthaltenen Elemente, um diese auch in anderen Dateien verfügbar zu machen. Beispielsweise kann der Benutzer Schnittstellen in einer eigenen Datei definieren und diese Definitionen dann in einer anderen benutzen, um eine Moduldefinition schreiben.

Die Modellierung sortiert Dateien nach ihrem Inhalt: Dateien mit Scheme-Code bilden Scheme-Dateien, Dateien mit Moduldefinitionen CL-Dateien. Die IDE speichert diese Dateien in zwei separaten Bibliotheksklassen.

- Scheme-Dateien (`SchemeFile`) speichern den Pfad im Dateisystem sowie jeweils die Menge der gebundenen und ungebundenen Variablen.

- CL-Dateien (`CLFile`) bestehen aus dem Pfad im Dateisystem sowie den definierten Modulen und Schnittstellen.

5.7.2 Strukturen

Der Texteditor braucht ein Modell von den Strukturen aus Moduldefinitionen, um diese korrekt mit Fehlerannotationen versehen zu können und Importe zu simulieren. Strukturen stellen außerdem Ausführungsumgebungen für bestimmte Implementierungen dar, die der Parser als Information braucht.

Weiterhin muss die IDE Strukturen selbst erstellen können, etwa wenn der Benutzer externe Strukturen vom Interpreter einbindet oder eine mit dem `New-Module-Wizard` von SDT konstruiert.

Die Definition von Strukturen markiert Scheme 48 mit den Schlüsselwörtern `define-structure` und `define-structures`. Die EBNF-Grammatik für diese ist wie folgt beschrieben [20]:

```
⟨definition⟩ ⟶ ( define-structure ⟨name⟩ ⟨interface⟩ ⟨clause⟩* )
               | ( define-structures ( ( ⟨name⟩ ⟨interface⟩)* ) ⟨clause⟩* )
               | ...
⟨clause⟩ ⟶ ( open ⟨structure⟩ )
           | ( access ⟨name⟩* )
           | ( begin ⟨program⟩ )
           | ( files ⟨filespec⟩* )
           | ( optimize ⟨optimize-spec⟩ )
           | ( for-syntax ⟨clause⟩* )
```

Eine Struktur besteht also im Wesentlichen aus einer Folge von Anweisungen zur Veränderung der sichtbaren Bindungen innerhalb der Struktur und zur Angabe von Implementierungen. Demnach ist eine Struktur (`Scheme48Structure`) ein Wert aus:

- Dem Bezeichner, an den die Struktur gebunden ist
- Dem Pfad zu der CL-Datei, in dem die Struktur definiert ist
- Der Schnittstelle der Struktur
- Einer Menge von importierten Strukturen
- Einer Menge von Implementierungsdateien

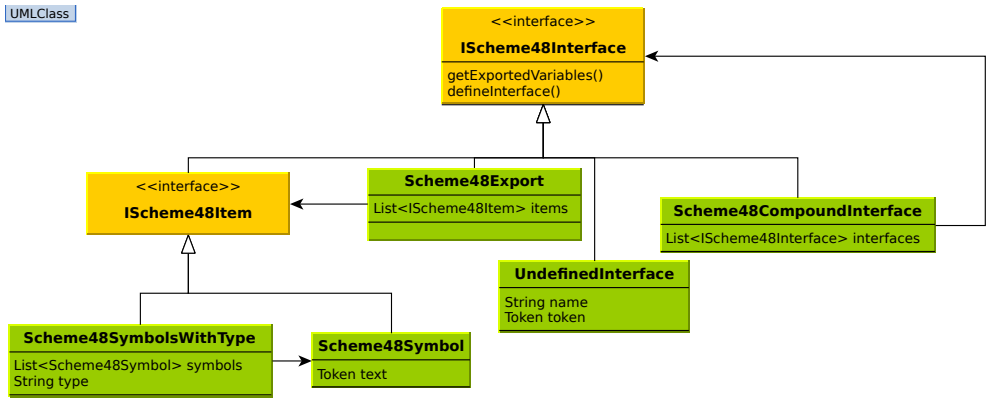


Abbildung 5.16: UML-Diagramm der Modellierung von Schnittstellen in SDT.

5.7.3 Schnittstellen

Schnittstellen sind separate logische Einheiten in Moduldefinitionen. Die IDE muss diese selbst dann modellieren, wenn sie anonym vorliegen, also nicht an einen Bezeichner gebunden sind. Die EBNF-Grammatik der Schnittstellen ist wie folgt definiert [20]:

```
⟨interface⟩ ⟶ ( export ⟨item⟩* )
              |  ⟨name⟩
              | ( compound-interface ⟨interface⟩* )
⟨item⟩ ⟶ ⟨name⟩
         | ( ⟨name⟩ type )
         | (( ⟨name⟩* ) type )
```

Grundlegende Einheit der Schnittstellen sind `Items`, welche die konkreten Bezeichner der Bindungen darstellen. Optional kann ein Item einen Typ in Form eines Bezeichners mit vorangestelltem Doppelpunkt (beispielsweise `:value`) haben. Allerdings ist es durch die dynamische Typisierung von Scheme dem Interpreter nicht möglich, bei Ausführung des Moduls die Typen zu überprüfen. Die Typangabe stellt so lediglich eine Hilfe für den Programmierer dar. Ausnahmen bilden hier Makros: Schnittstellen müssen Bindungen von Makros mit dem Typ `:syntax` deklarieren. Des Weiteren können Schnittstellen entweder aus einer Auflistung von Items bestehen, eine Vereinigung von anderen Schnittstellen sein oder einfach eine Referenz auf eine andere Schnittstelle darstellen.

Das in SDT verwendete Klassenmodell (Abbildung 5.16) für Schnittstellen ist direkt an die EBNF-Grammatik angelehnt. Eine Besonderheit ist das `UndefinedInterface`: Der Parser verwendet es als Platzhalter für Referenzen auf noch nicht definierte Schnittstellen. Später kann der Parser den Platzhalter durch Aufruf der Methode `defineInterface` durch eine Bindung ersetzen.

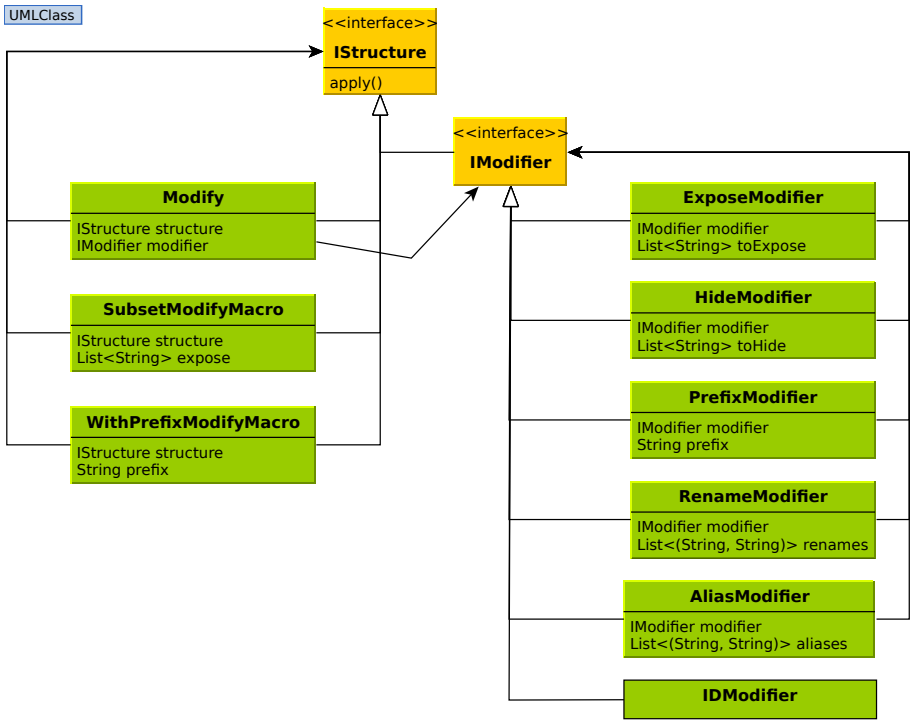


Abbildung 5.17: UML Diagramm der Modellierung von Modifier in SDT.

Dieser Mechanismus löst somit Vorwärtsreferenzen auf Schnittstellen auf. Weiterhin gibt es für Schnittstellen eine Methode `getExportedVariables`, die alle Bezeichner von exportierten Bindungen zurückgibt.

