AST-basierte Programmierung und ihre Eignung zur Umsetzung von Refaktorisierungen

Philipp Keller

Masterarbeit 29. März 2012
Fachbereich Programmiersysteme Prof. Dr. Steimann
Betreuer: Marcus Frenkel
FernUniversität Hagen

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt und durch meine Unterschrift, dass die vorliegende Arbeit von mir selbstständig, ohne fremde Hilfe angefertigt worden ist. Inhalte und Passagen, die aus fremden Quellen stammen und direkt oder indirekt übernommen worden sind, wurden als solche kenntlich gemacht. Ferner versichere ich, dass ich keine andere, außer der im Literaturverzeichnis angegebenen Literatur verwendet habe. Diese Versicherung bezieht sich sowohl auf Textinhalte sowie alle enthaltenden Abbildungen, Skizzen und Tabellen. Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Stadt, Datum Unterschrift

Abstract

Software wird durch das Schreiben von Quellcode in einer Programmiersprache erstellt. Der Quellcode ist Ausgangspunkt für den Compiler zur Erstellung von Maschinencode oder Bytecode. Der Quellcode in einer Sprache wird am häufigsten in textuellen Editoren verfasst. Diese Form des Schreibens von Quellcode ist fehleranfällig, da leicht syntaktisch falsche Sprachelemente in den Quelltext geschrieben werden können. Diese Fehler können aber auch dann entstehen, wenn durch Refaktorisierungen betroffene, abhängige Stellen innerhalb des Quelltextes geändert werden müssen, dies jedoch durch den Entwickler vergessen wird. Diese beschriebenen Anfälligkeiten verursachen syntaktische Fehler innerhalb des Quelltextes. Eine Weiterentwicklung der textuellen Programmierung ist das Schreiben des Codes auf Basis des Syntaxbaums eines Programms. Die hierarchische Struktur eines Programms wird durch einen Syntaxbaum dargestellt, welcher während der Kompilierung erstellt wird. Dieser Syntaxbaum wird als Abstract Syntax Tree (AST) bezeichnet. Innerhalb des AST werden die Ausdrücke des Programms in einer Baumstruktur dargestellt. Editoren, die auf dem AST einer Programmiersprache arbeiten, können den Aufwand für Formatierung und Prüfung des Codes reduzieren und somit die Produktivität des Entwicklers steigern, da der AST ein konsistentes Modell ist. Fehler, die aufgrund einer falschen Anwendung der Syntax entstehen, können in diesem Zusammenhang verhindert werden. Ziel dieser Arbeit ist es, die dahingehend existierenden Implementierungen und Studien zu untersuchen, um herauszufinden, welche Vorteile die AST-basierte Programmierung im Allgemeinen und bei der Umsetzung von Refaktorisierungen im Besonderen bietet. Gewonnene Erkenntnisse sollen durch die Implementierung eines AST-basierten Editors nachvollzogen und untersucht werden.

Inhaltsverzeichnis

Kapitel 1

Einführung

Nach Völler [Völ10, Compiler im Überblick] erzeugt ein Compiler aus einem Quellprogramm, das in einer Quellsprache geschrieben ist, ein Zielprogramm in einer Zielsprache. Während dieses Vorgangs muss der Quelltext gegen die Syntax der Quellsprache geprüft werden, bevor eine eigentliche Transformation möglich ist [Aho+08, Seite 3]. Der Vorgang des Kompilierens ist ein komplexer Vorgang und besteht unter anderem aus den Phasen der lexikalischen Analyse, der syntaktischen Analyse, dem Parsen des Quellcodes und dem Aufbau des AST sowie der semantischen Analyse, also der Prüfung des Codes auf Konsistenz. Erst auf Basis eines AST kann der Compiler die weiteren Schritte zur Erzeugung des Codes in der Zielsprache durchführen. Hierzu gehören zum Beispiel die Zwischencodegenerierung und die eigentliche Codegenerierung in der Zielsprache [Aho+08, Seite 6]. In modernen Programmierumgebungen wie der *Eclipse IDE* oder dem *Visual Studio* von *Microsoft* werden die Fehler im Quellcode direkt angezeigt. Dies geschieht, indem Entwicklungsumgebungen auf Eingaben reagieren und einen automatischen Syntax- und Semantik-Check starten, wenn dies als Option aktiviert wurde [Bei10, Seite 9]. Die Fehler, die diese Prüfungen aufzeigen, werden dann im eigentlichen Quellcode dem Programmierer angezeigt. Nach Restel [Res06, Seite 5] sind somit die Entwicklungsumgebungen zu einer Zwischenschicht geworden, die eine Vielzahl der möglichen Fehlerquellen ausschließen können.

Das Schreiben von Quellcode in einem Editor birgt ein hohes Maß an Fehleranfälligkeit, da erst durch die Prüfungen der syntaktischen Analyse festgestellt werden kann, ob der eingegebene Inhalt konform mit der Grammatik der Sprache ist. Durch die semantische Analyse können beispielsweise Fehler aufgedeckt werden, bei denen die Definition einer Variablen nicht mit den Referenzen oder der Nutzung im Quelltext übereinstimmt. Eine Möglichkeit

diese Fehlerquelle zu eliminieren wäre, dass der Programmierer stets ausschließlich korrekte Eingaben in seinem Editor tätigen kann. Dies könnte etwa dadurch gewährleistet werden, dass nur gültige Eingabe - und Bearbeitungsmöglichkeiten dem Programmierer zur Verfügung gestellt werden. In diesem Zusammenhang wäre es dann auch möglich, dass der Programmierer direkt auf dem Syntaxbaum des Programms seine Eingaben tätigt. Bei dieser Form der Eingabe wäre der zugrundeliegende AST also immer konsistent, da semantische und syntaktische Fehler ausgeschlossen würden. Diese Arbeit beschäftigt sich mit der Theorie der AST-basierten Programmierung und ihrer Eignung zur Umsetzung von Refaktorisierungen. Es soll herausgefunden werden, ob die AST-basierte Programmierung Vorteile bei der Umsetzung von Refaktorisierungen gegenüber der textuellen Programmierung bietet. Im Rahmen der Einführung wird der Unterschied zwischen textuellen, graphischen und AST-basierten Formen der Quellcodeerstellung behandelt. Im Anschluss werden Refaktorisierungen vorgestellt, deren Anwendbarkeit in AST-basierten Systemen untersucht werden soll. Daraufhin werden die Komponenten eines Compilers vorgestellt, die für den Aufbau des AST zuständig sind. Das darauf folgende Kapitel zeigt eine Übersicht der theoretischen Vorteile dieser Form der Programmierung und untersucht diese anhand bestehender Implementierungen. Den Abschluss dieser Arbeit bildet die Vorstellung der prototypischen Implementierung des ASTOP Editors. Dieser dient der Analyse der Implementierung von Refaktorisierungen auf dem AST-basierten Modell einer exemplarischen Sprache.

Kapitel 2

Grundlagen der Erstellung von Quellcode

Softwaresysteme werden immer größer und vielschichtiger, da immer neue Sachverhalte durch den Einsatz von Computersystemen realisiert werden. Somit wird natürlich auch die Erstellung der Software komplexer. Um die Erstellung dieser Systeme zu vereinfachen muss auch der eigentliche Produktionsprozess optimiert werden [Kel02, Seite 1]. Das Editieren von Quellcode kann auf unterschiedliche Arten durchgeführt werden, welche in den folgenden Abschnitten näher erläutert werden.

2.1 Textuelle Editoren

Textuelle Editoren stellen die am weitesten verbreitete Art und Weise zur Erstellung von Quellcode dar. Nach Leinenbach [Lei10, Seite 296] entstanden diese um die Programmierarbeit zu vereinfachen. Ziel war es, das handschriftliche Verfassen von Programmcodes und das Stanzen von Lochkarten abzuschaffen. In diesem Kontext entstanden Terminalarbeitsplätze mit textuellen Editoren zur gemeinsamen Durchführung der Arbeitsschritte. Über die Jahre wurden diese Editoren mehr und mehr zu integrierten Entwicklungsumgebungen. Diese bieten dem Programmierer eine Vielzahl von Funktionalitäten um zum Beispiel Quellcode schneller zu bearbeiten. Die Entwicklungsumgebungen bieten auch mehr und mehr Funktionalitäten an, um den Entwickler bei der Durchführung von Refaktorisierungen zu unterstützen. Bei Refaktorisierungen wird die Darstellung des Codes geändert, jedoch ohne die nach außen sichtbare Funktionsweise zu verändern. Weitere Funktionalitäten zur Optimierung des Entwicklungsprozesses sind das direkte Kompilieren aus dem Editor heraus oder die Code-Vervollständigung. Fehler innerhalb des Quellco-

de werden umgehend angezeigt, da moderne Entwicklungsumgebungen eine direkte Prüfung der Syntax und Semantik im Hintergrund durchführen. Der eigentliche Kern der Entwicklungsumgebung besteht jedoch nach wie vor aus der textuellen Erstellung von Quellcode in einem textbasierten Editor. Das textuelle Editieren von Quelltext bietet dem Programmierer eine Vielzahl an Freiheitsgraden. In vielen textuellen Editoren gibt es keine Einschränkungen hinsichtlich der Form und Gestaltung des Quelltexts, wie zum Beispiel die vorgegebene Anzahl von Zeilenabständen oder Einrückungen. Dies kann jedoch auch als Nachteil angesehen werden. Durch diese Möglichkeiten können persönliche Präferenzen hinsichtlich Abständen und Einrückungen uneinheitlich erstellt werden. Dies führt zu einer schlechteren Lesbarkeit und Wartbarkeit. Andere Entwickler denen der Quellcode übergeben wird benötigen eine längere Einarbeitungszeit um sich an den vorliegenden Formatierungsstil zu gewöhnen [Lau05, Seite 3]. Ein Vorteil jedoch liegt in der leichten Form der Dokumentation, die spezifisch und ohne Einschränkungen an die Bedürfnisse der Leser angepasst werden kann. Kommentare können an jeder Stelle innerhalb des Quelltextes geschrieben werden. Dies bietet zum Beispiel die Chance, dass Softwareentwicklungsteams gemeinsame Dokumentationsstandards definieren und anwenden können. Textuelle Editoren sind dadurch gekennzeichnet, dass mit ihnen sehr vereinfacht Software entwickelt werden kann, da selbst einfache Texteditoren ohne Funktionalitäten für bestimmte Sprachen geeignet sind um Quelltext zu schreiben. Eine andere Form der Editierung von Quelltext ist die Verwendung von graphischen Editoren.

2.2 Graphische Formen der Quellcodeerstellung

Quellcode kann nicht nur durch textuelle Editoren erstellt werden, sondern auch durch graphische Editoren. Graphische Editoren bieten die Möglichkeit, dass die Komplexität der Syntax einer Programmiersprache durch die Verwendung von vordefinierten Konstrukten verborgen werden kann. Damit abstrahieren graphische Editoren von der eigentlichen Syntax der zugrundeliegenden Programmiersprache. Darüber hinaus bieten sie eine bessere Übersicht über den erstellten Code [Ber11, Seite 18]. Dies kann bedeuten, dass der graphische Editor bestehende Konstrukte der Programmiersprache erstellt und im Hintergrund den eigentlichen Quelltext generiert. In diesem Fall bietet der graphische Editor eine einfachere Schnittstelle zur Erzeugung des Quelltexts.

2.3 AST-basierte Erstellung von Software

Es gibt auch Editoren, die direkt auf dem AST arbeiten, so das kein Quelltext erstellt wird. Diese Form der Editoren kann man auch als graphische Editoren bezeichnen, da sie auf dem Modell der Sprache arbeiten. Sie können beispielsweise den Quellcode als Graph visualisieren, so dass die Beziehungen von Elementen in der Sprache übersichtlicher dargestellt werden. Nach Ortmann [Ort96, Kapitel Vorteile grafischer Programmierung] bieten graphische Editoren zusätzliche Abstraktionsebenen durch die Darstellung einer zweiten Dimension. Dies kann zum Beispiel bedeuten, dass über die Darstellung des Quellcode auf Basis des AST die Abhängigkeiten von Variablendefinitionen, Prozeduraufrufen oder die Verwendung von externen Bibliotheken sichtbar gemacht werden können. Dieses Prinzip würde sich auch auf die Anwendung von Annotationen, externen Konfigurationen oder Aspekten beziehen lassen, deren Bindungen sich in einer textuellen Darstellung schlecht aufzeigen lassen.

Ein entscheidender Vorteil dieser Editoren ist die geringere Produzierbarkeit von syntaktischen und semantischen Fehlern durch den Programmierer, da der Editor bereits modellkonforme Elemente vorschlagen kann um damit dem Programmierer einen Teil der Arbeit abzunehmen. Somit kann der Programmierer direkt auf den Templates der Sprache arbeiten, ohne dass er die komplexe textuelle Syntax der Sprache kennen muss. Dies verringert den Einarbeitungsaufwand und führt zu schnelleren Ergebnissen, da eine Vielzahl von syntaktischen Fehlern ausgeschlossen werden kann. Des Weiteren können graphische Editoren bei denen der Benutzer auf dem AST einer Sprache arbeitet, bei der Modularisierung des Quellcode unterstützen, da die einzelnen Teile des Baumes leichter bearbeitet werden können. Die Darstellung der Struktur durch Graphen hat nicht nur den Vorteil der besseren Übersichtlichkeit, sondern unterstützt den Programmierer zudem bei der Programmausführung, der Refaktorisierung von Elementen und bei der Fehlersuche [Ort96, Seite 3].

Den hier beschriebenen Vorteilen stehen jedoch auch einige Nachteile gegenüber. Graphische Editoren bieten in der Regel feste Grenzen hinsichtlich der Funktionalität, da alle Templates und Validierungen bei der Implementierung des Editors bereits bekannt sein müssen. Damit gehen die Freiheitsgrade einer textuellen Beschreibung verloren. Hierdurch entsteht ein höherer Entwicklungsaufwand, da ein entsprechendes Datenmodell sowie graphische Elemente und Validierungen implementiert werden müssen. Bei einer Erweiterung des Modells müssen die Komponenten des Editors kontinuierlich erweitert werden [Ber11, Seite 18].

Mit zunehmender Komplexität des Quellcode in einer graphischen Darstellung verliert auch die gewonnene Übersichtlichkeit ihren Vorteil, da der Entwickler nicht mehr alle Zusammenhänge auf einen Blick überschauen kann. Somit verliert die einfachere Bedienung und Übersichtlichkeit schnell ihre Vorteile gegenüber einer textuellen Darstellung. Graphische Editoren haben sich in den letzten Jahren immer öfter in bestimmten Anwendungsdomänen durchgesetzt. In Kapitel 4 werden die am Markt existierenden Implementierungen mit ihren Vor- und Nachteilen ausführlich diskutiert. Hierbei wird untersucht ob diese Editoren direkt auf einem Syntaxbaum aufsetzen und welche Möglichkeiten diese zur Umsetzung von Refaktorisierungen bieten.

2.4 Konstruktion des Syntaxbaums

In diesem Abschnitt werden die Komponenten eines Compilers vorgestellt, die für den Aufbau und die Struktur eines AST im Rahmen eines Kompilierungsprozesses verantwortlich sind.

Nach Aho, Lam, Sethi und Ullman [Aho+08, Seite 3] handelt es sich bei einem Compiler um ein Programm, das ein Programm in einer Sprache lesen und in ein gleichwertiges Programm einer anderen Sprache übersetzen kann. Im Rahmen dieser Transformation erstellt der Compiler einen AST auf Basis des Quellcodes. Die folgenden Komponenten eines Compilers sind hierzu notwendig:

- Lexikalische Analyse
- Syntaktische Analyse
- Semantische Analyse

Typischerweise wird der Quellcode eines Programms beim Kompilieren in einen AST überführt. Dies geschieht während der syntaktischen Analyse. Der erstellte AST dient als Grundlage um Code in der eigentlichen Zielsprache zu erstellen. Der Zweck des AST liegt darin, dass der Compiler auf Basis dieser hierarchischen Baumstruktur die semantische Analyse durchführen kann. Die Bearbeitung in einer Baumstruktur führt dazu, dass unnötige Elemente wie Leerzeichen und Klammern wegfallen. Der AST einer Sprache ist also ein Zwischenschritt, der im Rahmen der Kompilierung in eine Zielsprache durchgeführt wird. Die Zielsprache ist zumeist Bytecode oder maschinenabhängiger Code. In den folgenden Abschnitten werden die Teilschritte, die zur Erstellung eines AST führen, erläutert.

2.4.1 Lexikalische Analyse

Bei der lexikalischen Analyse wird der eigentliche Quellcode eingelesen und in sogenannte Sequenzen überführt, die auch als Lexeme bezeichnet werden können. Diese Lexeme bestehen aus einem Namen, dem sogenannten Token, und dem eigentlichen Attributwert. Der Hauptzweck dieses Vorgangs ist, dass der Compiler die Struktur und Bedeutung des Programms versteht. Die Tokens können zum Beispiel Bezeichner, Schlüsselwörter, Zahlen und Operatoren innerhalb der Sprache kennzeichnen. Ein Beispiel hierfür wäre der Name einer Variable oder eines Schlüsselwortes. Hierbei kommt der Scanner des Compilers zum Einsatz, der den eigentlichen Quellcode analysiert. Dieser überspringt Leerzeichen, Kommentare und Tabulatorzeichen beim Analysieren des Quellcodes hinsichtlich der Lexeme [Aho+08, Seite 8]. Das Ergebnis der lexikalischen Analyse ist nicht die Darstellung als Baum, sondern eine Liste der gefundenen Typen in der Grammatik der Sprache mit seinen Attributwerten.

Zur Erkennung der Token und der Attributwerte werden *Reguläre Ausdrücke* verwendet. Die *Regulären Ausdrücke* sind auf die Grammatik der Sprache abgestimmt und ermöglichen somit die Unterscheidung zwischen Elementen der Sprache, Zahlen und Operatoren. Die definierten *Regulären Ausdrücke* kommen bei der Bearbeitung des Quellcodes durch den Scanner zum Einsatz. Dieser liest den Eingabestrom des Quelltexts und wendet dabei die *Regulären Ausdrücke* auf die Zeichen an. Die *Regulären Ausdrücke* definieren dabei einen *Endlichen Automaten*, der die zugehörige Sprache akzeptiert [Voe12, Endliche Automaten als Scanner]. Bei *Endlichen Automaten* handelt es sich um Modelle, die durch Aktionen verschiedene Zustände definieren. Bei der lexikalischen Analyse werden sie verwendet um zu bestimmen, ob

eine gefundenen Zeichenkette eine Instanz eines regulären Ausdrucks ist. Die *Endlichen Automaten* können in nichtdeterministische (*NEA)* und deterministische Automaten (*DEA*) unterschieden werden. Ein *NEA* kann von einem bestimmten Zustand durch die Eingabe eines weiteren Zeichens in mehrere andere Zustände wechseln. Durch die Eingabe eines Zeichens kann also der daraus resultierende Zustand nicht eindeutig bestimmt werden. Der *DEA* verhält sich dahingehend anders, da er von einem bekannten Zustand durch die Eingabe eines weiteren Zeichens in einen bestimmbaren anderen Zustand übergehen kann. Ein einfaches Beispiel für einen *DEA* ist ein Lichtschalter, der die Zustände *An* und *Aus* annehmen kann [Pol07, Seite 4]. Auf die lexikalische Analyse folgt die syntaktische Analyse, die für den Aufbau des abstrakten Syntaxbaums verantwortlich ist.

2.4.2 Syntaktische Analyse

Die syntaktische Analyse übernimmt die aus der lexikalischen Analyse hervorgegangenen Token. Hierbei wird geprüft, ob diese den syntaktischen Regeln der Programmiersprache entsprechen. Zudem wird untersucht, ob die Anordnung und Reihenfolge der gefundenen Token mit der Grammatik der Sprache übereinstimmt. Im Falle eines Fehlers ist diese Phase verantwortlich dafür entsprechende Fehlermeldungen zu generieren. Wenn Anordnung und Reihenfolge korrekt sind, so baut die syntaktische Analyse den Syntaxbaum auf. Wenn Fehler auftreten, dann wird zumindest teilweise der Syntaxbaum erstellt [Huh08, Seite 7]. Verantwortlich für die Durchführung der syntaktischen Analyse ist der Parser des Compilers. Innerhalb des Syntaxbaums stehen einzelne Knoten für Operationen und die Kindelemente dieser Knoten stehen für die Argumente dieser Operationen [Aho+08, Seite 9]. Der Syntaxbaum wird auch als Parse Tree bezeichnet und ist der Ausgangspunkt für die folgenden Phasen des Compiliervorgangs. Der Syntaxbaum enthält keine Informationen über Klammerungen und Leerzeichen, da diese bereits implizit durch die Struktur des Baums vorgegeben werden. Der Baum wird aus diesem Grund auch als Abstrakter Syntaxbaum bezeichnet, da viele Informationen des ursprünglichen Quelltexts weggefallen sind.

2.4.3 Semantische Analyse

Bereits nach der syntaktischen Analyse liegt der Code in einer Baumstruktur vor. Die eigentliche Semantik wurde bisher jedoch nicht validiert. Die Semantik eines Programms stellt die Bedeutung und die Beziehungen der einzelnen Elemente aus der syntaktischen Analyse her. Die semantische Analyse wird auf der kompletten Baumstruktur durchgeführt.

Hierbei werden zum Beispiel die Deklarationen von Variablen geprüft, die innerhalb des Baums in Zuweisungen verwendet werden. Des Weiteren werden Prozeduraufrufe dahingehend validiert, dass Anzahl, Typ und Reihenfolge der Parameter mit der aufgerufenen Prozedur übereinstimmen. Zu den Operationen werden auch die entsprechenden Typen gesucht. Die entsprechende Typbindung wird entweder am Operator oder über den entsprechenden Kontext analysiert. Eine weitere Analyse wird bei der Sichtbarkeit von Variablen durchgeführt. Hierbei wird zwischen lokalen und globalen Variablen unterschieden und es wird geprüft, ob eine lokale Variable eine globale Variable im entsprechenden Kontext überlagert, oder ob eine Referenz in einem globalen Kontext auf eine lokale Variable existiert. Zur Bestimmung der Sichtbarkeit werden Variablen auf einen Stapel gelegt, auf dem jeweils nur die aktuellen Variablen des Kontext sichtbar sind. Die gefundenen Attribute werden auf dem AST an den entsprechenden Knoten abgelegt [Pin00, Seite 4]. Ziel der semantischen Analyse ist die Erstellung des attributierten Syntaxbaums. Hierbei werden die Elemente des Baums mit ihren eigentlichen Werten und Typen versehen. Zu diesen Informationen gehören beispielsweise die Deklaration einer Variablen und ihre Verwendung innerhalb des Quellcode sowie der Typ der Variablen. Durch die Symboltabelle kann der Syntaxbaum mit den entsprechenden Werten attributiert werden.

2.4.4 Interne Darstellung von Syntaxbäumen

Durch die Verfahren der *Objektorientierten Modellierung* lassen sich Syntaxbäume darstellen. Attribute werden über Membervariablen dargestellt und die Bindung eines Bezeichners lässt sich als Referenz innerhalb des Syntaxbaums modellieren. Somit lassen sich auch die Abhängigkeiten der semantischen Analyse innerhalb eines Syntaxbaums darstellen. Baumstrukturen sind ein sehr effizientes Mittel, da sie sehr leicht programmatisch zu durchlaufen sind. Durch die bereits dargestellte leichte objektorientierte Darstellung lässt sich der Syntaxbaum gut zu Analysezwecken nutzen. Auch als Darstellungsform für Programmierer bietet der Syntaxbaum eine sehr gute

strukturelle Übersicht mit dem Vorteil der Darstellung der Referenzen der semantischen Analyse. Abhängigkeiten zwischen Variablen und deren Referenzen oder Prozeduren und deren zugeordneten Prozeduraufrufen können somit für den Entwickler direkt sichtbar gemacht werden [Pin00, Seite 5].

2.4.5 Anwendung in modernen Entwicklungsumgebungen

In modernen Entwicklungsumgebungen wie *Eclipse* oder *Visual Studio* werden bei jeder Änderung des Quellcodes die obigen Teilschritte durchgeführt. Dies geschieht dadurch, dass die Entwicklungsumgebungen nach jeder Anpassung den Kompilierungsprozess starten können. Fehler werden direkt an der richtigen Stelle angezeigt und zumeist werden auf Basis des angezeigten Fehlers auch Lösungsvorschläge angeboten. Es werden jedoch nicht nur die Fehler der syntaktischen Analyse angezeigt. Auch auf semantische Fehler wie Bezeichner ohne Definition oder falsche Prozeduraufrufe wird der Entwickler unmittelbar nach der Eingabe hingewiesen. Der Quelltext kann somit in einem inkonsistenten Zustand vorliegen. Erst nach einer Eingabe wird der Entwickler auf die Fehler aufmerksam gemacht. Eine Möglichkeit diese Fehler bei der Erstellung von Programmen zu vermeiden wäre es, wenn der Entwickler direkt auf dem AST einer Programmiersprache arbeiten könnte. Wie bereits definiert würden in diesem Fall die Definition von Schlüsselwörtern sowie Klammerungen und Leerzeichen bei der Editierung wegfallen. Ein Editor, der auf Basis eines AST arbeiten würde, müsste jedoch auch die Konsistenz der Eingaben validieren, bevor die Eingaben des Benutzers im AST angelegt werden. In diesem Kontext spricht man von AST-basierten Editoren. Das bedeutet, dass der Entwickler während der Erstellung bereits auf Fehler aufmerksam gemacht werden würde. Dieses Prinzip wurde in einigen Forschungsarbeiten sowie in kommerziellen und nicht-kommerziellen Produkten aufgegriffen. In Kapitel 4 werden AST-basierte Implementierungen hinsichtlich ihrer Fähigkeiten im Allgemeinen und ihren Eigenschaften zur Durchführung von Refaktorisierungen im Besonderen untersucht.

Kapitel 3

Refaktorisierungen in Entwicklungsumgebungen

3.1 Refaktorisierungen in textuellen Entwicklungsumgebungen

Nach Fowler [Fow99, Seite 6] werden bei Refaktorisierungen die Strukturen von Quellcode dahingehend geändert, dass nach außen die gleiche Benutzbarkeit bestehen bleibt, jedoch der Code hinsichtlich seiner Gestaltung und / oder Ausführungsgeschwindigkeit optimiert wird. Jeder Schritt bei einer Refaktorisierung muss sicherstellen, dass die Ausführbarkeit des Systems nicht beeinträchtigt wird. Das bedeutet, dass vor und nach einer Refaktorisierung der Quelltext in einem konsistenten Zustand sein muss. Bei Refaktorisierungen auf einem AST gilt dies im Besonderen, da ein AST eines Programms immer in sich konsistent sein muss. Gerade bei der Durchführung von Refaktorisierungen auf einem AST können abhängige Stellen innerhalb des Syntaxbaums direkt geändert werden, da der Baum in sich Referenzen auf abhängige Stellen hat. Dies wird nicht automatisch bei einer textuellen Repräsentation des Quelltext durchgeführt. Die nachfolgenden Refaktorisierungen gehören zu standardisierten Refaktorisierungen die bereits in Entwicklungsumgebungen implementiert sind. In der Eclipse Entwicklungsumgebung für Java werden zum Beispiel im Standard bereits einige Refaktorisierungen unterstützt, die auf Basis des textuellen Editors ausgeführt werden können. Somit unterstützt Eclipse den Entwickler dabei, die Signatur einer aktuellen Methode zu verändern oder einen selektierten Abschnitt im Quellcode in eine neue Methode auszulagern. Auf der Ebene einer Klasse kann die Definition einer Methode in ein Interface oder in eine übergeordnete Klasse ausgelagert werden. Des Weiteren können aus lokalen Variablen globale Variablen und

umgekehrt erstellt werden. Dies sind nur einige Beispiele wie moderne Entwicklungsumgebungen die Entwickler dabei unterstützen, Refaktorisierungen auf bestehendem Quellcode durchzuführen. Es ist jedoch anzumerken, dass die implementierten Refaktorisierungen nicht immer zu einem korrekten Quellcode führen. Diese Funktionalitäten sind direkt in den Entwicklungsumgebungen implementiert. Nach Schäfer und de Moor [Sd10, Seite 1] sind diese implementierten Refaktorisierungen schwer zu warten und stellen durch ihre Implementierung die einzige vorhandene Spezifikation dieser Maßnahmen dar. Es stellt sich nun die Frage, ob die bestehenden Refaktorisierungen in ihrer Komplexität leichter auf Basis eines Syntaxbaums realisierbar sind und ob die syntaktische und semantische Korrektheit des AST bewahrt werden kann. Entwicklungsumgebungen bieten stetig wachsende Funktionalitäten zur Steigerung der Produktivität. Dies gilt auch für die Durchführung von Refaktorisierungen. Die Durchführung von automatisierten Refaktorisierungen gestaltet sich vor allem dann als schwierig, wenn sie den Abschluss in einer syntaktisch und semantisch korrekten Form bieten muss [Sd10, Seite 1]. Damit Refaktorisierungen in einer Entwicklungsumgebung wie *Eclipse* oder *Visual Studio* durchgeführt werden können, werden auf dem selektierten Kontext die Vorbedingungen geprüft. Sind diese Vorbedingungen vorhanden, so kann die Refaktorisierung durchgeführt werden, jedoch ohne die Gewissheit dass das Ergebnis den Anforderungen der Konsistenz eines AST entsprechen würde.