5.7.4 Modifier

Modifier sind Konstrukte der Configuration Language, welche die Sicht auf eine importierte Struktur verändern. Beispielsweise können Modifier explizit die zu importierenden Bindungen angeben oder diese umbenennen. Folgende EBNF-Grammatik definiert Modifier in CL [20]:

```
⟨clause⟩ ⟶ ( open ⟨structure⟩* ) | ...
⟨structure⟩ ⟶ ⟨name⟩
              | ( modify ⟨structure⟩ ⟨modifier⟩* )
              | ( subset ⟨structure⟩ ( ⟨name⟩* ))
              | ( with-prefix ⟨structure⟩ ⟨name⟩ )
⟨modifier⟩ ⟶ ( expose ⟨name⟩* )
             | ( hide ⟨name⟩* )
             | ( rename ( ⟨name⟩ ⟨name⟩ )* )
             | ( alias ( ⟨name⟩ ⟨name⟩ )* )
             | ( prefix ⟨name⟩ )
```

Die Angabe eines Modifiers bei der Importierung einer Struktur ist optional. Es gibt drei Arten von Modifier:

- `expose` und `hide` erzeugen Teilmengen von der Menge der importierten Bindungen,
- `alias` und `rename` erzeugen neue Bezeichner für Bindungen
- `prefix` bildet eine Menge von Bezeichnern auf eine neue Menge mit vorangestelltem Prefix ab.

`Subset` und `with-prefix` sind Makros, die in entsprechende Formen von `modify` expandieren.

Abbildung 5.17 zeigt das Klassendiagramm des in SDT implementierten Modells für Modifier. Das Modell stellt nicht nur die Struktur des Aufrufs dar, sondern kann auch die resultierende Sicht auf die Struktur berechnen. Alle Modifier implementieren das Interface `IStructure`, das der `structure`-Regel der EBNF-Grammatik und einer Anweisung zum Import einer Sturktur entspricht. Zu den implementierenden Klassen gehören `Modify`, `SubsetModifyMacro` und `WithPrefixMacro`. Diese entsprechenen den jeweilgen Optionen der `structure`-Regel. `Modify` benötigt die Angabe von Operatoren, die das Interface `IModifier` implementieren. Bei der Deklaration einer importierten Struktur kann der Programmierer beliebig viele Modifier angeben. Diese Modifier werten von rechts nach links aus, das bedeutet, dass der Interpreter bei der Ausführung zuerst den am weitesten rechts stehenden Modifier auf die Sicht der Struktur anwendet. Da der Parser den Ausdruck von links nach rechts liest, ist das Klassenmodell für die Modifier so geschachtelt, dass jeder Modifier einen unterliegenden Modifier hat. Aus diesem Grund ist ein Modifier nötig, der eine Sicht auf die Struktur nicht verändert und daher das Ende der Schachtelung markiert. Dieser ist der `IDModifier`, er gibt die Sicht unverändert zurück. Ein `IDModifier` ist immer nötig, selbst dann, wenn kein Modifier angegeben ist. Die Methode zur Berechnung der Sicht auf die Struktur wendet dann den innersten Modifier zuerst an. Folgendes Beispiel verdeutlicht das Modell:

Im Beispiel in Abbildung 5.18 importiert eine Struktur das Modul `foo`. Im Folgenden wird angenommen, dass `foo` unter anderem `faculty`, `make-foo`, `foo?` und `delete-foo` exportiert. Das Modul verändert die Sicht auf die Struktur mittels einer `modify`-Anweisung. Ein `Modify` besteht aus einer `IStructure` und einem `IModifier`, der die Sicht auf die `IStructure` modifiziert. Die `IStructure` in diesem Beispiel ist ein `SubsetModifyMacro`, das die Menge der importierten Bindungen des Moduls `foo` auf die vier angegebenen beschränkt. Der Ausdruck hat insgesamt drei Modifier, von denen der Interpreter zuerst das `expose` anwenden würde, als zweites das `rename` und schließlich

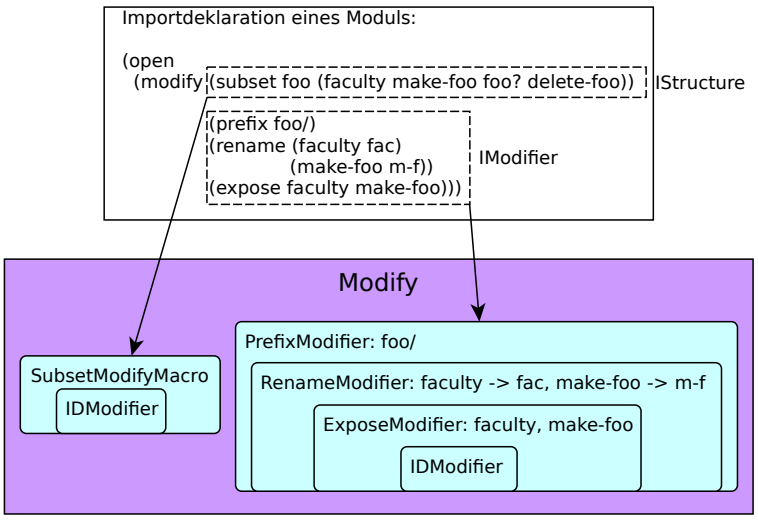


Abbildung 5.18: Ein Beispiel für die Übersetzung eines Modifiers in das Klassenmodell.

das `prefix`. In dem resultierenden Modell ist ein `PrefixModifier` der äußerste Modifier. Er enthält als unterliegenden Modifier einen `RenameModifier`, der einem `ExposeModifier` übergeordnet ist. Der unterste Modifier ist wieder ein `IDModifier`. Um die resultierende Sicht auf die Struktur zu berechnen, ,,wendet" die IDE zuerst die `IStructure` des Modifier an, indem sie die `apply`-Methode mit der Menge der exportierten Bindungen des Moduls `foo` als Argument aufruft. Folgender Ablauf entsteht:

1. Exportierte Bindungen von `foo`: {`faculty`, `make-foo`, `foo?`, `delete-foo`, ...}
2. `Modify` wendet das `SubsetModifyMacro` auf die Sicht der Struktur durch Aufruf dessen `apply`-Methode an und liefert: {`faculty`, `make-foo`, `foo?`, `delete-foo`}
3. `Modify` wendet den `IModifier` an, welcher rekursiv den unterliegenden Modifier anwendet, bis der `IDModifier` erreicht ist und die Sicht unverändert zurückgibt. Er liefert das Ergebnis: {`faculty`, `make-foo`, `foo?`, `delete-foo`}
4. Der `ExposeModifier` bekommt das Ergebnis und verändert die Sicht, in dem er alle Bindungen außer `faculty` und `make-foo` verdeckt. Die Sicht auf die Struktur nach dem `ExposeModifier` ist: {`faculty`, `make-foo`}
5. Der `RenameModifier` benennt die Bindungen um und liefert: {`fac`, `m-f`}
6. `Der PrefixModifier` stellt den Bindungen das Präfix ,,foo/" voran. Das Ergebnis ist:{`foo/fac`, `foo/m-f`}

Die resultierende Sicht auf die importierte Struktur enthält also die beiden Bindungen `foo/fac` und `foo/m-f`.

5.7.5 Bibliothekssystem

Wie bereits erwähnt sollen die Definitionen einer Datei auch in anderen Dateien verfügbar sein. Dazu benötigt die IDE ein Bibliothekssystem, dass für die Speicherung, Verwaltung und Synchronisation der bereits gelesenen Dateien verantwortlich ist. Dabei muss die IDE zum einen die vorher beschriebenen Scheme- und CL-Dateien als auch die Verlinkungen von Modulen und Implementierungen als *Default Packages* speichern. Dazu existieren drei Klassen:

- `SchemeLibrary`, die Bibliothek für Scheme-Dateien
- `CLLibrary`, die Bibliothek für CL-Dateien
- `DefaultPackageLibrary`. die Bibliothek für Zuweisungen von Modulen und Implementierungen

Im Folgenden beschreibe ich die Funktionsweise der Bibliotheken und die verwalteten Elemente näher.