3.2 Refaktorisierungen in AST-basierten Umgebungen

Refaktorisierungen in textuellen Entwicklungsumgebungen sind dahingehend implementiert, dass der Quelltext eingelesen und analysiert wird. Auf Basis der Analyse können dann die Refaktorisierungen durch die Veränderung der Zeichenketten und das Zurückschreiben der Änderungen in den Quelltext realisiert werden. AST-basierte Implementierungen arbeiten jedoch nicht auf einem Quelltext sondern auf einem Syntaxbaum. Somit sind in diesem Zusammenhang Refaktorisierungen auf eine andere Art zu realisieren. Die Veränderungen müssen also direkt auf dem Syntaxbaum des entsprechenden Programms durchgeführt werden und anschließend in der graphischen oder textuellen Projektion dem Entwickler angezeigt werden. Des Weiteren ist bei der AST-basierten Programmierung die Konsistenz des Syntaxbaums einzuhalten. Refaktorisierungen müssen also immer in einem konsistenten Zustand enden, was bei der textuellen Programmierung nicht notwendigerweise gegeben ist. Inwiefern Refaktorisierungen auf AST-basierten Systemen sinnvoll und realisierbar sind, wird im Rahmen dieses Kapitels untersucht.

3.3 Standardisierte Refaktorisierungen in Entwicklungsumgebungen

Im Folgenden werden einige Refaktorisierungen vorgestellt, die als automatisierte Prozesse in der Entwicklungsumgebung *Eclipse* implementiert sind. Die Abschnitte stellen jeweils die einzelnen Refaktorisierungen und notwendige Bedingungen zur Sicherstellung der Konsistenz vor. Diese Refaktorisierungen sollen im Verlauf dieser Arbeit auf bestehenden AST-basierten Editoren geprüft werden und deren Implementierbarkeit in einem prototypischen AST-basierten Editor validiert werden.

3.3.1 Refaktorisierungen von Variablen

Eine in *Eclipse* und *Visual Studio* implementierte Refaktorisierung ist die Umwandlung einer lokalen Variable zu einem Feld innerhalb einer Klasse. Nach Schäfer und de Moor [Sd10, Seite 3] ist eine wesentliche Bedingung zur Durchführung dieser Refaktorisierung, dass alle Referenzen nach der Umwandlung einer Variable zu einem lokalen Feld nun von diesem abhängen und die lokale Variable nicht mehr existent ist. In einer objektorientierten Sprache im Kontext der Vererbung kann diese Refaktorisierung noch erweitert werden.

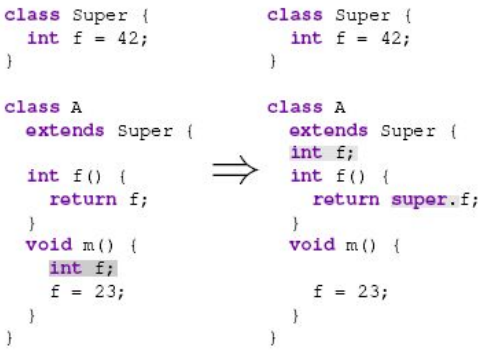
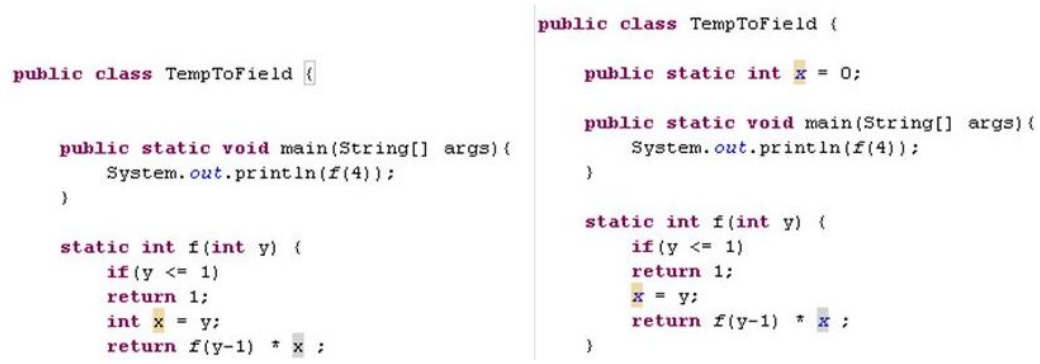


Abbildung 3.1: Umwandlung einer lokalen Variable zu einem Feld (entnommen aus [Sd10, Seite 3])

In Abbildung 3.1 überschreibt eine lokale Variable ein von einer Superklasse geerbtes Feld. Die lokale Variable soll ebenfalls zu einem Feld der Subklasse werden. Eine wesentliche Voraussetzung zur Durchführung der Refaktorisierung ist, dass analysiert wird, welche Referenzen sich auf das geerbte Feld und welche sich auf die lokale Variable beziehen. Durch gezielte Sperrungen der Referenzen werden somit nur die Referenzen geändert, die durch die Refaktorisierung angesprochen werden. Ein besonders schwieriger Fall tritt auf, wenn eine lokale Variable im Kontext einer Rekursion verwendet wird und diese als Feld ausgelagert werden soll. Schäfer und de Moor [Sd10, Seite 1] zeigen auf, dass bei einem derartigen Fall wie in Abbildung 3.2 zu sehen, der Wert von x durch die Rekursion ständig überschrieben wird. Diese Refaktorisierung in Abbildung 3.2 lässt sich in textuellen Editoren erschwert analysieren, da die Rekursion vor der Durchführung erkannt werden muss. Im Rahmen der Analyse bestehender AST-basierter Editoren wird geprüft, wie sich eine derartige Refaktorisierung auf Basis eines Syntaxbaums besser realisieren lässt, indem die Zyklen analysiert werden.



```
public class TempToField {

    public static void main(String[] args){
        System.out.println(f(4));
    }

    static int f(int y) {
        if(y <= 1)
        return 1;
        int x = y;
        return f(y-1) * x ;
```

```
public class TempToField {

    public static int x = 0;

    public static void main(String[] args){
        System.out.println(f(4));
    }

    static int f(int y) {
        if(y <= 1)
        return 1;
        x = y;
        return f(y-1) * x ;
    }
```

Abbildung 3.2: Rekursionen bei der Refaktorisierung lokaler Variablen zu Eigenschaften

In Schäfer und de Moor [Sd10, Seite 6] wird ein Vorgehensschema beschrieben, welches aus vier verschiedenen Phasen besteht um eine Refaktorisierung vom Typ *Move Temp to Field* durchzuführen. Hierbei werden zuerst die existierenden Abhängigkeiten gesperrt. Im Anschluss werden die Initialisierungswerte der Referenz zugewiesen. Daraufhin muss geprüft werden, ob es nicht zu Namenskonflikten kommt und ein Feld mit dem entsprechenden Bezeichner auf Klassen - oder Programmebene erstellt werden kann. Wenn das Feld erfolgreich erstellt werden konnte, können nicht benötigte Referenzen im Quelltext gelöscht werden.
Im Rahmen dieser Arbeit wird in den folgenden Kapiteln nun untersucht, inwiefern sich diese Vorgehensmodelle auf Basis eines AST-basierten Editors ausführen lassen. In Kapitel 6, welches sich mit dem ASTOP Editor beschäftigt, werden dann auf Basis einer konkreten Implementierung die einzelnen Schritte implementiert.

3.3.2 Refaktorisierungen von Methoden

Die hierarchische Verschiebung von Variablen im Rahmen einer Refaktorisierung kann auch auf Methoden angewandt werden. In einer objektorientierten Sprache kann eine Methode einer Superklasse in die Subklassen verlagert werden. In dem Vorgehensschema von Schaefer und de Moor [Sd10, Seite 7] werden Phasen beschrieben, um die Methoden über mehrere Schritte von der übergeordneten Superklasse in die Subklasse zu verlagern.

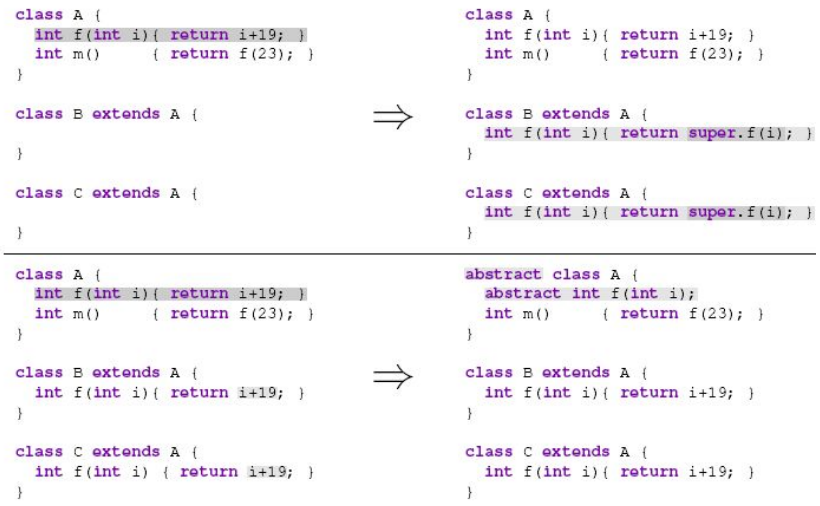


Abbildung 3.3: Refaktorisierung Move Down Method (entnommen aus [Sd10, Seite 8])

In der Untersuchung von Schaefer und de Moor [Sd10, Seite 6 - 10] wird auch für die Durchführung dieser Refaktorisierung ein Vorgehensmodell beschrieben. Hier müssen zunächst Vorbedingungen geprüft werden, ob es sich zum Beispiel um finale oder abstrakte Methoden handelt. Ziel ist die Einführung einer neuen Methode, die auf die noch bestehende Methode der Superklasse verweist. Das Ergebnis stellt eine syntaktische und semantisch korrekte Refaktorisierung dar, die Ausgangspunkt für weitere Refaktorisierungen sein kann. Die Methode der Superklasse kann nun gelöscht oder Klasse und Methode können als abstrakt bezeichnet werden. Ziel jeder Transformation ist auch hier, dass nur die Refaktorisierungen durchgeführt werden, bei denen das Ergebnis syntaktisch und semantisch korrekt ist. Abbildung 3.3 aus der Untersuchung von Schäfer und de Moor [Sd10] zeigt die Durchführung dieser Refaktorisierung.

3.3.3 Weitere Refaktorisierungen

Neben den oben beschriebenen Refaktorisierungen, deren Durchführbarkeit auf AST-basierten Editoren geprüft wird, gibt es noch eine weitere Refaktorisierung, deren Ausführbarkeit und Implementierung im Rahmen dieser Arbeit untersucht werden soll. Bei der Refaktorisierung *Extract to Procedure* wird ein Teil des Quelltextes selektiert und in eine neue Prozedur ausgelagert. Hierbei geht es darum die Komplexität in einer Prozedur zu reduzieren indem ein Teil der Funktionalität ausgelagert wird. Eine weitere Refaktorisierung ist das Hinzufügen und Entfernen von Parametern an eine Funktion. Gerade in AST-basierten Editoren muss die Konsistenz geprüft werden, es darf zum Beispiel kein Parameter gelöscht werden, der innerhalb einer Prozedur verwendet wird.

Kapitel 4

Existierende AST-basierte Editoren

Die Programmierung auf dem AST bietet theoretisch interessante Perspektiven. In diesem Kapitel werden verschiedene am Markt vorhandene Implementierungen sowie im Rahmen wissenschaftlicher Arbeiten entstandene Realisierungen vorgestellt und hinsichtlich ihrer Fähigkeiten und Funktionalitäten im Kontext von Refaktorisierungen untersucht.

4.1 Eigenschaften AST-basierter Editoren

Im Rahmen dieser Arbeit wird nicht nur die Anwendbarkeit von Refaktorisierungen auf AST-basierten Implementierungen geprüft. Es sollen auch allgemeine Funktionalitäten und die Reife derzeitiger Realisierungen geprüft werden. Aus diesem Grund werden im Folgenden Abschnitt Erwartungen an AST-basierte Entwicklungsumgebungen diskutiert. Hierbei handelt es sich um die theoretischen Möglichkeiten AST-basierter Systeme. Ob und inwiefern diese Funktionalitäten bereits existieren ist ebenfalls Gegenstand dieser Untersuchungen.

4.1.1 Erwartungen an AST-basierte Entwicklungsumgebungen

Wie bereits erläutert entfallen bei der AST-basierten Programmierung die Phasen der syntaktischen und semantischen Analyse, die zur Erstellung eines AST durch den Compiler durchlaufen werden würden. Somit sind direkte Validierung und direktes Feedback auf Eingaben ein entscheidendes Kriterium dieser Form der Programmierung. Korrekte Syntax und Semantik ermöglichen, dass der Baum direkt ausgeführt werden kann. Aufwände für Validierungen entfallen, da diese bereits bei der Eingabe vorgenommen wurden. Die Beziehungen in einem Syntaxbaum können ebenfalls Vorteile bei der Visualisierung bieten. Textuelle Darstellungen ermöglichen keine Ansicht der semantischen Referenzen innerhalb des Quelltexts, außer durch das Parsen des Quelltexts. AST-basierte Editoren sollten also die Möglichkeit verschiedener Sichten bieten, so dass auch semantische Beziehungen dargestellt werden können. Der AST, der als Objektmodell abgebildet wird, kann zum Beispiel die Referenzen zwischen der Definition einer Variablen und ihrer Verwendungen aufzeigen. In diesem Zusammenhang können ohne großen Aufwand Änderungen an den Bezeichnern vorgenommen werden. Diese Änderungen wirken sich dann auch auf alle Referenzen innerhalb eines AST aus. Dieser Vorteil sollte auch auf andere Referenzen innerhalb des Quelltexts angewandt werden können. Ein Beispiel wären die Referenzierungen von abhängigen Bibliotheken durch die Verwendung derer Objekte im eigenen Code. Bei der Arbeit mehrerer Entwickler auf einer Codebasis müssen natürlich auch Versionsverwaltungssysteme zum Einsatz kommen. AST-basierte Editoren müssten also ihre Strukturen auch in Versionsverwaltungssystemen speichern können. Die Speicherung der konsistenten Objektmodelle in den Datenbanksystemen von Versionsverwaltungssystemen kann Vorteile im Rahmen der Codeanalyse bieten. Da Code nur als konsistentes Modell eingecheckt werden kann, müssten auch keine automatischen Kompilierungsprozesse gestartet werden, um syntaktische und semantische Fehler festzustellen.

4.1.2 Bearbeitungssichten in AST-basierten Entwicklungsumgebungen

Da der AST ein konsistentes Modell des Programms darstellt, sollte dieser auch dazu genutzt werden um Entwicklern verschiedene Bearbeitungssichten auf den Code zu gewähren. Vorstellbar sind graphische Sichten auf die Ablauflogik ähnlich einem Ablaufdiagramm in der *Unified Modeling Language* (*UML*) oder graphenbasierte Bearbeitungsmöglichkeiten. Durch unterschied-

liche Sichten kann der Programmierer besser verschiedene Teilaufgaben realisieren. Auf Basis einer Klassendiagramm-Sicht könnten zum Beispiel die Klassen und Pakete erstellt werden, bevor über eine Detailansicht der Entwickler in die Definition von Eigenschaften und Methoden wechseln kann. Auf Basis eines konsistenten Objektmodells würden sich also eine Vielzahl von Bearbeitungssichten für verschiedene Phasen des Entwicklungsprozesses erstellen lassen, die die Arbeit vereinfachen und damit auch den Entwicklungsprozess beschleunigen.

4.1.3 Risiken AST-basierter Editoren

Den im obigen Abschnitt vorgestellten Chancen stehen jedoch auch einige Nachteile gegenüber. Textuelle Editoren sind sehr leicht zu erstellen. Eine besondere Schwierigkeit liegt jedoch in der Erstellung AST-basierter Editoren. Bei dieser Form von Editoren muss besonders bei der Eingabe und bei der Änderung von Abschnitten im Code darauf geachtet werden, dass alle syntaktischen und semantischen Fehler abgefangen werden, so dass der Syntaxbaum des Programms weiter ausführbar bleibt. Inkonsistenzen können vor allem dann auftreten, wenn der Entwickler größere Refaktorisierungen plant und diese zur Folge haben, dass der Code partiell in einen inkonsistenten Zustand gerät. Die Validierung und Prüfung der Eingaben, aber auch die Validierung der geplanten Refaktorisierungen stellen einen großen Teil des Entwicklungsaufwands bei der Erstellung von AST-basierten Editoren dar. Änderungen in der Sprache gehen zwangsläufig einher mit der Anpassung und Erweiterung des zugrundeliegenden AST-basierten Editors. AST-basierte Editoren werden somit zu einem Bestandteil der Sprache, da sie den gleichen Entwicklungszyklus durchlaufen müssen.

AST-basierte Programmierung ist Gegenstand einiger Implementierungen und wissenschaftlichen Arbeiten. Unter den analysierten Implementierungen sind auch Strukturediteren, bei denen im Rahmen dieser Arbeit auch untersucht werden muss ob es sich hierbei um echte AST-basierte Editoren handelt. Hauptaugenmerk liegt auf Anwendungsbereichen und Funktionalitäten sowie der im vorigen Kapitel vorgestellten Eignung für Refaktorisierungen.

Die analysierten Implementierungen haben jeweils unterschiedliche Schwerpunkte und Anwendungsszenarien. Einige sind experimenteller Natur und auf Basis von Forschungsarbeiten entstanden. Bei anderen Systemen liegt der Schwerpunkt eher auf vereinfachter Anwendungsentwicklung für technisch versierte Benutzer. Diese Systeme zeichnen sich vor allem dadurch aus,

dass die Komplexität der Anwendungsentwicklung reduziert werden soll. Ein weiterer Anwendungsbereich sind Entwicklungsumgebungen, bei denen das spielerische Erlernen der Programmierung im Vordergrund steht und Entwicklungsumgebungen zur Definition von neuen Sprachen. In den folgenden Abschnitten dieser Arbeit werden die verschiedenen Editoren vorgestellt, deren allgemeine Funktionen und insbesondere die Durchführung von Refaktorisierungen im Besonderen geprüft wird.

4.2 LavaPE

4.2.1 Allgemeine Einführung in LavaPE

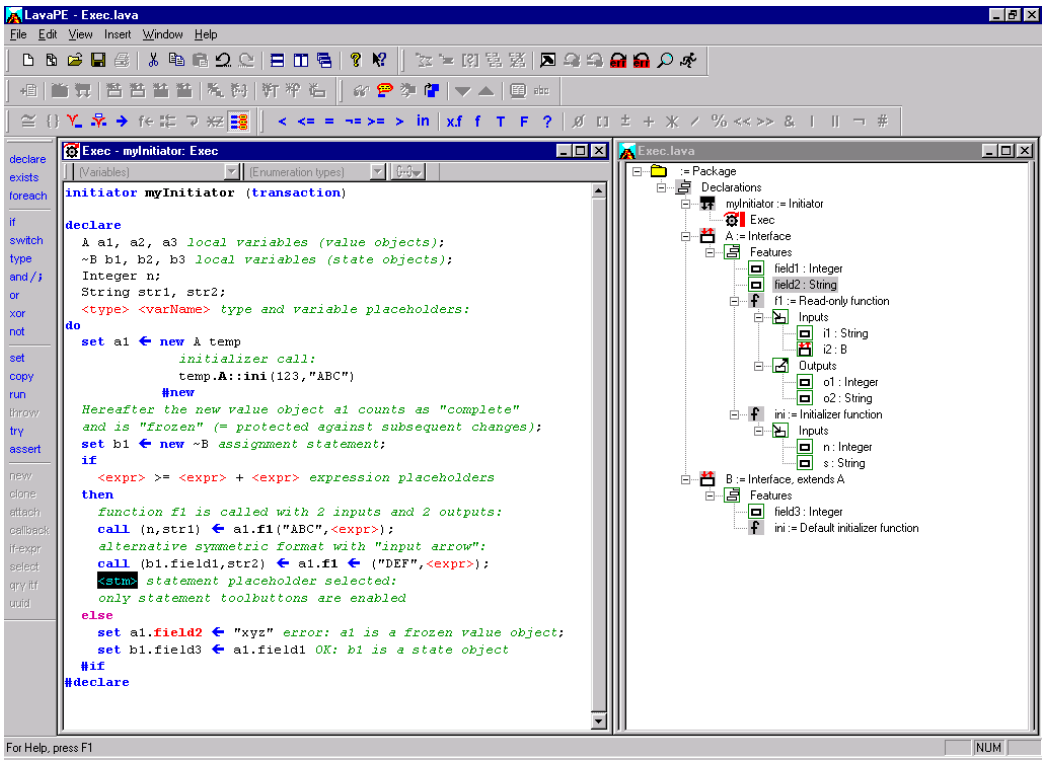


Abbildung 4.1: LavaPE Editor

Die Ursprünge der Entwicklung der Programmiersprache Lava und der Entwicklungsumgebung LavaPE lagen in dem Wunsch, den Softwareentwicklungsprozess, aber auch die Wartung und Weiterentwicklung bestehender Systeme zu optimieren. Bei Lava handelt es sich um eine eigenständige ob-

jektorientierte Sprache, die lediglich über einen strukturierten Editor bearbeitet werden kann. Textuelle Eingaben für den Programmierer werden nur bei Bezeichnern und Konstanten im Programmablauf eingegeben. Die einzelnen Deklarationen einer Klasse können über eine Baumstruktur angezeigt werden. Einzelne Methoden innerhalb einer Klasse werden über einen Editor dargestellt, welcher die elementaren Programmierkonstrukte in einer textuellen Darstellung zur Verfügung stellt. Abbildung 4.1 zeigt den entsprechenden Editor. Der Editor innerhalb der Methoden reagiert sensitiv auf seine Umgebung. Das bedeutet, dass nur die Strukturelemente aktiv sind, die als neue Codebestandteile eingesetzt werden können. Die einzigen editierbaren Elemente sind die Bezeichner oder Kommentare, die in den einzelnen Programmierkonstrukten eingebracht werden können. Textueller Code in herkömmlichen Editoren sieht von Programmierer zu Programmierer unterschiedlich aus. Die Zeilenabstände und die Einrückungen werden durch die Templates in LavaPE exakt vorgegeben. Somit entsteht für jeden Leser der Syntax ein einheitliches Bild, was die Lesbarkeit verbessert. Dadurch wird die Einarbeitungszeit reduziert und die Wartbarkeit erhöht. Durch die sensitiven Strukturelemente können Syntaxfehler bei der Erstellung nicht auftreten. Semantische Fehler werden unmittelbar nach der Eingabe angezeigt. Ein Beispiel ist die Verwendung eines Bezeichners, der nicht in der Deklaration einer Variable oder eines Objekts verwendet wird. Zwischen der sogenannten Struktursicht, die den Aufbau der Klassen und ihrer Eigenschaften anzeigt und der eigentlichen Methodensicht, in der der Programmablauf mit textuellen Fragmenten angezeigt wird, kann der User beliebig wechseln. Somit kann von der Verwendung eines Objekts oder einer Variable direkt zur eigentlichen Deklaration innerhalb der Klasse oder in einer anderen Klasse gesprungen werden. Umbenennungen von Bezeichnern wirken sich direkt auf alle Referenzen aus. Schwachstellen der Implementierung liegen jedoch im Bearbeitungsmodus. Die textuelle Form innerhalb der Methoden bietet für den Benutzer trotz der Vielzahl an implementierten Funktionalitäten nur wenige Freiheitsgrade. Der Programmierer muss also innerhalb der Prozeduren genaue Kenntnis über seine verfügbaren Variablen, Methoden und Objekte besitzen, so dass er diese in die Codefragmente ohne Fehler einsetzen kann. Auch die einzelnen Codefragmente müssen über eine Bedienoberfläche ausgewählt und eingefügt werden. Die einzelnen Objekte innerhalb der Struktur besitzen kein Kontextmenü über welches Operationen auf dem selektierten Element möglich sind. Der Programmierer ist dazu gezwungen sämtliche Operationen über das Navigationsmenü des Editors aufzurufen [GG00, Seite 1].

4.2.2 Refaktorisierungen in LavaPE

Der Lava Editor bietet keine Unterstützung bei der Analyse der Referenzen. Das Umbenennen eines virtuellen Typs oder einer Klasse wird unterstützt indem sämtliche Referenzen ebenfalls bei einer Namensänderung automatisch neu dargestellt werden, jedoch bewirkt die Löschung einer Definition, dass mögliche Abhängigkeiten nicht untersucht werden und damit der Syntaxbaum automatisch in einen inkonsistenten Zustand gerät. In LavaPE können Packages und Interfaces mit Typ Parametern ausgestattet werden. Hierbei handelt es sich um die oben genannten virtuellen Typen. Diese können in abgeleiteten Interfaces oder Packages auch überschrieben werden. Somit können die Typen von Eigenschaften, lokalen Variablen und Funktionsparametern auch virtuelle Parameter sein. Basierend auf diesem Konzept von virtuellen Typen ermöglicht Lava die Definition von Familien zusammengehöriger, wechselseitig rekursiver Interfaces mit *kovarianter Spezialisierung* virtueller Member- und Funktionsargument-Typen [Gün00, Seite 11]. In Abbildung 4.2 soll der Sachverhalt des Löschens von virtuellen Typen mit existierenden Referenzen im aktuellen Kontext untersucht werden.

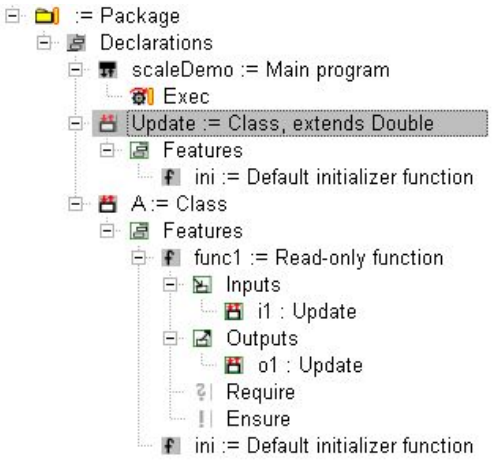


Abbildung 4.2: Lava Rename und Delete Refactoring

Abbildung 4.2 zeigt den Syntaxbaum eines Programms mit der Klasse *Update*, die in den zwei Funktionen *Input* und *Output* referenziert wird. Das Ändern des Namens bewirkt auch, dass der Name in den Funktionen geändert wird. Wird jedoch die Klasse gelöscht, so verbleibt der Baum in einem inkonsistenten Zustand. Das Ergebnis dieser Refaktorisierung kann in Abbildung 4.3 gesehen werden.

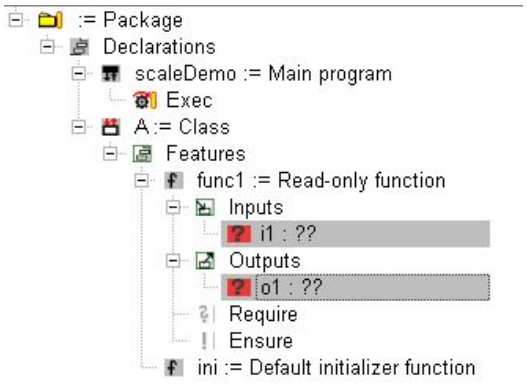


Abbildung 4.3: Lava Delete Refactoring

Die Eigenschaft der Umbenennung funktioniert auch für Funktionsaufrufe. Diese werden direkt konsistent auf dem Syntaxbaum ausgeführt. Das Hinzufügen oder Löschen von Parametern wird auch direkt auf den vorhandenen Referenzen vorgesehen. Durch das Hinzufügen werden neue Platzhalter in den Referenzen eingefügt, so dass der Baum zwar nicht in einem konsistenten Zustand ist, jedoch werden diese vor einer Ausführung dem Entwickler angezeigt, so dass die notwendigen Anpassungen vorgenommen werden können. Der Typ eines definierten Parameters kann nachträglich nicht geändert werden. Der Parameter muss zuerst gelöscht und dann erneut der Funktion zugefügt werden. Abbildung 4.4 veranschaulicht die Änderung der Definition der Prozedur und ihre Auswirkung auf eine vorhandene Referenz.

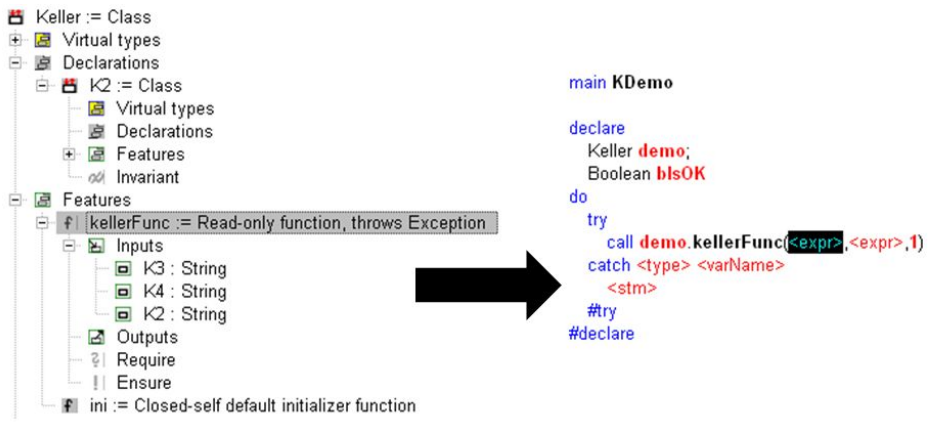


Abbildung 4.4: Erweiterung von Funktionsaufrufen

Das Löschen der Funktionsdefinition verursacht jedoch keinen Fehler, wodurch der Benutzer gewarnt wird dass es noch abhängige Referenzen gibt. Somit bietet die Lava Entwicklungsumgebung einen AST-basierten Entwicklungsansatz, allerdings ohne die Konsistenz eines echten Syntaxbaums. Der Editor unterstützt des Weiteren den Entwickler nicht durch vordefinierte Refaktorisierungen, wodurch zum Beispiel eine lokale Variable zu einem Feld innerhalb der Klasse umgewandelt wird. Bei der manuellen Durchführung dieser Schritte fällt auf, dass der Editor das Löschen der ursprünglichen Referenz bemerkt und keine neue Referenz auf Basis des definierten Felds herstellt. Durch eine Fehlermeldung *Broken Reference* wird der Entwickler jedoch über diesen Sachverhalt informiert. Abbildung 4.5 zeigt die Darstellung der fehlenden Referenz im Editor.