- Dateien

  Liest der Parser eine Datei, so legt er neben dem abstakten Syntaxbaum und Informationen über den Aufbau noch ein Objekt vom Typ `SchemeFile` bzw. `CLFile` an, das die Datei repräsentiert, und teilt dieses der entsprechenden Bibliothek `SchemeLibrary` bzw. `CLLibrary` mit. Jede Bibliothek speichert die vom Parser produzierten `SchemeFile`- bzw. `CLFile`-Objekte und identifiziert diese über ihren Pfad im Dateisystem. Wird vom Parser eine neues Objekt mitgeteilt, so löscht die Bibliothek zunächst das eventuell vorhandene alte Objekt aus ihrem Bestand und speichert dann das neue ein. Weiterhin muss jede Bibliothek auch mit den auf der Festplatte vorhandenen Dateien synchron bleiben, welche die Objekte in ihren Beständen repräsentieren, damit sie nur die tatsächlich vorhandenen Dateien abbilden. Daher implementieren die Bibliotheken für Dateien das `IResourceChangeListener`-Interface von Eclipse. Dadurch erhalten diese bei Veränderungen im Workspace ein `ResourceDelta`, das die Veränderung repräsentiert. Veränderungen können das Hinzufügen, das Löschen, das Verschieben oder das Verändern einer Datei sein. Im Fall der Löschung einer Datei löscht das Bibliothekssystem das entsprechende Objekt aus ihren Beständen. Dadurch repräsentieren die Bibliotheken zu jedem Zeitpunkt die tatsächlich vorhandenen Dateien im Workspace. Optimistischerweise geht die IDE davon aus, dass Dateien

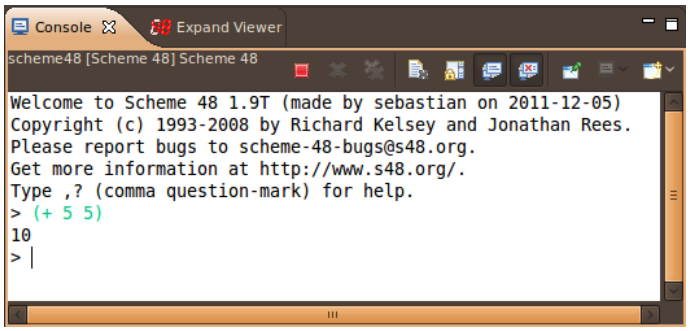


Abbildung 5.19: Der Console Viewer in Eclipse mit eingebundenem Scheme-48-Prozess.

nur innerhalb des Editors verändert werden, daher reagieren die Bibliotheken nur auf die Löschung, aber nicht auf die Veränderung einer Datei. Weiterhin ignoriert das Bibliothekssystem neu hinzugefügte Dateien, bis diese von der IDE mit dem Aufruf des Editors geparst werden. Beim Verschieben einer Datei aktualisieren die Bibliotheken den Pfad der verschobenen Datei.

- Default Packages von Scheme-Dateien

  Deklariert ein Modul eine Implementierungsdatei, so führt der Interpreter diese mit dem Modul als Umgebung aus. Um den Programmierer beim Bearbeiten der Scheme-Datei mit Fehlerannotationen und Content Assist zu unterstützen, muss die IDE wissen, welches Modul die Umgebung der Scheme-Datei darstellen soll. Der Benutzer kann in einer Scheme-Datei über das Kontext-Menü oder einem Button in der Toolbar einstellen, welches Modul, dass diese Datei als Implementierung angibt, die IDE als Default Package verwenden soll. Die `DefaultPackageLibrary` speichert diese Zugehörigkeiten der Implementierungen zu Modulen ab.

Die Bibliothek speichert dabei nicht nur Module von selbst definierten Dateien, sondern auch die vom Interpreter geladenen Module. Im Interpreter sind eine Menge von eingebauten Modulen zugänglich, die Implementierungen für häufig benötigte Funktionalität (beispielsweise Records oder Listenfunktionen) anbieten. Die IDE liest dabei nicht direkt die Moduldefinitionen des Interpreters, sondern nur die Informationen, die notwendig sind, um die eingebauten Module hinreichend zu beschreiben. Die Anbindung des Interpreters in der IDE und die Kommunikation mit dieser ist also von entscheidender Bedeutung für die Semantik und Modellierung der Programme. Als nächstes beschreibe ich daher konkret, wie die IDE den Interpreter in Eclipse einbindet und mit ihm kommuniziert.

5.8 Anbindung des Scheme-48-Prozesses in die IDE

Die Integration des Scheme-48-Prozesses in Eclipse ist sowohl zur Ausführung der geschriebenen Module und Implementierungen als auch zum Beziehen von Informationen über die Semantik eines Programms für die Unterstützung des Benutzers durch die IDE nutzbar. Normalerweise muss der Benutzer zur Ausführung der Programme lange Kommandozeilen-Befehle eingeben und die richtige Umgebung wählen. Die IDE kann diese Schritte abnehmen und durch einen Menü-Eintrag oder Button in der Toolbar ersetzen.

5.8.1 Prozessintegration in Eclipse

Eclipse führt Prozesse innerhalb eines eigenen *Launch Framework* aus, dass eine API zur Erstellung und Handhabung von Prozessen darstellt. Um einen Prozess zu erstellen, muss die IDE Eclipse mitteilen, mit welcher *Launch Configuration* es den Prozess ausführen soll. Eine Konfiguration stellt spezifische Informationen dar, mit der Eclipse einen Prozess starten soll, beispielsweise optionale Parameter oder die Umgebung. Konfigurationen fasst Eclipse in *Launch Configuration Types* zusammen, die den Typ des Prozesses beschreiben, den Eclipse mit einer Konfiguration gestart. Beispielsweise besitzt Eclipse JDT u. a. die Konfigurationstypen *Java Application* und *JUnit Tests*. Als Java Application kann Eclipse eine Klasse mit einer Main-Methode starten. Unter `Run As...` kann der Benutzer eine Konfiguration erstellen, die beispielsweise zusätzliche Argumente definiert, die Eclipse der Main-Methode übergibt.

Um einen externen Prozess starten und integrieren zu können, muss die IDE zunächst einen neuen Konfigurationstyp über den Einhängepunkt `org.eclipse.debug.core.launchConfigurationTypes` definieren. In dieser Extension gibt die IDE eine Klasse vom Typ `ILaunchConfigurationDelegate` an, die den eigentlichen Prozess über die Eclipse Runtime startet und die verwendete Konfiguration erhält. SDT unterscheidet keine verschiedenen Konfigurationen, sondern fordert lediglich die Angabe des Kommandozeilen-Befehls zum Starten des Scheme-48-Interpreters in den Einstellungen. Die Klasse `Scheme48Interpreter` enthält neben Methoden zur Interaktion mit dem Prozess auch den Prozess selbst und bietet Methoden zum Starten, Beenden und Neustarten des Prozesses. Der Benutzer kann über einen Menü-Eintrag oder Button in der Toolbar den Prozess starten lassen. Das Plugin ruft dann die `start`-Methode des `Scheme48Interpreter` auf, wodurch der Prozess mittels der `ILaunchConfigurationDelegate` von Eclipse gestartet wird. Eclipse erstellt automatisch eine Konsole (eine View mit integrierter Verbindung zum Prozess, der `Console Viewer`) her (Abbildung 5.19), in dem der Benutzer Befehle eingeben und das Resultat sehen kann.

5.8.2 Kommunikation mit dem eingebundenen Prozess

Das Schema der Kommunikation des Plugins mit dem Scheme-48-Intepreter ist in Abbildung 5.20 dargestellt. Die IDE kann über den `IStreamsProxy` des Prozesses Befehle an den Interpreter senden. Diese Befehle erscheinen nicht in der Konsole in Eclipse. Die Antwort des Interpreters erfolgt im `OutputStream-Monitor` des Prozesses, auf dem die Konsole als Listener eingetragen ist und somit die Antwort in der View anzeigt. Da die Kommunikation mit dem Prozess zur Abfrage von Informationen wie beispielsweise die exportierten Bindungen eines Moduls dem Benutzer verborgen bleiben soll, weil die Antworten des Interpreters oft umfangreich sind und den Programmierer verwirren würden, entfernt SDT die Konsole als Listener auf den `OutputStreamMonitor` immer, solange eine Anfrage auf den Interpreter läuft. Die IDE unterscheidet folgende Anfragen (*Queries*) an den Interpreter:

- `ExportQuery` übergibt dem Interpreter den Namen eines Moduls und erhält als Antwort die Menge der exportierten Bindungen des Moduls.
- `ExpandQuery` übergibt dem Interpreter einen Ausdruck und erhält als Antwort eine expandierte Form des Ausdrucks, falls dieser ein Makro war.
- `CurrentPackagesQuery` fordert vom Interpreter eine Liste mit allen gerade bekannten Modulen an. Dadurch kennt die IDE die eingebauten Module des Interpreters sowie deren exportierten Bindungen und kann diese in die Semantik von Programmen miteinbeziehen.
- `DefaultPackageQuery` übergibt dem Interpreter den Namen einer Datei und erhält als Antwort den Namen des der Datei zugewiesenem Moduls. Der Interpreter merkt sich beim Ausführen den Modulkontext einer Implementierungsdatei und legt diesen als Default Package fest. Beim Laden der Datei stellt der Interpreter automatisch das Default Package als Modulkontext der Datei ein. Beim erneuten Ausführen der Datei mit einem anderen Modulkontext ändert der Interpreter diesen entsprechend ab. Es wird also immer das zuletzt ausgewertete Modul als Default Package gespeichert. Dies könnte von der IDE benutzt werden um Implementierungen den Modulen zuzuordnen. In der aktuellen Version überlässt das Plugin die Zuordnung allerdings noch vollkommen dem Benutzer. Dieser benötigt nämlich die Zuweisung von Modulkontexte vor der eigentlichen Auswertung, um Implementierungsdateien mit Unterstützung der IDE bearbeiten zu können, wodurch diese Information obsolet ist.