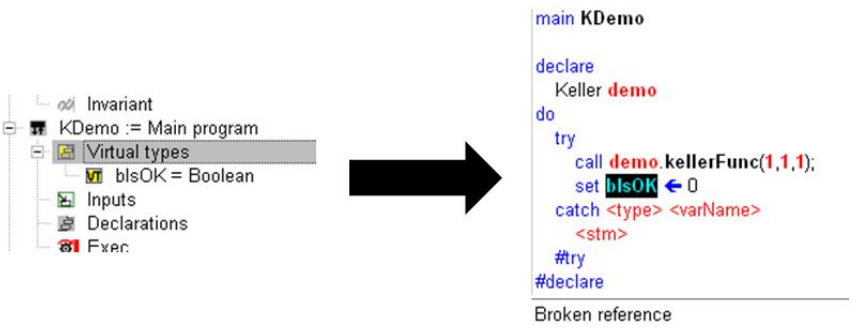


Abbildung 4.5: Fehlende Referenzen

Positiv hervorzuheben sind die bereits vorgestellten Sichten auf den Baum. Der Entwickler hat hierbei die Möglichkeiten des *Copy & Paste* aber auch des *Drag & Drop*. Auf diese Weise können sehr schnell ganze Strukturen innerhalb des Syntaxbaums kopiert oder verschoben werden. Durch diese Funktionalitäten und durch die Sichten entsteht eine übersichtlichere Darstellung des Quelltext als in einem textuellen Editor. Der Entwickler hat direkt die Möglichkeit zwischen Referenzen und Definitionen zu springen. LavaPE bietet nicht nur eine Unterstützung für die Ausführung von Nebenläufigkeiten und der Synchronisation, sondern verfügt auch über eine integrierte Datenbankkomponente, die den Programmierer bei der Erstellung von Applikationen unterstützt und Fehler bei der Erstellung der Querries vermeidet [GG00, Seite 12].

Zusammenfassend ist die Lava Entwicklungsumgebung für die Erstellung von objektorientierten Datenbankanwendungen durchaus geeignet. Die Erstellung von Methoden durch vordefinierte Programmkonstrukte ist jedoch noch optimierbar, da der User sehr viel Zeit dafür aufwenden muss innerhalb der Menüs und der Bedienoberflächen die entsprechenden Elemente auszusuchen. Des Weiteren bietet Lava außer der Umbenennung von Deklarationen und dem Ändern von Parametern keine Funktionalitäten für das Durchführen von Refaktorisierungen. Das Löschen von Elementen mit abhängigen Referenzen verletzt die Konsistenz des Syntaxbaums und der Entwickler wird vor der Durchführung dieser Schritte nicht auf die Konsistenzen aufmerksam gemacht.

4.3 Barista

4.3.1 Allgemeine Einführung in Barista

An der Carnegie Mellon Universität wurde das Basic Abstractions for Rapidly Implementing Structured Text-editing Applications (Barista) Framework entwickelt, welches bei der Erstellung von Strukturediteren unterstützen soll. Ziel von Barista ist es, die Vorteile des Schreibens von Code mit visuellen Präsentationstechniken zu verbinden um somit eine optimale Unterstützung des Entwicklers zu gewährleisten. Das Framework, welches in der Programmiersprache Citrus geschrieben wurde, bietet einen *Model View Controller* (MVC) Ansatz zur Realisierung der Editoren und die Unterstützung von *Drag & Drop* Features. Auf Basis dieses Framework wurden exemplarische Editoren für die Programmiersprache Citrus und Java entwickelt, die dem User verschiedene Bearbeitungsmöglichkeiten offerieren. Zum einen kann der User die Syntax direkt eingeben und der Editor erkennt während der Eingabe die selektierten Schlüsselwörter. Zum anderen es möglich auf Basis von Templates entsprechende Strukturen per Drag & Drop in den Editor hineinzuziehen. Diese Form der Editierung erinnert sehr stark an die Ansätze von Lava (siehe 4.2), wo der Programmierer ebenfalls die Quelltextfragmente aus einem Menü heraus auf der Oberfläche generiert. Somit bietet diese Form der Editierung bereits einen Vorteil gegenüber der Implementierung in Lava, da hier auch direkt via Code Vervollständigung auf der abstrakten Struktur der Programmiersprache gearbeitet werden kann und nicht nur über die Menüpunkte des Editors. Barista ist so konzipiert, dass es Editoren für jede Sprache generieren kann. Notwendige Voraussetzungen sind die Erstellung der zugrundeliegenden Datenstrukturen und eine API. Abbildung 4.6 zeigt

einen Ausschnitt eines prototypischen Editors, der auf Basis des Barista Framework entwickelt wurde und die Erstellung von Java Quellcode vereinfachen soll.

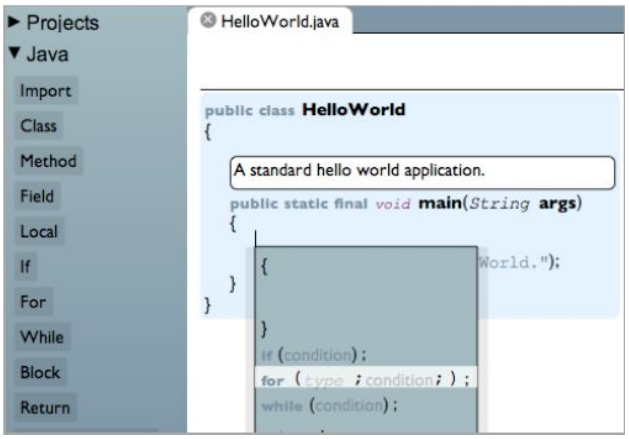


Abbildung 4.6: Barista Editor in Java (entnommen aus [AB06, Seite 3])

4.3.2 Erstellung von Editoren in Barista

Die folgenden Schritte wurden von Andrew und Brad [AB06, Seite 20] beschrieben, um Editoren für beliebige Programmiersprachen zu erzeugen. Im ersten Schritt muss der Entwickler die Grammatik der Sprache beschreiben. Diese wird benötigt, damit der Editor später auch textuelle Bearbeitungsmöglichkeiten anbieten kann. Hierbei werden in Citrus die Elemente der Sprache definiert. Hierzu müssen die neu erstellten Klassen von einer gemeinsamen Oberklasse *Structure* erben. Die auf diese Weise erstellten Klassen stellen das Modell der Sprache dar, für die ein Editor erstellt werden soll. Nach der Erstellung der Klassen zur Definition der Grammatik der Sprache können die Klassen zur Definition der Sichten realisiert werden. Hierbei erben alle Klassen von einer gemeinsamen Oberklasse *StructureView*. Durch das Erstellen von Subklassen für jedes Element der definierten Sprache entsteht ein konsistentes Model zwischen den definierten *Model* Klassen und den *View* Klassen [AB06, Seite 3]. Der hier beschriebene Sachverhalt wird durch die Abbildungen 4.7 und 4.8 verdeutlicht.

```
an IfStatement is a Structure that
   has IfKeyword   if
   has LeftParen   left
   has Expression  condition
   has RightParen  right
   has Statement   thenStatement
   has ElseKeyword else
   has Statement   elseStatement
.
```

Abbildung 4.7: If Statement im Modell (entnommen aus [AB06, Seite 3])

```
an IfStatementView is a StructureView that

 refs IfStatement model = ?
 has Real width <- (this rightmostChildsRight)
 has Real height <- (this tallestChildsHeight)
 has List<View> children = [
    (model.@if toView)
    (model.@left toView)
    (model.@condition toView)
    (model.@right toView)
    (model.@thenStatement)
    (an ElseView)
 ]
```

Abbildung 4.8: If Statement als View(entnommen aus [AB06, Seite 3])

Mit dem oben beschriebenen Erstellen von *Model* und *View* Klassen werden somit die Schlüsselwörter einer Sprache für einen AST-basierten Editor erstellt. Um jedoch einen vollständigen Editor entwickeln zu können müssen auch die Tokens der Zielsprache definiert werden. Diese Definition wird durch das Erstellen von Subklassen der Klassen *Token* und *TokenView* erstellt. Durch das Erstellen einer Subklasse der Klasse *Token* werden die Schlüsselwörter definiert. Durch die Subklassen von *TokenView* werden Größe, Form und Farbe der Statements definiert [AB06, Seite 4]. Abbildung 4.9 zeigt einen Screenshot der Projektwebsite von Barista und verdeutlicht die Bearbeitungsansicht und die zugrundeliegende Bearbeitungsstruktur in Form eines AST.

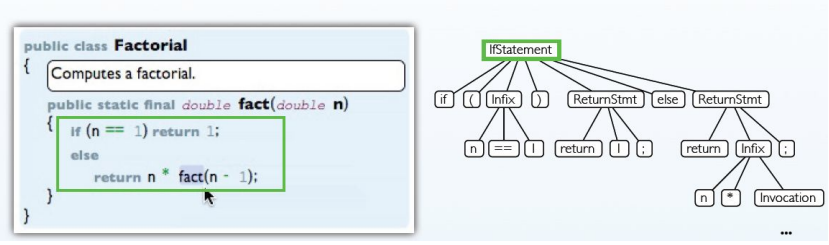


Abbildung 4.9: Screenshot (entnommen aus [AB06, Seite 27])

Der aufgezeigte Ansatz ist sehr interessant, da er versucht die AST-basierte Programmierung durch die Projektion einer textuellen Eingabeform zu verbinden. Es ist jedoch anzumerken, dass bisher keinerlei Referenzimplementierungen zu finden sind, die dieses Framework selbst benutzt. Das Framework an sich steht ebenfalls nicht zum Herunterladen zur Verfügung, wodurch eine weitergehende Evaluierung nicht möglich ist.

4.4 Alice

4.4.1 Allgemeine Einführung in Alice

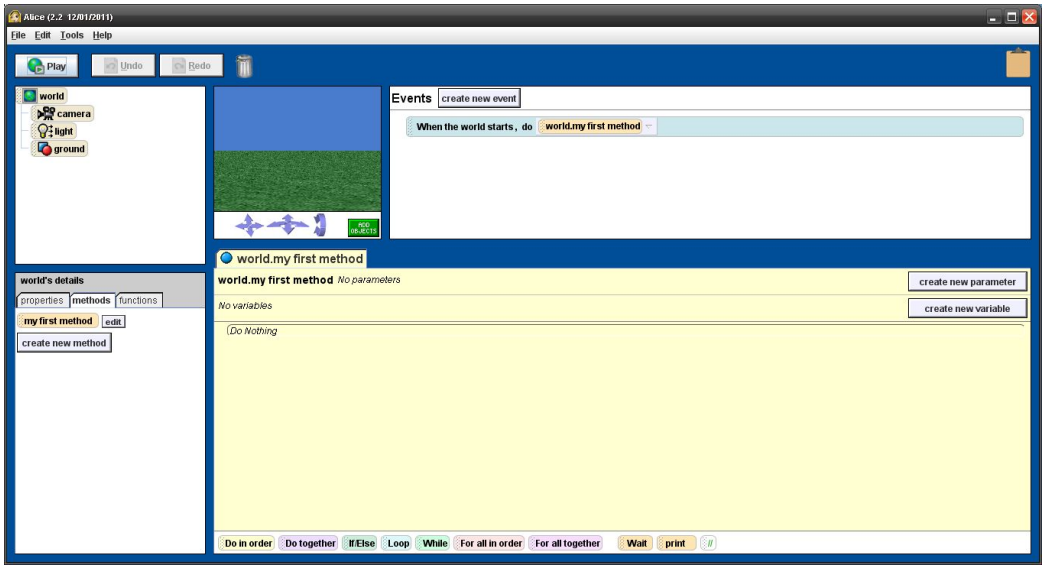


Abbildung 4.10: Alice Entwicklungsumgebung

Vor der Entwicklung des Barista Frameworks wurde an der Carnegie Mellon University das Alice System entwickelt. Der ursprüngliche Anwendungszweck dieser Software war das schnelle Entwickeln und Testen von dreidimensionalen Prototypen. Auf Basis einer objektorientierten Sprache ermöglicht Alice die einfache Erstellung von virtuellen Szenen, die direkt nach ihrer Erstellung auch zur Ausführung gebracht werden können [Pau+95]. Aktuell wird die Software für das Erlernen der objektorientierten Programmierung verwendet. Alice ermöglicht es hierbei interaktive Szenen und Filme zu erstellen, die über eine Community im Netz mit anderen Anwendern ausgetauscht werden können. Alice kann in einem Atemzug mit der Entwicklungsumge-

bung Scratch genannt werden, die ebenfalls in diesem Kapitel vorgestellt wird. Auch bei Alice handelt es sich um eine vollständig AST-basierte Entwicklungsumgebung, die in Abbildung 4.10 verdeutlicht wird. Sie ist derart aufgebaut, dass alle Facetten der objektorientierten Programmierung im Rahmen eines Fensters bearbeitet werden können. Auf der linken Seite im oberen Bereich der Entwicklungsumgebung werden die einzelnen Objekte, die im Rahmen eines neuen Programms verwendet werden sollen erstellt und bearbeitet. Objekte und ihre Eigenschaften können selektiert werden, wodurch in der unteren Hälfte der Entwicklungsumgebung ebenfalls auf der linken Seite die Details des selektierten Objekts angezeigt werden. Im Rahmen dieser Sicht können dann die entsprechenden Eigenschaften, Methoden und Funktionen der Objekte angezeigt und bearbeitet werden. Der eigentliche Entwicklungsbereich befindet sich auf der rechten Seite der Entwicklungsumgebung. Wird also eine Methode oder Funktion selektiert, so wird diese auf der rechten Seite eingeblendet. Dies wird in Abbildung 4.11 verdeutlicht.

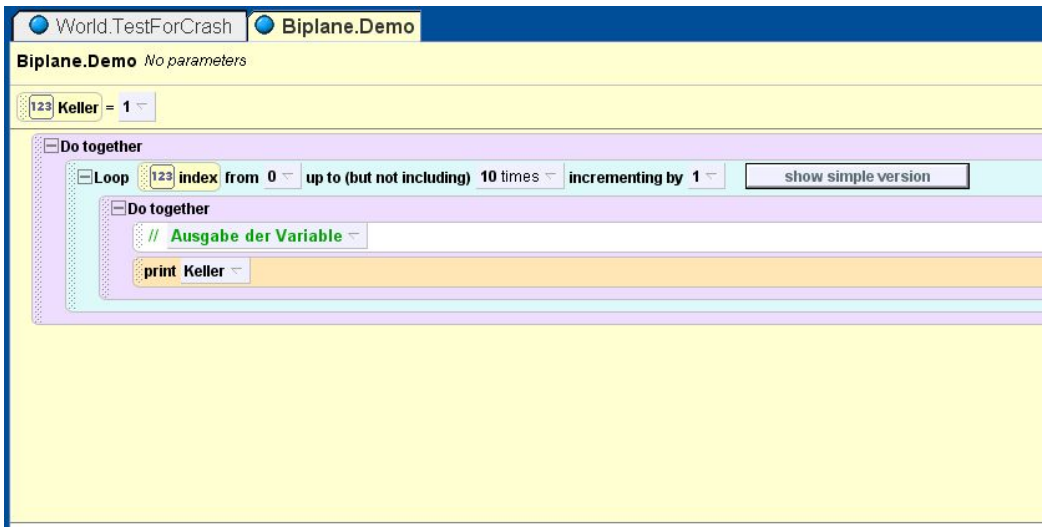


Abbildung 4.11: Alice Entwicklungssicht

4.4.2 Refaktorisierungen in Alice

Die Abbildung 4.11 zeigt den Entwicklungsbereich, in dem die eigentliche Ablauflogik innerhalb des Programms realisiert werden kann. Der Entwickler wird bei der Erstellung dahingehend unterstützt, dass die einzelnen Programmierkonstrukte wie zum Beispiel Schleifen und Bedingungen im unteren Bereich vorgegeben werden, so dass dieser sie nur noch in die Entwicklungsumgebung per *Drag & Drop* ziehen muss. Die Grafik zeigt, wie die einzelnen Elemente der Sprache ineinander geschachtelt werden. Die graphische Darstellung erinnert hier sehr an die von Programmablaufplänen. Durch die Verwendung des Schlüsselwortes *Set* lassen sich den einzelnen Eigenschaften der Objekte neue Werte zuweisen. Eine definierte Eigenschaft eines Objekts kann per *Drag & Drop* in die Bearbeitungssicht eingefügt werden. Die Entwicklungsumgebung erkennt diesen Vorgang und schlägt dem Entwickler verschiedene Aktionen vor. Zu diesen Aktionen gehört auch, dass der Eigenschaft ein neuer Wert zugewiesen werden kann. In Abbildung 4.12 wird gezeigt wie der Eigenschaft *Keller2* ein neuer Wert zugewiesen wird.

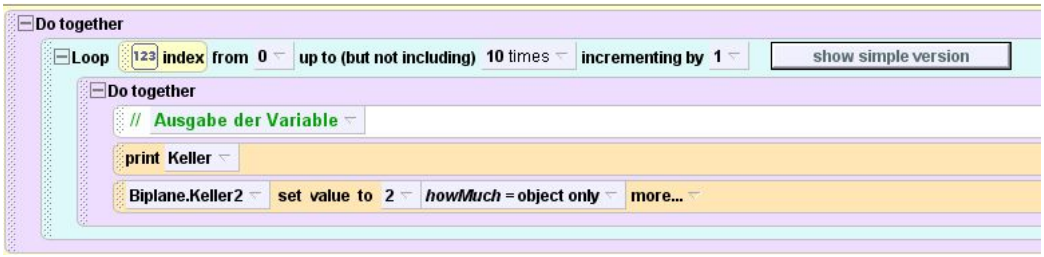


Abbildung 4.12: Alice Zuweisung von Werten

Alice bietet weitreichende Unterstützung für das Umbenennen von Eigenschaften. Objekte, Eigenschaften, Methoden, Funktionen und Variablen werden auf Basis ihrer Definition umbenannt und die Änderungen direkt auf allen referenzierten Stellen im Quellcode durchgeführt. Auf jedem dieser Objekte kann über das Kontextmenü eine entsprechende Umbenennung durchgeführt werden. Abbildung 4.13 zeigt die Umbenennung einer lokalen Variable innerhalb einer exemplarischen Methode.

Diese Umbenennungen können auch auf Objekten und ihren Eigenschaften und Funktionen durchgeführt werden. Abbildung 4.14 zeigt auf dass die Umbenennungen auch auf einem Objekt direkt durchgeführt werden können. Diese Umbenennungen wirken sich auf alle Referenzen aus und überführen das Objektmodell in einen konsistenten AST.

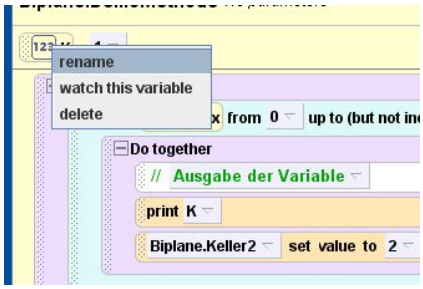


Abbildung 4.13: Alice Umbenennen

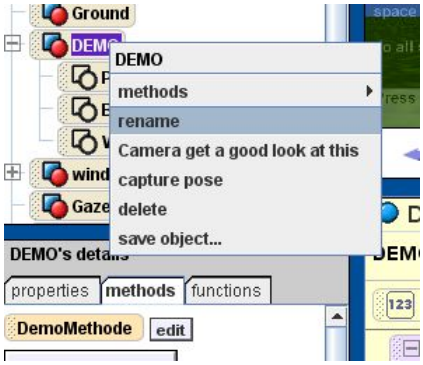


Abbildung 4.14: Alice Umbenennen eines Objekts

Alice unterstützt den Entwickler auch bei der Löschung. Wie in Abbildung 4.13 zu sehen, bietet das Kontextmenü auch eine Möglichkeit zum Löschen von Variablen, Eigenschaften, Methoden und Objekten. Die Programmierung auf einem AST erfordert, dass der Baum in sich konsistent ist. Aus diesem Grund validiert Alice auch die Löschanforderung mit dem Modell um herauszufinden, ob die Durchführung dieser Operation Inkonsistenzen verursachen würde. Ist dies der Fall, wird der Entwickler durch eine Fehlermeldung auf diesen Sachverhalt hingewiesen und die Operation kann abgebrochen werden.

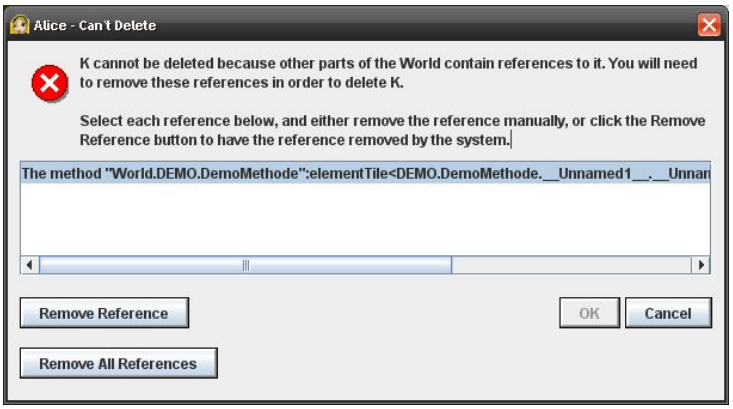


Abbildung 4.15: Anzeigen der abhängigen Referenzen

Abbildung 4.15 zeigt den Dialog der angezeigt wird, falls das Löschen eines Elements den AST des Programms in einen inkonsistenten Zustand bringen würde. Der Entwickler kann sich die Referenzen anzeigen lassen und entscheiden ob die ganze Operation abgebrochen wird, oder ob lediglich einzelne Referenzen gelöscht werden sollen. Über die Funktion *Remove all References* können alle Referenzen gelöscht werden. Erst wenn keine Konflikte mehr auftreten, kann über die *OK* Schaltfläche das Element aus dem Editor gelöscht werden.

Die Kontextmenüs der einzelnen Sprachelemente bieten keine weiteren direkten Refaktorisierungsmuster an. Die Veränderung des Gültigkeitsbereich einer Variablen durch die Refaktorisierungen *Move Temp to Field* und *Move Field through Temp* müssen also durch eine Abfolge von Einzeloperationen realisiert werden. Die Refaktorisierung *Move Temp to Field* würde also durch das Erstellen einer gleichnamigen Eigenschaft, dem Löschen aller Referenzen auf die Variable sowie der Variable selbst und der neuen Zuweisung des Feldes realisiert werden können. Abbildung 4.16 zeigt, wie die lokale Variable K durch eine Eigenschaft K ersetzt wurde.

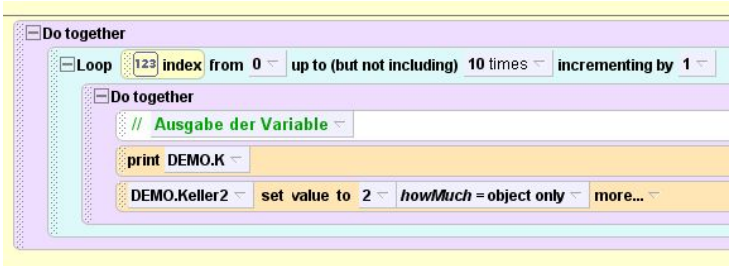


Abbildung 4.16: Alice Move Temp to Field

Die gleichen Schritte müssen somit auch bei der umgekehrten Refaktorisierung durchgeführt werden, wenn ein Feld zu einer lokalen Variable umgewandelt werden muss. Da die Objekte in Alice nicht voneinander erben können, werden an dieser Stelle die weiteren Refaktorisierungen, die im Rahmen dieser Arbeit analysiert werden sollen, nicht weiter betrachtet.

4.5 Visual Functional Programming Environment

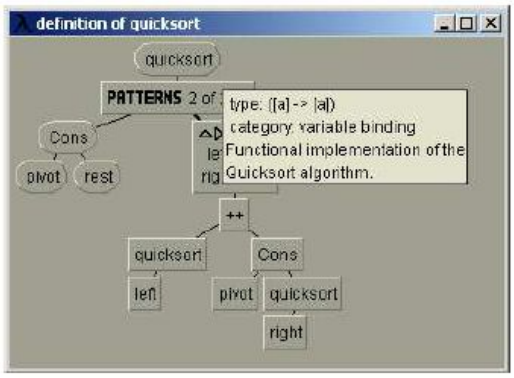


Abbildung 4.17: Darstellung VFPE Editor (entnommen aus [Kel02, Seite 143])

Im Rahmen seiner Dissertation [Kel02] hat Joel Kelso einen prototypischen Editor für eine funktionale Sprache entwickelt. Sein Ziel war es, auf Basis des Syntaxbaums der Sprache einen Editor zu programmieren, der gerade bei der Erstellung von Zuweisungen, Bedingungen und Schleifen im Rahmen

einer einfachen Baumstruktur eingesetzt werden kann. Die von ihm entwickelte Programmierumgebung enthält dabei eine graphische Entwicklungsumgebung, bei der es nicht möglich ist, dass der Programmierer syntaktisch inkorrekte Elemente erstellen kann. Die Informationen zu einem verwendeten Typ sind an jeder Stelle innerhalb des Baums für den Programmierer sichtbar [Kel02, Abstract]. Joel Kelso verwendet bei der Implementierung eine eigene funktionale Sprache, um auf dieser Basis ein einfaches Objektmodell zu erstellen. Der Editor bietet des Weiteren ein Interface, so dass dieser auch Quellcode erzeugen kann. Ein Interpreter zur Ausführung des modellierten AST wurde ebenfalls implementiert [Kel02, Seite 27]. Ein besonderes Ziel in der beschriebenen Arbeit war die Erstellung eines einfach zu bedienenden User Interfaces. Es sollte vor allem auch für Anfänger in der Programmierung leicht zu erlernen sein. Das User Interface soll besonders durch seine Einfachheit und Konsistenz aufwarten, so dass eine Vielzahl von zusätzlichen Funktionalitäten nicht dem eigentlichen Zweck der Validierung der AST-basierten Programmierung entgegensteht. Der entwickelte Editor stellt die einzelnen Knoten innerhalb des AST auch als graphische Baumstruktur dar. Somit wird jeder Ausdruck der Sprache auch als Knoten innerhalb der Baumstruktur angezeigt. Untergeordnete Elemente werden auch unterhalb eines entsprechenden Knotens angezeigt. Variablen und Konstanten werden als Blätter innerhalb der Baumstruktur dargestellt [Kel02, Seite 70].

Wie bereits beschrieben ist die Darstellung des VFPE Editors durch einen Syntaxbaum realisiert. In der Arbeit von Joel Kelso werden jedoch auch unterschiedliche Möglichkeiten der Darstellung angesprochen. Dies basiert vor allem auf der bereits beschriebenen Tatsache, dass bei zunehmender Komplexität des Graphen auch der Vorteil der verbesserten Übersichtlichkeit verloren geht. In diesem Kontext werden verschiedene Formen der Darstellung angesprochen, die zur Eingrenzung der Komplexität möglich sind. Die Arbeit geht hierbei vor allem auf die dynamische graphische Darstellung als Baum ein, schlägt jedoch auch Konzepte vor, die der Darstellung von Programmablaufplänen ähneln [Kel02, Seite 80].

Im Editor realisiert wurde jedoch die graphische Darstellung als Syntaxbaum auf einer Leinwand. Abbildung 4.17 verdeutlicht die Form der Realisierung. Die verwendete Sprache basiert auf einer Untermenge der Programmiersprache Haskell. Der Editor ist ebenfalls in der Lage, das Modell in die Codestruktur von Haskell umzuwandeln um somit die Begrenzungen die der Editor durch seine Funktionalitäten bietet aufzuheben. Im Editor werden unterschiedliche Symbole verwendet. Es gibt Platzhalter, Literale, Variablen,

Ausdrücke, globale und lokale Definitionen, Datentypen, einfache und komplexe konditionale Ausdrücke, Listen sowie Variablen und Literalbindungen. Aus diesem Satz von Sprachelementen lässt sich Programmcode erstellen.

Die Grammatik der Sprache die hinter dem VFPE Editor steht ist wie folgt strukturiert. Die Hauptklasse der Grammatik heißt *Syntax*. Die *Syntax* Klasse stellt hierbei das Kernelement dar, von der alle Klassen erben. Jedes Element, bei dem es sich nicht um ein Leerzeichen oder ein Kommentar handelt ist also ein *Syntax* Element. Eine weitere Klasse sind die *Value expressions*. Bei jedem Knoten innerhalb des Syntaxbaums, der einen Wert aufnehmen kann, handelt es sich um eine *Value expression*. Nach Kelso [Kel02, Seite 101] sind dies Objekte, die im Rahmen der Bearbeitung kopiert, evaluiert und gespeichert werden können. Von den *Value expression* erben die *Abstraction expressions*. Diese Typen unterscheiden sich durch die zusätzliche Aufnahme von Beziehungen, sogenannten *Bindings*. *Bindings* stellen die Beziehung zwischen Name und Werten her, um diese im Rahmen des Editors abzubilden. Dieser Typ ist direkt sichtbar bei der Programmierung mit VFPE, da dieser zum Beispiel für die Definition von Variablen im Editor verwendet wird [Kel02, Seite 103]. Darüber hinaus gibt es noch die *Application Expressions* zur Darstellung von Sets, Knoten und Listen von Elementen innerhalb der Entwicklungsumgebung. Bei den hier vorgestellten Klassen handelt es sich um die Basisklassen, von denen alle weiteren Klassen innerhalb der Implementierung des VFPE Editors erben.