Der Interpreter von Scheme 48 stellt standardmäßig keine Werkzeuge bereit, um die gewünschten Informationen wie die exportierten Bindungen eines Moduls oder die Auflistung der aktuell sichtbaren Module anzeigen

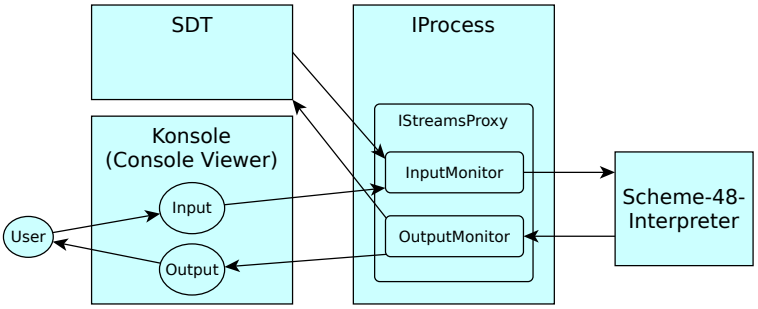


Abbildung 5.20: Schema der Kommunikation des Plugins mit dem Scheme-48-Interpreter.

zu lassen. Daher habe ich den Interpreter modifiziert, um entsprechende Kommandozeilen-Befehle verfügbar zu machen.

5.8.3 Modifikation des Scheme-48-Interpreters

Der Interpreter von Scheme 48 kann neben der Evaluierung von Scheme-Ausdrücken auch mit einem Komma beginnende Befehle entgegennehmen, welche die Umgebung des Interpreters verändern oder Informationen anzeigen. Die Befehle zur Interaktion mit der IDE extrahieren dabei lediglich Informationen aus dem Interpreter und bereiten diese auf eine Weise auf, welche die IDE leicht verarbeiten kann. Folgende Befehle habe ich dem Interpreter hinzugefügt:

- `,show-interface`

  Zeigt tabellarisch die Bezeichner und Typen der exportierten Bindungen eines Moduls an. Hat eine Bindung keinen Typ angegeben, gibt der Interpreter `:undeclared` als Typ aus.

- `,show-known-packages`

  Zeigt eine Auflistung aller in der aktuellen Umgebung bekannten Module. Dazu gehören sowohl die eingebauten Module des Interpreters als auch die in den Interpreter geladenen Module.

- `,show-default-package`

  Zeigt das Modul an, das standardmäßig als Umgebung für die angebene Implementierungsdatei definiert ist.

Mit diesen Befehlen ist es möglich, in der IDE entsprechend mit dem Interpreter zu kommunizieren. Importiert der Benutzer in einer Struktur ein Modul, so fragt der Parser den Interpreter, ob das Modul bekannt ist und wenn ja, welche Bindungen es exportiert. Diese Information genügt der IDE, um eine

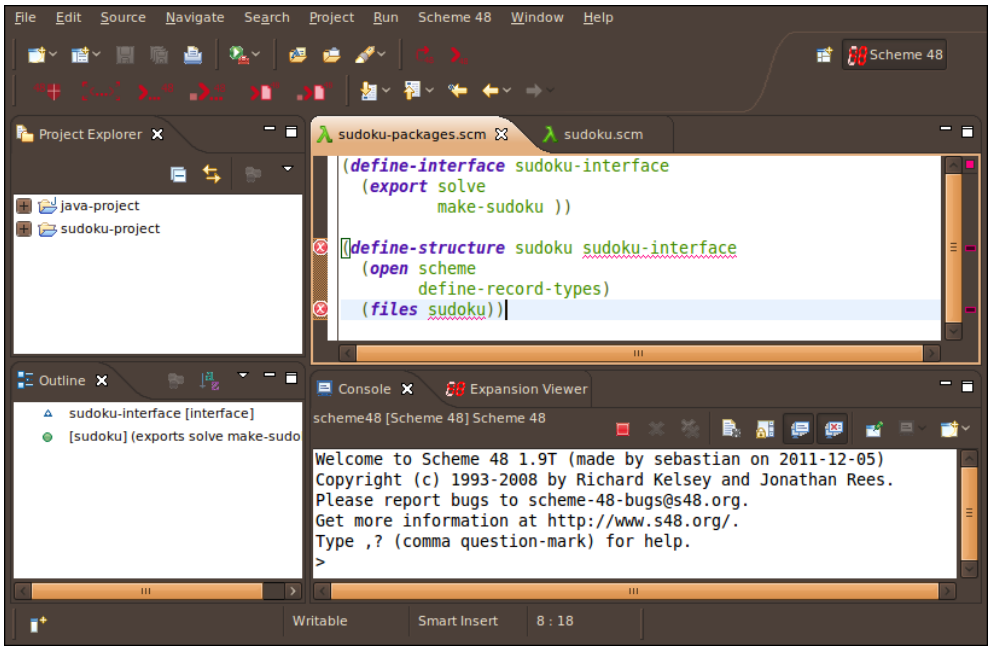


Abbildung 5.21: Die Benutzeroberfläche von SDT.

`Scheme48Structure` in der Bibliothek zu speichern, welche den Namen und die exportierten Bindungen des abgefragten Moduls repräsentiert und so die IDE bei erneutem Gebrauch nicht nochmal mit dem Interpreter kommunizieren muss. Dieser Vorgang bleibt dem Benutzer verborgen, da die IDE die Antwort des Interpreters vor der Konsole versteckt.

Im Folgenden beschreibe ich die andere für den Benutzer sichtbaren Komponenten von der SDT-Benutzeroberfläche, mit denen er hauptsächlich interagiert.

5.9 Andere UI-Komponenten

Die gesamte Benutzeroberfläche von SDT ist in Abbildung 5.21 dargestellt. Die voreingestellte Perspektive besteht aus dem Project Explorer von Eclipse und der Outline auf der linken Seite sowie der Konsole und dem Expansion Viewer am unteren Rand. Der Editor befindet sich in der Mitte. Die Toolbar enthält einige Buttons, die das Plugin beisteuert, und unter `File > New` oder Rechtsklick im Project Explorer sind der `New-Project-` und `New-Module-Wizard` von SDT zu finden.

5.9.1 Expansion Viewer

Da in Scheme der Programmierer häufig Makros verwendet und erstellt, kann es hilfreich sein, die expandierte Form eines Makros anzeigen zu lassen. Im SDT-Editor kann der Benutzer dazu einen selektierten Ausdruck über das Kontext-Menü oder einem Button in der Toolbar von der IDE in expandierter Form anzeigen lassen, falls dieser ein Makro war. Die IDE frägt die expandierte Form direkt vom Interpreter über den Befehl `,expand <exp>` ab. SDT stellt dann die Antwort im Expansion Viewer dar. Wie alle Views in Eclipse leitet der Expansion Viewer die Klasse `ViewPart` ab, welche Standardimplementierungen für Views bereitstellt. Der Expansion Viewer visualisiert den expandierten Ausdruck genauso wie der Editor über einen `SourceViewer`, der den Inhalt mittels einer `SourceViewerConfiguration` einfärbt.

5.9.2 Actions

Actions stellen Aktionen dar, die der Benutzer betätigen kann, wie beispielsweise das Drücken eines Buttons, und erscheinen zum leichten und intuitiven Finden in Form von Menü-Einträgen oder Buttons in der Toolbar. Wo und in welcher Form Eclipse die Actions darstellt, ist in der `plugin.xml` über Einhängepunkte definiert. Folgende Actions sind verfügbar, sobald die Scheme-48-Perspektive geöffnet ist:

- `Start Interpreter`

  Startet den Interpreter mit der Konfiguration, die in den Einstellungen vom Benutzer angegeben ist und welche den Namen des Interpreters, den Kommandozeilen-Befehl zum Ausführen und den Ordner, in dem Eclipse den Prozess starten soll, enthält und zeigt die Konsole an. Falls nicht bereits vorhanden, erstellt Eclipse eine Konsole, die mit dem Prozess verbunden ist.

- `Restart Interpreter`

  Terminiert den Scheme-48-Prozess und startet ihn wieder. Die IDE löscht dabei auch alle vom Interpreter geladenen Module aus ihrem Bibliotheksystem.

Neben den Actions, die standardmäßig in der Scheme-48-Perspektive vorhanden sind, gibt es folgende Actions, die nur mit geöffnetem SDT-Editor in der Toolbar und dem Menü sichtbar sind:

- `Set default package`

Legt den Modulkontext einer Implementierungsdatei fest. Der Benutzer kann zwischen Modulen wählen, die diese Datei als Implementierung angeben. Die Information wird in der `DefaultPackageLibrary` gespeichert und vom Parser geladen, um die Bindungen verfügbar zu machen, die in der Implementierungsdatei mit dem gewählten Modul als Kontext sichtbar sind.