Der Editor bietet eine Vielzahl von Funktionalitäten mit denen die Komposition der beschriebenen Elemente einfacher zu gestalten ist. Der *VFPE Editor* ist ein graphischer Editor, der sich durch standardisierte Elemente mit Schaltflächen, Listen und Selektionen auszeichnet, aber auch *Drag & Drop* Operationen zulässt. Der Aufbau des Editors gliedert sich in zwei Sichten. Die erste Sicht stellt die verschiedenen Schaltflächen zur Verfügung, mit denen die oben beschriebenen Sprachelemente in den aktuellen Syntaxbaum integriert werden können. Die zweite Sicht ist die graphische Darstellung des aktuellen Programmcodes auf Basis des Syntaxbaums. Hierbei reagiert der Syntaxbaum sensitiv im Kontext, was dazu führt, dass nur jeweils die möglichen Sprachelemente sensitiv geschaltet sind. Somit verhindert der Editor direkt die Integration von Elementen, die der Grammatik der Sprache widersprechen. Der Editor bietet auch eine Form der textuellen Repräsentation an. Diese ist jedoch lediglich zur Ansicht gedacht und wird als Einblendung auf dem selektierten Element angezeigt. Die einzelnen Knoten innerhalb des Baums können ein- und ausgeklappt werden, so dass eine bessere Übersichtlichkeit entsteht wenn der beschriebene Code eine ent-

sprechende Komplexität erreicht hat. Referenzen innerhalb des Syntaxbaums sowie zwischen der Definition von Variablen und ihrer Verwendung können ebenfalls dargestellt werden.

Wie bereits angesprochen findet die Darstellung der einzelnen Knoten innerhalb des VFPE Editors über eine baumartige Struktur statt. Auf Basis der Dissertation von Kelso werden nun im Folgenden die einzelnen Elemente mit denen ein Entwickler arbeiten kann vorgestellt. Die weiter oben vorgestellten *Value expressions* werden innerhalb der graphischen Repräsentation durch Fragezeichen abgebildet. Dies bedeutet, dass an dieser Stelle durch den Entwickler noch eine Wertzuweisung vorgenommen werden kann. Abbildung 4.18 zeigt den Platzhalter in seiner Darstellung innerhalb des Editors.

Abbildung 4.18: Platzhalter in VFPE (entnommen aus [Kel02, Seite 105])

Die Platzhalter können durch Wertzuweisungen gefüllt werden. Hierdurch können innerhalb des Editors Literale dargestellt werden. Abbildung 4.19 zeigt die Darstellung eines Literals in VFPE.

Abbildung 4.19: Literale in VFPE (entnommen aus [Kel02, Seite 108])

Ähnlich wie die bereits vorgestellten Literale, werden auch die Variablen innerhalb des Editors dargestellt. Abbildung 4.20 zeigt die Darstellung einer Variable innerhalb des Editors. Nach Kelso [Kel02, Seite 109] stehen alle Variablen innerhalb des Editors für Referenzen auf definierte Werte.

Abbildung 4.20: Variablen in VFPE (entnommen aus [Kel02, Seite 109])

Wie weiter oben erwähnt übernimmt das Kernelement *Application Expressions* die Realisierung und Darstellung von Funktionsaufrufen. Eine *Application Expression* nimmt als Kindelemente wiederum *Value Expressions* als

Übergabeparameter auf. Funktionsaufrufe können in VFPE auf zwei verschiedene Weisen innerhalb des Graphen dargestellt werden. Abbildung 4.21 und 4.22 zeigen die verschiedenen Darstellungsformen.

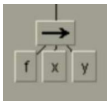


Abbildung 4.21: Darstellung einer Funktion in VFPE (entnommen aus [Kel02, Seite 111])

Bei den hier vorgestellten graphischen Repräsentationen handelt es sich um einen Ausschnitt der möglichen graphischen Elemente und ihrer Implementierung in VFPE. Darüber hinaus gibt es eine Vielzahl weiterer Elemente zur Realisierung der folgenden Konstrukte innerhalb des Editors.

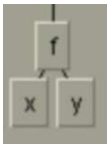


Abbildung 4.22: Weitere Darstellungsform einer Funktion in VFPE (entnommen aus [Kel02, Seite 111])

Die folgende Liste zeigt verschiedene Datentypen, die im Rahmen der Realisierung des VFPE Editors realisiert wurden:

- *Let* (Lokale und Globale Definitionen)
- *Datatype* (Definition von Datentypen)
- *Prelude* (Import von Modulen)
- *Conditional* (If / Else Anweisungen)
- *Guard-Set* (Geschützte Cond. Ausdrücke)
- *List* (Listen von Literalen)

Der Editor bietet auch die Möglichkeit des Verschiebens von Ausdrücken mit einer *Drag & Drop* Funktionalität. In diesem Kontext wird auch die Konsistenz des AST Editors berücksichtigt, da das Verschieben von Elementen durch den Editor auch abgelehnt werden kann [Kel02, Seite 140 - 150]. Der Editor steht leider nicht zum Download zur Verfügung, was eine weitergehende Analyse erschwert.

4.6 Scratch

4.6.1 Allgemeine Einführung in Scratch

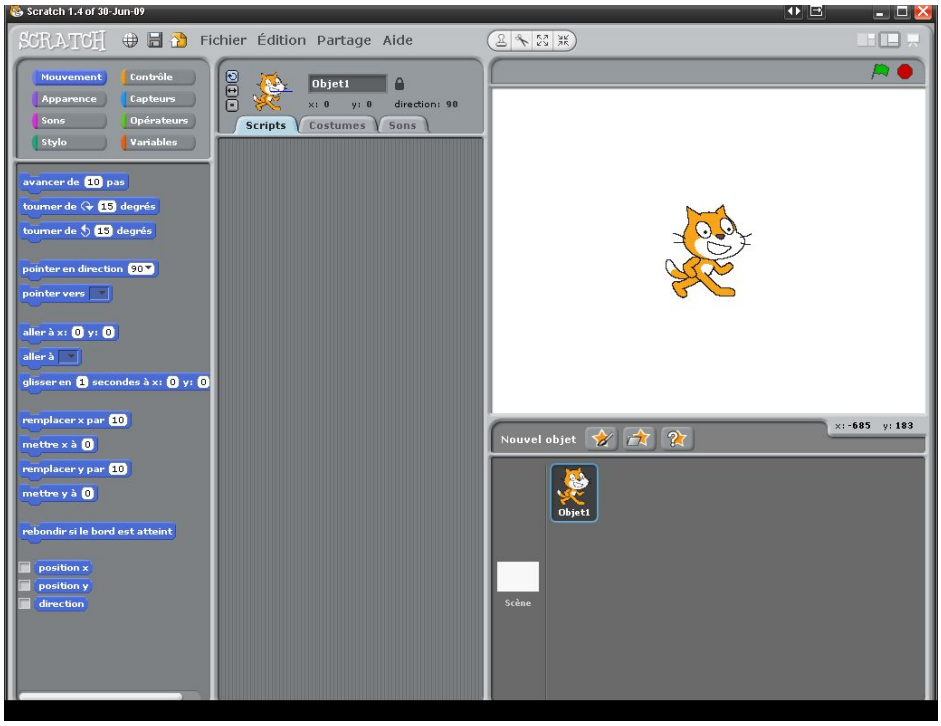


Abbildung 4.23: Scratch

Scratch wurde am Massachusetts Institute of Technology (MIT) entwickelt und stellt eine Entwicklungsumgebung mit Programmiersprache dar, die vor allem auf das spielerische Erlernen von Themen der Mathematik und Informatik abzielt. Das Projekt wurde im Jahr 2003 initialisiert und ab 2008 öffentlich verfügbar gemacht. Die Entwicklungsumgebung ist in 50 verschiedenen Sprachen verfügbar [Mal10, Seite 1]. Die Elemente der zugrundeliegen-

den Sprache werden durch graphische Elemente angezeigt, die durch *Drag & Drop* auf eine Leinwand gezogen werden können. Hierbei wird ein Prinzip wie bei dem Kinderspielzeug LEGO angewandt, da diese Elemente Schnittstellen besitzen, so dass sie mit anderen Elementen kombinierbar sind. Scratch wird aktuell immer häufiger in Schulen eingesetzt bei Kindern zwischen 8 und 16 Jahren. Durch die graphischen Elemente sollen vor allem Schüler angesprochen werden, die bisher keine Erfahrungen in der Programmierung gesammelt haben. Ziel ist die Komposition von Filmen, Texten und Audioaufnahmen. Die erstellten Anwendungen können als XML serialisiert, ausgetauscht und erweitert werden. Hierzu dient auch eine Community, die zum Zwecke des Austausches von Ideen unterhalten wird, bei der junge Menschen gemeinsam an Lösungen arbeiten können [MIT]. Scratch kann sehr gut mit der Entwicklungsumgebung Alice verglichen werden, da beide das spielerische Erlernen der Programmierung ermöglichen.

Wie in Abbildung 4.23 bereits zu erkennen ist, handelt es sich bei Scratch um eine Entwicklungsumgebung die innerhalb eines Fensters eine Palette mit Elementen, einen Bereich zur Erstellung der Scripte und einer Vorschau besitzt. Die Palette auf der rechten Seite bietet verschiedene Funktionsgruppen an, die im Kontext von Scratch verwendet werden können. Realisiert werden können Bewegungsabläufe, das Aussehen der Figuren und Zeichnungen, Geräusche und Soundeinstellungen sowie die Bewegungseinstellungen. Jede dieser Funktionsgruppen bietet eine Vielzahl von Funktionen an, die miteinander kombiniert werden können. Hierzu wird der Scriptbereich verwendet. Bei der AST-basierten Programmierung soll der AST in sich immer konsistent sein. Dieses Prinzip findet sich auch in Scratch wieder, da die einzelnen Elemente wie LEGO Bausteine aneinander andocken müssen. Somit können auch nur die Funktionen miteinander kombiniert werden, die auch im aktuellen Kontext verwendet werden. Der Editor verhindert auch, dass der User Eingaben tätigt die einen Fehler bei der Ausführung auslösen könnten. Zum Beispiel können keine alphabetischen Zeichen in numerischen Feldern eingegeben werden. Die einzelnen Funktionen, die auf die Leinwand gezogen werden, können direkt ausgeführt werden. Somit ist es zu jedem Zeitpunkt möglich auch einzelne Elemente auszuführen, ohne das ganze Script ausführen zu müssen. Es gibt keine Kompilierung, das erstellte Script ist zu jeder Zeit direkt ausführbar. Die einzelnen Funktionen, die zum Script hinzugefügt werden können, sind bereits mit initialen Parametern initialisiert. Somit kann direkt die hinzugefügte Funktion getestet werden. Bei der Definition von Variablen ermöglicht Scratch die Definition von Monitoren. Hierbei kann überwacht werden wie sich der Zustand der Variable während der Scriptausführung ändert. Definierte Variablen können direkt auf die ein-

zelnen Funktionen gezogen werden um dort an die Stelle mit den initialen Platzhaltern gesetzt zu werden. In diesem Kontext fällt auf, dass eine Variable hinsichtlich ihres Namens nicht mehr geändert werden kann. Ein Refactoring ist somit in Scratch nicht möglich. Ebenfalls bietet der Editor keine Prüfung an, wenn die verwendete Variable nicht mehr existiert. Eine Variablendefinition lässt sich löschen ohne dass ein Fehler ausgelöst wird. Scratch unterstützt die Datentypen Zeichenketten, Numerische Werte und Boolesche Werte. Die Syntax der zugrundeliegenden Sprache ist einfach gehalten und die Anzahl der möglichen Funktionen ist begrenzt. Die Syntax bietet als Basistypen der Sprache Kontrollstrukturen, Kommandos, Statements und Ausdrücke an. Kommandoblöcke werden gekennzeichnet, sodass sie an der Ober- und Unterseite eine Einkerbung respektive einen Vorsprung besitzen. Durch diese wird es möglich, dass mehrere Kommandos in Ketten miteinander verknüpft werden können. Funktionen geben Rückgabewerte, die in Kommandos integriert werden können. Auch das Konzept der *Event Handler* ist definiert, so dass diesen Triggern Kommandos zugeordnet werden. Diese besitzen auch wieder einen kleinen Vorsprung, so dass an dieser Stelle die Kommandos andocken können. Dies gilt auch für Kontrollstrukturen, da diesen auch Bedingungen zugewiesen werden können. Die Kontrollstrukturen besitzen eine Aussparung, so dass Funktionen als Bedingungen eingefügt werden können. In Scratch gibt es keine Prozeduren, jedoch sollen nach Malony [Mal10, Seite 12] Prozeduren in kommenden Versionen wieder eingefügt werden, da dies Anwender dazu befähigen würde ihre eigenen Blöcke innerhalb des Editors zu erstellen.

4.6.2 Refaktorisierungen in Scratch

Die Entwicklungsumgebung ist sehr übersichtlich gestaltet und ermöglicht einem jungen Entwickler einen direkten Einstieg. Die Laufzeitumgebung erfordert keinen Kompilierungsprozess. Die Realisierung findet direkt auf dem Objektmodell der Sprache statt und verhindert durch den implementierten Editor die Eingabe von Syntaxfehlern. User können somit schnell Ergebnisse erzielen ohne sich lange einarbeiten zu müssen. Die Usability ist durch die Integration von *Drag & Drop* Operationen sehr leicht verständlich. Refaktorisierungen sind in Scratch nicht implementiert. Es ist zum Beispiel nicht möglich eine Variable die definiert wurde umzubenennen. Eine Variable die definiert und innerhalb eines Script verwendet wird, kann gelöscht werden ohne dass die Referenzen innerhalb des Scripts durch diesen Vorgang beeinflusst werden. Somit lässt sich feststellen, dass in der Entwicklungsumgebung keine Möglichkeiten definiert sind um Refaktorisierungen vorzunehmen.

4.7 Elements for Smalltalk

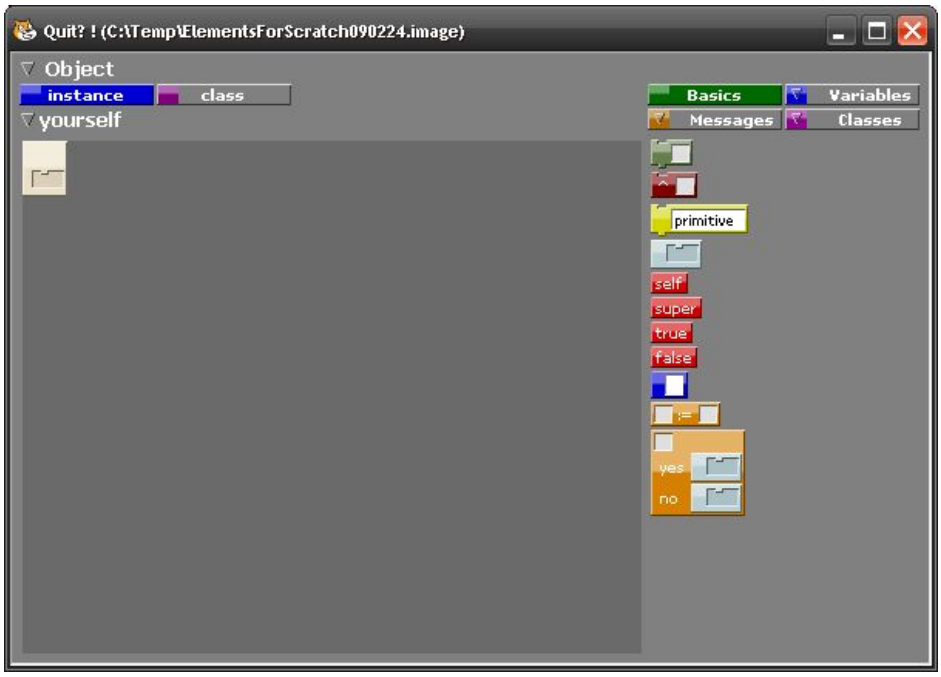


Abbildung 4.24: Elements for Smalltalk

4.7.1 Allgemeine Einführung in Elements

Elements for Smalltalk ist ein prototypischer Editor für die Programmiersprache Smalltalk. In Abbildung 4.24 ist die Oberfläche des Editors zu sehen. Die Implementierung dieses Editors basiert auf der Entwicklungsumgebung von Scratch in 4.6 und bietet somit ähnliche Bearbeitungsmöglichkeiten. Die prototypische Implementierung wurde von Jens Mönig [Mön09] implementiert und steht auch zum Download zur Verfügung. Auch bei dieser Implementierung liegt ähnlich wie bei Scratch der Anwendungszweck in der erleichterten Erstellung von Software. Ziel des Autors war es zu beweisen, dass das Konzept des spielerischen Erlernens einer Programmiersprache auch auf die vollständige Anwendungsentwicklung portierbar ist. Zu diesem Zweck wurde in Elements for Smalltalk eine Untermenge der Sprache Smalltalk implementiert, um herauszufinden, ob diese Form des Programmierens auch bei der

praktischen Softwareentwicklung verwendet werden kann [Mön09, Seite 1]. Die implementierte Untermenge von Smalltalk besteht aus sieben verschiedenen Elementen. Abbildung 4.25 zeigt die verschiedenen Elemente. Hierbei handelt es sich um:

1. *Literale*: Einfache Zeichenketten
2. *Closure*: Abschlüsse von Anweisungen
3. *Steps*: Arbeitsschritte
4. *Primitives*: Primitive Werte
5. *Answers*: Rückgabewerte.

Abbildung 4.25: Syntaxelemente für Elements

Nach Mönig ([Mön09, Seite 1]) handelt es sich bei den ersten drei Elementen innerhalb der Abbildung 4.25 um Variationen von Objekten und bei den letzten drei Elementen handelt es sich im Grundsatz um Sequenzen. Ein *Closure* wird als Zwischenstück zwischen den internen Sequenzen und den externen Objekten angesehen. Die Bedienung der vorgestellten Elemente lehnt sich an die Benutzerführung in Scratch an. Die einzelnen Elemente können ineinander geschachtelt oder in Sequenzen aneinander gereiht werden. Zusätzlich zu den Funktionalitäten, die bereits in Scratch implementiert wurden, kann der Entwickler innerhalb der Umgebung Elements noch über ein Kontextmenü verschiedene Operationen auf Basis eines selektierten Syntaxelements auslösen. Jedes im Quelltext selektierte Element bietet die Möglichkeit, dass der entsprechende Code angezeigt werden kann, aber auch dass das entsprechende Statement ad hoc evaluiert und das Ergebnis angezeigt werden kann. Für das schnelle Evaluieren und die Anzeige des Zwischenergebnisses ist die

Funktion *show result* verantwortlich. Die Entwicklungsumgebung ist in drei verschiedene Bereiche eingeteilt. Am oberen Ende befindet sich der Header Bereich. Auf der rechten Seite befindet sich die Palette. Die Palette zeigt die Grundelemente an und bietet die Möglichkeit durch Funktionen im Header Klassen, Variablen und Nachrichten anzuzeigen. Im linken Bereich befindet sich der eigentliche Editor auf dem die Programme erstellt werden können. Beim Speichern versucht der Compiler den vorhandenen Syntaxbaum direkt zu kompilieren und den entsprechenden Bytecode zu speichern. Falls dies zu Fehlern führt, so werden diese dem Entwickler direkt im Speichern-Dialog angezeigt. Abbildung 4.26 zeigt das Auftreten der Fehlermeldungen in diesem Zusammenhang.

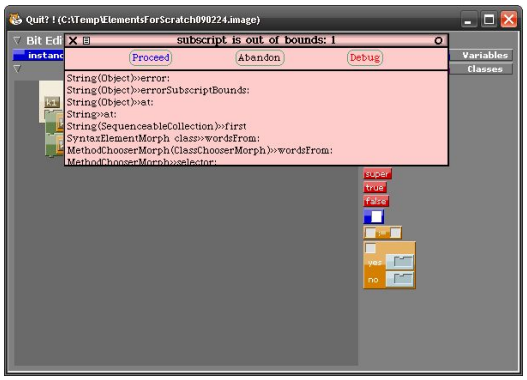


Abbildung 4.26: Kompilieren in Elements

Innerhalb von Elements können neue Methoden und auch Klassen definiert werden. Neue Methoden können nach Mönig [Mön09, Seite 5] erzeugt werden, in dem Methoden zu einer bereits existierenden Klasse hinzugefügt werden. Abbildung 4.27 zeigt die verschiedenen Schritte.

Abbildung 4.27: Erstellen von Methoden [Mön09, Seite 5]

Neben der Erstellung von Methoden können auch Klassen erstellt werden.

Hierzu muss der Entwickler eine geeignete Oberklasse aussuchen auf deren Basis er dann eine entsprechende Subklasse generiert. Innerhalb dieser Subklasse kann der Entwickler dann eigene Variablen und Methoden definieren. Abbildung 4.28 zeigt das Erstellen einer exemplarischen Subklasse von *String*, die eine Instanzvariable besitzt, welche im Rahmen einer Methode ausgegeben wird.

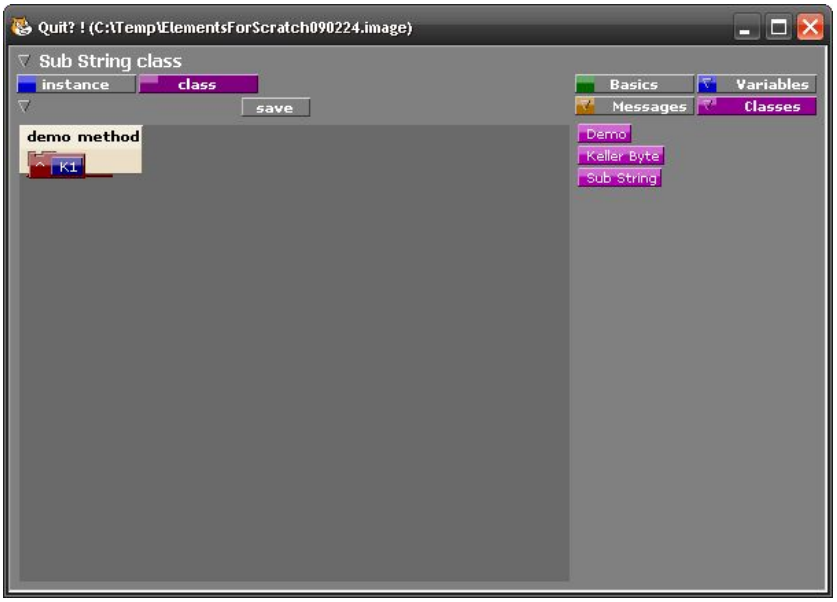


Abbildung 4.28: Erstellen einer Klasse

4.7.2 Refaktorisierungen in Elements

Ähnlich wie Scratch bietet Elements for Smalltalk keine direkte Unterstützung von Refaktorisierungen. Es können jedoch Variablen, Methoden und Klassen nachträglich umbenannt werden. Der getestete Prototyp ließ es leider nicht zu herauszufinden, ob das Löschen der Definition einer Variablen einen entsprechenden Hinweis ausgibt. Im Rahmen des Tests sind verschiedene Fehlermeldungen aufgetreten, deren genauer Hintergrund nicht analysiert werden konnte. Da es sich hier um eine Implementierung auf Basis von Scratch und um einen Prototypen handelt, der bisher keine weitere Verbreitung gefunden hat, werden weitere Refaktorisierungen in diesem Zusammenhang nicht analysiert.

4.8 Star Logo und das Open Blocks Framework

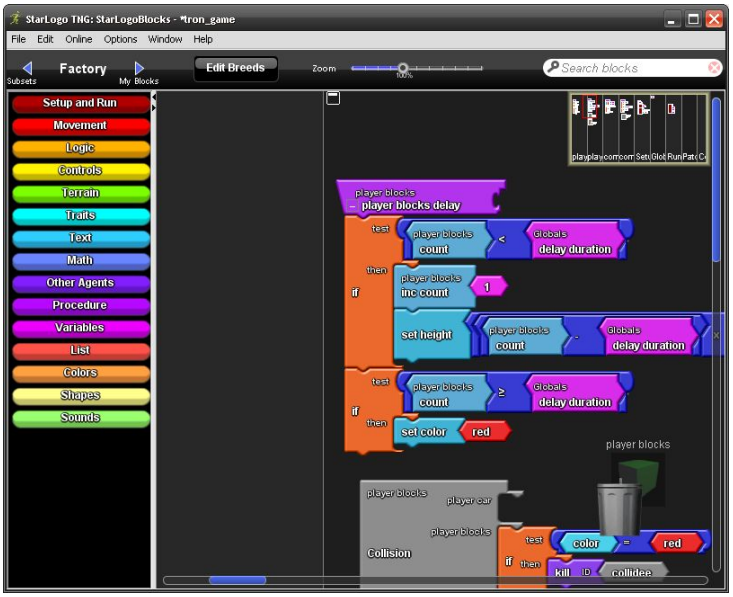


Abbildung 4.29: Star Logo TNG

4.8.1 Allgemeine Einführung in Star Logo

Bei Star Logo The Next Generation (TNG) handelt es sich um eine graphische Entwicklungsumgebung, die besonders für Lehrer und Studenten gedacht ist um dreidimensionale Modelle, Spiele oder Simulationen erstellen zu können. Die Software stammt aus dem MIT und steht für verschiedene Plattformen zur Verfügung [Log06, Seite 1]. Der Entwickler kann in Star Logo einzelne Bausteine auf die Leinwand ziehen. Die Bausteine sind ähnlich wie in der Scratch Umgebung und in der Erweiterung für Smalltalk durch bestimmte Formen realisiert. Abbildung 4.29 zeigt den Editor mit exemplarischen Sprachkonstrukten. Nur Formen, die gemeinsame Schnittstellen haben können auch miteinander verbunden werden. Die Programmierung ähnelt also sehr dem Zusammenfügen von LEGO Bausteinen. Die Syntax der zugrundeliegenden Sprache ist also durch die Formen der graphischen Elemente sichtbar [Roq07, Seite 9].

Ähnlich wie bei Scratch oder Alice lassen sich die in diesen Entwicklungsprogrammen erstellten Anwendungen nicht auf spezifische Problemstellungen anwenden. Die Anwendungen sind also auf einen bestimmten Kontext festgelegt, welcher sich bei den angesprochenen Entwicklungsumgebungen um das spielerische Erlernen der objektorientierten Sprache oder der einfachen graphischen Simulation handelt. Star Logo TNG basiert auf einer älteren Version Star Logo, bei der die Erstellung der Anwendungen noch durch die textuelle Programmierung realisiert wurde. Bei Star Logo TNG handelt es sich um eine Erweiterung, die konkret auf das User Interface bezogen ist. Im Hintergrund dieser Applikation wird der Quellcode erstellt. Es handelt sich somit nicht um einen AST-basierten Editor, wobei das graphische User Interface jedoch durch seine syntaktischen und semantischen Prüfungen die direkte Erstellung eines AST ermöglichen würde.

4.8.2 Refaktorisierungen in Star Logo

Refaktorisierungen werden in Star Logo nicht direkt unterstützt. Es besteht die Möglichkeit des Umbenennens von Prozeduren und Variablen und dies wirkt sich direkt auf alle Referenzen innerhalb des aktuellen Programms aus. Es lassen sich diese aber auch ohne Konsistenzprüfung löschen. Erst zur Ausführungszeit wird der Entwickler dann über die Inkonsistenzen informiert.

4.8.3 Open Blocks Framework

Das prototypische Open Blocks Framework entstand im Rahmen der Master Thesis von Ricaroso Vallarta Roque [Roq07] am MIT. Ziel seiner Arbeit war es auf Basis von Star Logo ein Framework zu erstellen, welches die die Vorteile der blockbasierten Programmierung in die professionelle Anwendungsentwicklung überführt. Prinzipiell richtet sich das Framework an Anwendungsentwickler und an Entwickler von domänenspezifischen Sprachen. Als User Interface soll die blockorientierte Programmierung angewandt werden, die bereits in Star Logo TNG eingesetzt wurde. Das Framework bedient sich einer Language Definition Datei. Auf Basis dieser Datei, die in einem XML-Dialekt gespeichert wird, werden die einzelnen Blöcke und Elemente dargestellt. Der Entwickler einer Sprache kann also durch die Definition einer XML-Datei seine eigene Sprache auf Basis des Frameworks editieren. Die erstellten Programme werden dann in Form einer XML-Datei abgespeichert. Aktuell gibt es für das Framework keine Compiler oder eine Laufzeitumgebung [Roq07, Seite 32 - 41]. Die XML-basierte Struktur könnte jedoch in einen Syntaxbaum überführt werden, um daraus im Rahmen eines Kompilierungsprozesses Maschinencode oder Bytecode zu erstellen. Das Open Blocks Framework wendet also Konzepte der AST-basierten Programmierung an. Der Prototyp steht nicht zum Download zur Verfügung und kann im Kontext dieser Arbeit nicht weiter hinsichtlich der Fähigkeit zur Durchführung von Refaktorisierungen analysiert werden.

4.9 Jet Brains MPS

4.9.1 Allgemeine Einführung in MPS

Besonders hervorzuheben unter den bereits existierenden Implementierungen ist das MetaProgramming System (MPS) Lösung der Firma Jet Brains. Das Hauptaugenmerk dieser Implementierung liegt darin, bestehende Sprachen zu erweitern oder eigene domänenspezifische Sprachen zu erstellen. Dies ist notwendig, da existierende Sprachen durch eine feste Definition ihrer Grammatik definiert sind und somit nicht selbständig durch Entwickler erweitert werden können. Hierdurch werden die Möglichkeiten einer Sprache terminiert. Ein Problem bei der Erweiterung von Sprachen liegt hauptsächlich in ihrer textuellen Repräsentation. Das Editieren in MPS findet direkt auf dem AST einer Sprache statt. Hierbei wird das Konzept der Projektion angewandt, bei der lediglich eine textuelle Darstellung des AST im eigentlichen Editor angezeigt wird. Projekte in MPS bestehen aus den *Solutions* und den *Lan-*