- `Expand expression`

  Schickt den selektierten Text im Editor mit dem Befehl `,expand` an den Interpreter, der die Makros im Text expandiert, und stellt das Ergebnis im Expansion Viewer dar.

- `Evaluate expression`

  Lässt den im Editor selektierten Text vom Interpreter evaluieren. Die Action veranlasst den Interpreter zur Evaluation des Ausdrucks und die Konsole stellt das Ergebnis dar. Das Plugin schickt dabei den Befehl `,from-file`. Dieser gibt die Datei an, in welcher der auszuwertende Ausdruck definiert ist und der Interpreter kann den Modulkontext ermitteln, in dem er der Ausdruck evaluieren soll.

- `Evaluate expression in module context`

  Lässt den im Editor selektierten Text im Kontext eines vom Benutzer wählbaren Moduls evaluieren. Der vom Plugin geschickte Befehl ist `,in <module> ,from-file`. Dieser führt die Auswertung im Kontext des angegebenen Moduls aus.

- `Load file`

  Lädt die aktuelle Datei mit dem Befehl`,load` in den Interpreter. Dieser Befehl führt alle Ausdrücke in der angegebenen Datei im Interpreter in dessen aktuellem Kontext aus.

- `Load file in module context`

  Lädt die aktuelle Datei mit dem Befehl `,in <module> ,load` mit einem vom Benutzer gewähltem Modul in den Interpreter. Dieser Befehl führt alle Ausdrücke innerhalb der angegebenen Datei im Kontext des angegebenen Moduls aus.

Jede Action ist eine Klasse, die das Interface `IAction` von JFace implementiert. Die Methode `run` enthält den Code, der bei Aktivierung der Action (beispielsweise wenn der Benutzer den Menü-Eintrag auswählt) vom Plugin ausgeführt wird. Um die Actions in die Toolbar zu intergrieren, ist eine Extension von `org.eclipse.ui.editorActions` und für jede Action eine Klasse, die das Interface `IEditorActionDelegate` implementiert, notwendig. Die

Delegate-Klasse wird von Eclipse benötigt, um die entsprechende Action zu aktivieren.

5.9.3 Code Templates

Code Templates sind vorgefertigte Code-Blöcke, in denen der Programmierer nur bestimmte Felder ausfüllt. Sie sind mit Formularen zu vergleichen, die an einen kurzen Bezeichner gebunden werden. Ein bekanntes Beispiel für ein Code Template ist `sysout` in Java, welches die IDE mit `Ctrl + Space` in `System.out.println();` verwandelt und den Cursor innerhalb der Klammern platziert. SDT benutzt das System von DLTK, um benutzerdefinierte Code Templates zur Verfügung zu stellen.

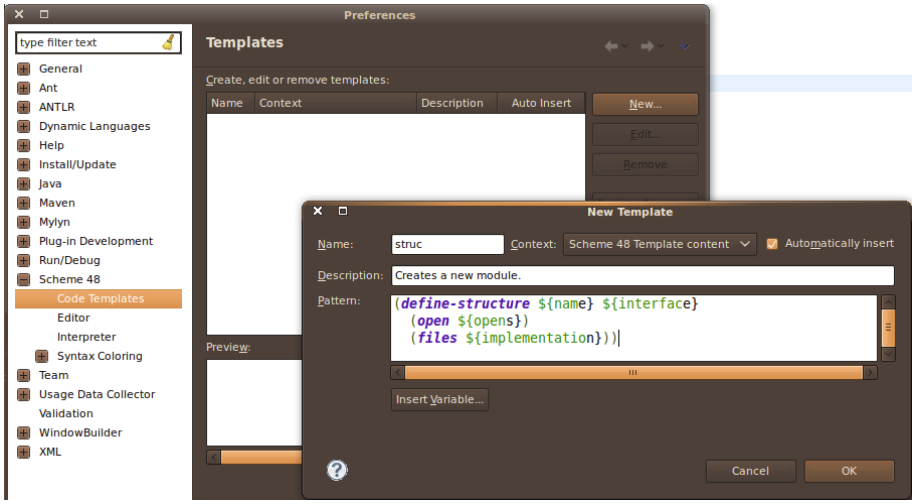


Abbildung 5.22: Definition eines Code-Templates in den Einstellungen.

Abbildung 5.22 zeigt die Definition eines Templates in den Einstellungen. Der Name des Templates ist gleichzeitig die Abkürzung, unter welcher der Benutzer das Template später aufrufen kann, in diesem Fall `struc`. Dieses Template soll die Konstruktion eines Moduls vereinfachen, indem es die Syntax vordefiniert und Formularfelder bereitstellt, die der Benutzer nur noch ausfüllen muss. Ein Modul hat in seiner gängigsten Form einen Namen, eine Schnittstelle, eine Reihe von importierten Modulen und eine Implementierungsdatei. Formularfelder für Templates sind von der Form `${Feld}` . Die IDE stellt die Felder nach Einfügen des Templates in den Quelltext umrandet dar und der Benutzer kann zwischen ihnen mit der Tabulator-Taste springen. Der Aufruf eines Templates erfolgt entweder durch Drücken der Tastenkombination `Ctrl + Space` oder durch Auswahl im Content Assist Fenster (Abbildung 5.23). Wählt der Benutzer das Template aus der Liste, ersetzt die IDE den Bezeichner mit der Definition des Templates und der Benutzer muss anschließend nur die vorher definierten Felder des Templates ausfüllen (Abbildung 5.24).

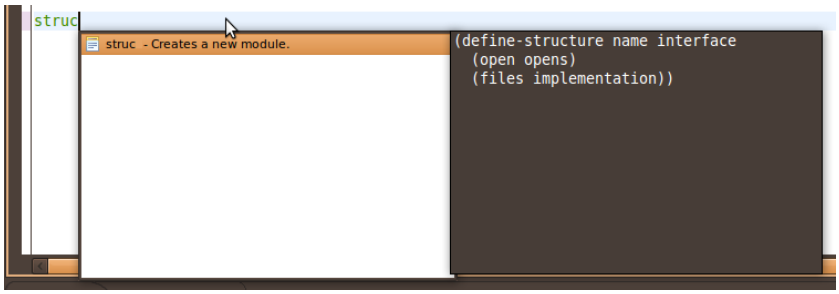


Abbildung 5.23: Das Code Template erscheint als Auswahlmöglichkeit in der Content Assist.

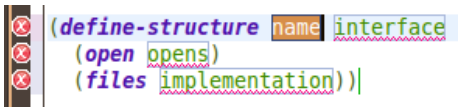


Abbildung 5.24: Eingefügtes Template mit den auszufüllenden Formularfeldern.

5.9.4 Wizards

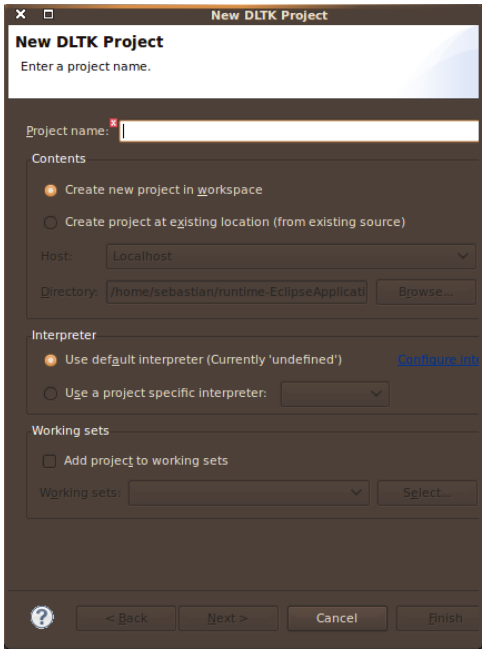
Wizards sind ein- oder mehrseitige Dialoge, in denen der Benutzer Daten auswählen und eingeben kann, die der Wizard dann beispielsweise benutzt, um komplex Elemente automatisch zu erstellen. SDT bietet zwei Wizards zur einfachen Erstellung von Projekten und Modulen an:

- `New-Project-Wizard`

  Erstellt ein neues SDT-Projekt. Der Wizard (Abbildung 5.25(a)) ist der von DLTK standardmäßig bereitgestellte Wizard für neue Projekte und hat daher noch die Optionen `Contents`, `Interpreter` und `Working Sets`. Die letzten beiden Einstellungen können beim Erstellen eines neuen SDT-Projekts ignoriert werden. In `Contents` kann der Benutzer auswählen, ob er ein neues Projekt anlegen oder ein bereits existierendes importieren möchte.

- `New-Module-Wizard`

  Erleichtert die Erstellung eines neuen Moduls, ähnlich zu dem `New-Class-Wizard` in JDT, der eine neue Java Klasse anlegt. Abbildung 5.25(b) zeigt die Benutzeroberfläche des Wizards. Der Benutzer kann einen Namen für das neue Modul eingeben, den Namen der Datei, welche die IDE erstellen soll, und eine Implementierungsdatei für das Modul. Für die Schnittstelle des neuen Moduls kann der Benutzer über den Button `Add` bereits definierte Schnittstellen anderer Dateien auswählen oder mit `New` eigene Schnittstellen über einen Dialog anlegen, welche die IDE ebenfalls in der neuen Datei definiert. Die Strukturen,



(a) Der New-Project-Wizard von SDT.

(b) Der New-Module-Wizard von SDT.

Abbildung 5.25: Die Wizards von SDT.

die das neue Modul importieren soll, sind in der `Opens` Liste aufgeführt. Über `Add` kann der Programmierer Module anderer Dateien angeben, oder über `Add external` die bekannten Module des Interpreters abrufen. Außerdem ist es mit `Modify` möglich, für jedes importierte Modul über einen Dialog einen Modifier festzulegen, der die Sicht auf die importierte Struktur verändert. Nachdem der Benutzer fertig ist, legt der Wizard eine neue Datei für das Modul an und erstellt entsprechende Definitionen für Schnittstellen sowie für das Modul. Existiert die angebene Implementierungsdatei noch nicht, erstellt der Wizard diese Datei. Anschließend öffnet das Plugin beide Dateien im SDT-Editor.

5.9.5 Content Assist

Die Content Assist in SDT zeigt die in einer Datei sichtbaren top-level Bindungen an. Das sind einerseits Schlüsselwörter von Scheme oder CL, andererseits definierte Bindungen. Um die Content Assist von DLTK zu nutzen, muss die IDE zwei Einhängepunkte nutzen: `org.eclipse.dltk.core.completionEngine` und `org.eclipse.dltk.ui.scriptCompletionProposalComputer`. Die eigentliche Berechnung der Vorschläge für die Content Assist findet in der Klasse statt, die das Interface `ICompletionEninge` von DLTK implementiert und

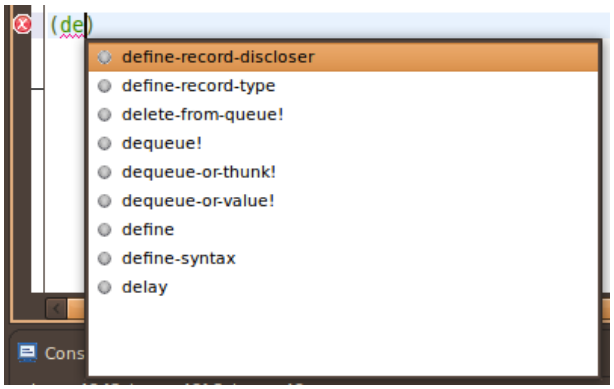


Abbildung 5.26: Content Assist in einer Scheme-Datei. Welche Bindungen sichtbar sind, hängt von dem Default Package der Datei ab.

die das Plugin in den ersten der beiden genannten Einhängepunkte einhängt. Das Interface fordert die Implementierung der Methode `complete`, welche den Inhalt und AST der Datei sowie die Position des Zeigers im Text bekommt und Vorschläge (`Proposals`) an einen `CompletionRequestor` weitergibt, der diese in der eigentlichen Content Assist anzeigt.

SDT bietet für Scheme- und CL-Dateien verschiedene Content Assist an.

- **Scheme-Dateien**

  Die Content Assist zeigt die in der Datei sichtbaren top-level Bindungen sowie die Schlüsselwörter des R^{5}RS an. Die in der Datei selbst definierten top-level Bindungen ermittelt die `CompletionEngine` über den AST vom Parser. Weiterhin kann die Datei als Implementierung von einem Modul definiert sein, womit ebenfalls die importierten Bindungen anderer Module sichtbar sind. Dazu liest die `CompletionEngine` die Verlinkungen in der `DefaultPackageLibrary` aus und sucht das entsprechende Modul, um dessen importierte Bindungen zu erhalten. Abbildung 5.26 zeigt die Content Assist in einer Datei für den Präfix ,,de" an, die als Implementierung eines Moduls markiert ist, das die Module `define-record-types` und `queues` importiert.

- **CL-Dateien**

  Für CL-Dateien sind neben den Schlüsselwörtern und Bindungen in der selben Datei vor allem die Module und Schnittstellen wichtig, die in anderen Dateien definiert sind. Dazu liest die `Completion` den Inhalt der `CLLibrary` aus und kreiert für jeden Eintrag einen Vorschlag. Die Content Assist zeigt also die komplette Umgebung des Moduls an und welche anderen Module es importieren kann.

Zusätzlich dazu enthalten die Vorschläge der Content Assist auch die vorher beschriebenen Code Templates.

Kapitel 6

Fazit

Mein Ziel beim Entwurf von *Scheme 48 Development Tools* war es, ein Werkzeug zu schaffen, mit dem Scheme 48 für Programmierer, die Entwicklungsumgebungen wie Visual Studio oder Eclipse gewöhnt sind, zugänglicher wird, und mit dem erfahrenen Scheme-48-Programmierern eine Alternative zu GNU Emacs mit erweiterter Funktionalität zur Verfügung steht.

Die fertige IDE steht unter

`http://www-pu.informatik.uni-tuebingen.de/sdt/`

zum Download bereit, zusammen mit Installationsanweisungen, einer Anleitung und diversen anderen Informationen.

In diesem Kapitel ziehe ich ein Resümee über das Erreichte, erläutere die Schwierigkeiten und Schwachpunkte, auf welche Weise diese verbessert oder beseitigt werden können und welche Komponenten SDT noch fehlen.

6.1 Zusammenfassung und Bewertung des Erreichten

SDT ist ein Plugin für Eclipse Indigo und als solches nahtlos in die Plattform integriert. Es lässt sich direkt über Eclipse durch den Update Manager installieren und benötigt zudem eine aktuelle Installation von Scheme 48.

Die konkreten Fähigkeiten des Plugins:

- **Unterstützung des R^{5}RS und der CL**

  SDT ist eine IDE sowohl zur Erstellung und Bearbeitung von Moduldefinitionen als auch zur Entwicklung der Implementierungen von Modulen im R^{5}RS.

- **Mächtiger Texteditor**

  Der Texteditor bietet für Scheme und CL separat einstellbare Syntaxhervorhebungen sowie Formatierungshilfen zum flüssigen Erstellen von Programmen mit Scheme-Syntax.

- **Fehlerannotation**

  Die IDE erkennt und markiert syntaktische und semantische Fehler in Programmen vor der Ausführung. Dazu gehört vor allem die Anzeige von freien Variablen und die Durchsetzung von Exporten in Schnittstellen.

- **Outline**

  Die Outline bietet eine Auflistung der top-level Definitionen eines Programms.

- **Intelligente Suche**

  Mit der Suche ist das Finden von top-level Definitionen in allen bekannten Scheme- und CL-Dateien möglich.

- **Selbst-definierbare Code-Templates**

  Der Programmierer kann in den Einstellungen eigene Code-Templates definieren und somit Schreibarbeit sparen.

- **Content Assist**

  Die Content Assist listet die global sichtbaren Bindungen und Schlüsselwörter auf. Angefangene Wörter kann die IDE automatisch vervollständigen.

- **Wizards**

  Mit den Wizards lassen sich neue Scheme-48-Projekte sowie Module samt Implementierung komfortabel erstellen.

- **Unterstützung von Makros**

  Makros lassen sich per Knopfdruck in einem separaten Expansion Viewer mit Syntaxhervorhebung darstellen und die IDE expandiert automatisch bekannte Makros im Scheme-Code und erfasst so innerhalb dieser Makros angelegte Bindungen.

- **Komfortable Ausführungsmöglichkeiten**

  Es ist die Evaluierung einzelner Code-Teile als auch gesamter Dateien per Knopfdruck im Interpreter möglich, auch im Kontext auswählbarer Module.

- **Integrierter Scheme-48-Prozess und REPL**

  Der Scheme-48-Prozess ist direkt mit der IDE verbunden und dem Benutzer ist die Interaktion mit der REPL über die Eclipse-Konsole möglich.

Scheme 48 Development Tools stellt also eine gute Alternative zu GNU Emacs dar, das im Gegensatz zu SDT eine lange Einarbeitungszeit und Zusammenstellung der notwendigen Komponenten benötigt. Es bietet also eine schnelle und umfangreiche *All-In-One*-Lösung zum Einstieg und zur fortgeschrittenen Programmierung mit Scheme 48.

Des Weiteren bestehen umfangreiche Möglichkeiten, die IDE zu verbessern. Im nächsten Abschnitt erläutere ich die vorhandenen Schwachpunkte von SDT und wie man diese angehen kann.

6.2 Verbesserungsmöglichkeiten

Die Schwachpunkte von SDT basieren meist auf zwei Dingen: Der Unfähigkeit des Parsers, aus Fehlerzuständen zurückzukehren und das Fehlen eines eigenen, kompletten Scheme- und CL-ASTs anstelle der Verwendung eines auf den JDT-AST abgebildeten. Sollten Verbesserungen an SDT vorgenommen werden, so schlage ich vor, diesen beiden Dingen die höchste Priorität einzuräumen. Im Folgenden erläutere ich, warum diese Schwachpunkte darstellen, welche Folgen sie haben und wie man sie beheben kann.

6.2.1 Rückkehr des Parsers aus Fehlerzuständen

Die IDE zeigt zwar syntaktische Fehler an, jedoch zwingen diese den Parser meist in einen Fehlerzustand, aus dem er nicht mehr herauskommt. Das bedeutet, ein Syntaxfehler am Anfang einer Datei führt dazu, dass der Parser den Rest der Datei nicht mehr erkennen und verarbeiten kann. In JDT von Eclipse ist ein Fehler in einem Teil des Codes nicht weiter von Belang, der Editor markiert die Stelle und der Parser liest weiter als ob der Fehler nicht vorhanden wäre. In SDT hingegen führt beispielsweise die falsche Klammerung eines Ausdrucks dazu, dass alles, was danach kommt, nicht mehr vom Parser erkannt werden kann.

Der Grund ist, dass der von ANTLR generierte Parser schlecht aus Fehlerzuständen zurückkehren kann, wenn nicht bekannt ist, wieviele Tokens auf einen Ausdruck folgen sollen. In Scheme-Syntax können Ausdrücke an vielen Stellen beliebig oft vorkommen, was es für ANTLR zu schwierig macht, eine automatische Rückkehr aus Fehlerzuständen zuverlässig zu generieren. Allerdings kann man in ANTLR für jede Regel separat angegeben, was der Parser im Falle eines Fehlerzustands machen soll. Durch die Implementierung eines intelligenten Algorithmus zur zuverlässigen Rückkehr aus Fehlerzuständen wäre

die IDE generell robuster, da nicht jeder Syntaxfehler zu einem Totalabsturz des Parsers führen würde.

Eine intelligente Rückker aus Fehlerzuständen hätte auch Folgen auf andere Funktionen der IDE wie die Content Assist. Diese hängt nämlich entscheidend vom aufgebauten AST ab. Wenn der Quelltext einen Syntaxfehler enthält und der Parser nicht mehr aus dem Fehlerzustand zurückkehren kann, ist auch die weitere Generierung des ASTs verhindert. Da sich der Quellcode während des Editierens durch den Programmierer ständig in einem fehlerhaften oder zumindest unvollständigen Zustand befindet, kann die Conten Assist ohne robusten Parser nicht zuverlässig funktionieren.

6.2.2 AST

Der derzeit in SDT verwendete AST zur Darstellung von Scheme- und CL-Programmen kann nur wenige Informationen über die Struktur eines Programms speichern, da nur top-level Bindungen im AST erscheinen. Der Grund für diese Vereinfachung war die Abbildung auf einen JDT-AST, um die von DLTK angebotene Funktionalität bezüglich Outline und Suche nutzen zu können. Der Informationsverlust hat u. a. Auswirkungen auf folgende Bereiche:

1. **Suche**

   Da die Suche über den AST operiert und der AST nur top-level Bindungen und keine Referenzen speichert, ist es nicht möglich, über die Suche die Benutzung von Variablen anzeigen zu lassen. Das ist aber zum Beispiel dann sehr nützlich, wenn der Programmierer eine Deklaration ändert und alle Referenzen auf diese Variable ändern muss.

2. **Content Assist**

   Die Content Assist von SDT ist nicht- oder nur semi-kontextual. Sie zeigt die in einer Datei global sichtbaren Bindungen und Schlüsselwörter an. Kontextual ist daran allein, dass die IDE Module und Implementierungen zuordnen kann und dann die Bindungen in der Implementierung anzeigt, die das Modul von anderen Strukturen importiert. Das allein ist zwar schon eine sehr wichtige Information, wenn ein Programmierer in einem bestimmten Modulkontext arbeitet, allerdings sind weiterführende kontextuale Hilfen nicht möglich. Beispielsweise ist es nicht möglich, innerhalb einer Prozedur die von einem Lambda-Ausdruck angelegten lokalen Variablen anzeigen zu lassen. Das liegt daran, dass es für die IDE keine einfache Möglichkeit gibt, den Kontext für eine beliebige Stelle des Quelltextes zu berechnen. Mit einem kompletten AST wäre das einfacher, da die IDE nur die entsprechenden Stelle im AST nachschlagen müsste und damit den kompletten Kontext der Stelle kennen würde.

3. **Refactoring**

   Refactoring im Sinne von struktureller Verbesserung eines Quelltextes bei Erhalt der Semantik ist nur dann möglich, wenn alle Informationen des Quelltextes im AST repräsentiert sind und somit eine Äquivalenzumformung zwischen dem Quelltext und dem AST möglich ist. Refactoring findet direkt auf dem AST statt und ordnet diesen neu oder entfernt und fügt Knoten hinzu, weshalb die IDE diese Änderungen im AST in Quelltextänderungen umwandeln muss.

Daher ist die Implementierung eines vollständigen AST für Scheme und CL notwendig. Ein vollständiger AST ist heterogen, d. h. nicht alle Knoten sind vom selben Typ, und idealerweise repräsentiert jede Parserregel einen eigenen Knotentyp im resultierenden AST. Daher ist es notwendig, von der Abbildung auf einen JDT-AST abzuweichen und die Nutzung der Funktionalität von DLTK an dieser Stelle aufzugeben und eine eigene Outline und Suche basierend auf dem Scheme- und CL-AST zu implementieren.

6.2.3 Content Assist

Eine Verbesserung der Content Assist ist nicht nur über die Anzeige von lokalen Variablen möglich. Generell umfasst die Content Assist nicht nur die Vorschläge, welche die IDE während des Programmierens anzeigt. In JDT beispielsweise erscheint ein kleines Fenster, wenn man den Zeiger über ein Element wie etwa eine Methode bewegt, das Meta-Informationen über dieses Element enthält. Diese Fenster (*Hover*) stellt der Editor bereit, welcher ebenfalls die Informationen dazu aus dem AST bezieht. In Scheme könnten diese Hover beispielsweise den Ort der Definition einer Variable anzeigen oder aus welcher Struktur das Modul die Variable importiert. Auch die Vorschläge in der Content Assist erscheinen mit Meta-Informationen in der Auswahl.

Die Liste der Verbesserungsmöglichkeiten für die Content Assist ist offen, denn es lassen sich immer Wege finden, eine IDE intelligenter zu machen. Ein gutes Beispiel, an dem man sich orientieren kann und dessen Quelltext frei verfügbar ist, stellt die Content Assist von JDT dar.

6.2.4 REPL

SDT zeigt den Scheme-48-Prozess über die eingebaute Eclipse-Konsole an. Das hat einige Nachteile: Es gibt keine Liste mit den zuletzt benutzten Befehlen, wie man es beispielsweise von einer Shell oder einem Terminal gewohnt ist. Außerdem ist die Handhabung umständlich: Erscheint eine Nachricht vom Prozess auf der Konsole, springt der Zeiger nicht automatisch an die Stelle der Eingabe, was für den Benutzer verwirrend sein kann. Das ist auch kein Bug, sondern

eine Eigenschaft der Eclipse-Konsole und wird auch von den Verantwortlichen in absehbarer Zukunft nicht verändert[1]. Eine Änderung der Eclipse-Konsole ist aufwändig und schwer durchführbar, da Eclipse die Standardkonsole für jeden neuen Prozess automatisch erstellt und anzeigt, ohne dass das Plugin in diesen Mechanismus eingegreifen kann. Zumindest theoretisch ist es daher nur möglich, eine zweite, eigene Konsole zu starten und die erste zu verstecken.

Es gibt von DLTK eine Implementierung einer fortgeschrittenen Konsole, jedoch konnte zumindest ich nicht herausfinden, ob diese auch mit dem in SDT verwendeten Mechanismus zur Einbindung des Scheme-48-Prozesses funktioniert und welche besondere Funktionalität diese Konsole überhaupt hat.

6.2.5 Migration von Actions zu Commands

Die in SDT verwendeten JFace-Komponenten zum Interagieren mit dem Benutzer, die Actions, sind in der aktuellen Eclipse-Version veraltet und werden wahrscheinlich in kommenden Versionen nicht mehr unterstützt. An Stelle der Actions treten `Commands`, die einige Vorteile gegenüber Actions besitzen[2]. Die in SDT verwendeteten Actions sollten daher durch entsprechende Commands ersetzt werden.

6.3 Ausblick

Neben der Verbesserung von bereits bestehenden Komponenten gibt es noch eine offene Liste mit Dingen, die SDT fehlen und hinzugefügt werden könnten.

6.3.1 Refactoring

Warum Refactoring im jetzigen Zustand von SDT nicht möglich oder nur schwer zu realisieren ist, habe ich im vorhergehenden Abschnitt erläutert. Es gibt jedoch viele nützliche Anwendungen, für die sich eine Implementierung von Refactoring lohnt:

- Automatisches Anlegen einer lokalen Variable mit einem `let`-Konstrukt für einen gewählten Ausdruck
- Vereinfachung von Audrücken
- Hinzufügen einer Bindung von einer Implementierungsdatei zu den exportierten Bindungen des zugewiesenen Default Package

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=96611
[2] http://wiki.eclipse.org/FAQ_What_is_the_difference_between_a_command_and_an_action%3F

- Umbenennung von Variablen und Aktualisierung aller Referenzen auf diese
- Extrahieren von Prozeduren aus Ausdrücken

Refactoring würde die Nützlichkeit der IDE erheblich steigern und das Programmieren mit Scheme 48 vereinfachen.

6.3.2 Unterstützung weiterer Scheme-Sprachen neben R^{5}RS

Module in Scheme 48 sind bezüglich der verwendeten Scheme-Sprache in den Implementierungsdateien agnostisch und geben alle Bindungen über das Importieren von entsprechenden Modulen vor. Dadurch ist es auch möglich, Module in PreScheme und teilweise in R^{6}RS zu schreiben. SDT unterstützt in seiner jetzigen Form ausschließlich R^{5}RS. Generell sind CL und R^{5}RS in SDT in manchen Komponenten, beispielsweise der semantischen Unterstützung, eng miteinander verwoben. Eine Einbindung anderer Sprachen ist aber über die Einführung von Einhängepunkten möglich, also Stellen, an denen andere Plugins Funktionalität für SDT bereitstellen können. Denkbar wäre, dass man eine entsprechende Sprache in den Einstellungen wählen kann oder einer Datei explizit zuweist.

Um weitere Sprachen verfügbar zu machen, sind folgende Schritte notwendig:

- Erstellung eines Einhängepunktes für Parser und `TreeWalker` einer neuen Scheme-Sprache. Dazu muss für beide Komponenten ein Interface erstellt werden, dass die gemeinsame Methoden für Parser und `TreeWalker` fordert. Der Einhängepunkt fordert Klassen, die diese Interfaces implementieren.
- Abstraktion über die Semantik von Programmen in den verschiedenen Scheme-Sprachen. Da die IDE nicht wissen soll, welche Scheme-Sprache der Benutzer explizit angibt, aber dennoch eine semantische Unterstützung anbieten soll, muss über die Semantik von Scheme-Programmen in den verschiedenen Sprachen abstrahiert werden. Dies ist automatisch mit Informationsverlust über die jeweiligen Eigenheiten der Sprachen verbunden. In der Semantik haben alle Scheme-Sprachen beispielsweise folgende Gemeinsamkeiten:
  - Konstrukte, die globale Bindungen anlegen
  - Konstrukte, die lokale Bindungen anlegen
  - Anlegen von Problemen (`DefaultProblems`), die der Editor markiert

  - Macros
  - ...

  Welche Gemeinsamkeiten der Semantik abstrahiert werden können, bedarf eine genauen Untersuchung der Sprachen, die Scheme 48 letztendlich unterstützt.

- Abstraktion des vorhandenen Codes in den Actions des R^{5}RS-Parsers und `-TreeWalkers`. Der Parser für R^{5}RS sammelt aktuell Informationen über die Semantik hauptsächlich über Actions in den Parser-Regeln. Da über die Semantik abstrahiert wird, sollten diese Actions in einer gemeinsamen Basisklasse gelagert werden, die der Parser nur noch an den entsprechenden Stellen aufgeruft. Das kann zum Beispiel die Bindung einer Variable sein oder das Erstellen eines Fehlers.

  Zur Abstraktion des Parsers gehört auch die Abstaktion aller Stellen, die zwar nicht direkt zum Parser gehören, aber implizit auf die Sprache Bezug nehmen. Dazu gehören beispielsweise die Liste von Schlüsselwörtern oder die Strategie der Einrückung nach bestimmten Schlüsselwörtern.

- Die Modellierung der Scheme-Dateien in der Bibliothek muss an die Abstraktion angepasst werden. Die neuen Parser sollten jeweils die vorhandenen Bibliotheken nutzen.

- Abstraktion über den AST einer Scheme-Datei. Dies kann auf zwei Arten geschehen: Entweder man implementiert einen AST, auf den alle Scheme-Sprache abbilden, oder man abstrahiert über die Algorithmen, die auf den Syntaxbäumen arbeiten, also beispielsweise die Outline erstellen, Content Assist berechnen oder ein Refactoring durchführen. Bei beiden Lösungen muss ein Einhängepunkt angelegt werden, der die entsprechenden Implementierungen entgegennimmt.

Literaturverzeichnis

[1] Borland Developer Studio 2006. URL http://www.codegear.com/en/products/bds2006.

[2] Eclipse CDT. URL http://eclipse.org/cdt/.

[3] Dynamic Languages Toolkit. URL http://eclipse.org/dltk/.

[4] Eclipse. URL http://eclipse.org/.

[5] Kawa. URL http://www.gnu.org/software/kawa/.

[6] NetBeans. URL http://netbeans.org/.

[7] SISC. URL http://sisc.sourceforge.net/.

[8] Visual Studio 2010. URL http://www.microsoft.com/germany/visualstudio/.

[9] Xtext. URL http://www.eclipse.org/Xtext/.

[10] H. Abelson, G. Sussman, J. Sussman, and A. Perlis. *Structure and interpretation of computer programs*, volume 2. MIT Press Cambridge, MA, 1996.

[11] W. Beaton and J. d. Rivieres. *Eclipse Platform Technical Overview*. The Eclipse Foundation, 2006.

[12] A. Bieniusa, M. Degen, P. Heidegger, P. Thiemann, S. Wehr, M. Gasbichler, M. Crestani, H. Klaeren, E. Knauel, and M. Sperber. Auf dem Weg zu einer robusten Programmierausbildung. In *Hochschuldidaktik der Informatik: 3. Workshop des GI-Fachbereichs Ausbildung und Beruf, Didaktik der Informatik, 04.-05. Dezember 2008 an der Universität Potsdam*, volume 1, page 67. Universitätsverlag Potsdam, 2009.

[13] *Turbo Pascal Reference Manual*. Borland International, 1984. URL http://www.textfiles.com/bitsavers/pdf/borland/Turbo_Pascal_Reference_Manual_Feb84.pdf.

[14] Dominique Boucher. SchemeWay. URL `http://sourceforge.net/apps/wordpress/schemeway/`.

[15] J. Bovet. ANTLRWorks: The ANTLR GUI Development Environment. URL `http://www.antlr.org/works/index.html`.

[16] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[17] R. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: A programming environment for scheme. *Journal of functional programming*, 12(2):159–182, 2002.

[18] H. Funke. Model based test specifications: Developing of test specifications in a semi automatic model based way. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 496–500. IEEE, 2011.

[19] R. Kelsey, W. Clinger, and J. Rees. *Revised^5 Reports on the Algorithmic Language Scheme*, 1998.

[20] R. Kelsey, W. Clinger, J. Rees, and M. Sperber. *The incomplete Scheme 48 reference manual for release 1.8*, 2008. URL `http://s48.org/1.8/s48manual.pdf`.

[21] B. D. O. Medeiros. Creation of an Eclipse-based IDE for the D programming language, 2007.

[22] T. Parr. *The definitive ANTLR reference: building domain-specific languages*. The Pragmatic Bookshelf, 2007.

[23] M. Schlüter. *Einführung in die Programmierung mit NATURAL*. Walter de Gruyter, 1993.

[24] F. Skopik. Moderne Softwareentwicklungsumgebungen – Evaluierung C++/C#-basierter Ansätze, 2007.

[25] M. Sperber, R.K. Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language scheme, 2009.

[26] G. Sussman and G. Steele. The first report on scheme revisited. *Higher-Order and Symbolic Computation*, 11(4):399–404, 1998.

[27] T. Wei, R. Zhang, X. Su, S. Chen, and L. Li. Gaussianscripteditor: an editor for gaussian scripting language for grid environment. In *Grid and Cooperative Computing, 2009. GCC'09. Eighth International Conference on*, pages 39–44. IEEE, 2009.

[28] M. Wihsböck. Moderne Softwareentwicklungsumgebungen – Evaluierung Java-basierter Ansätze, 2007.

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Masterarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift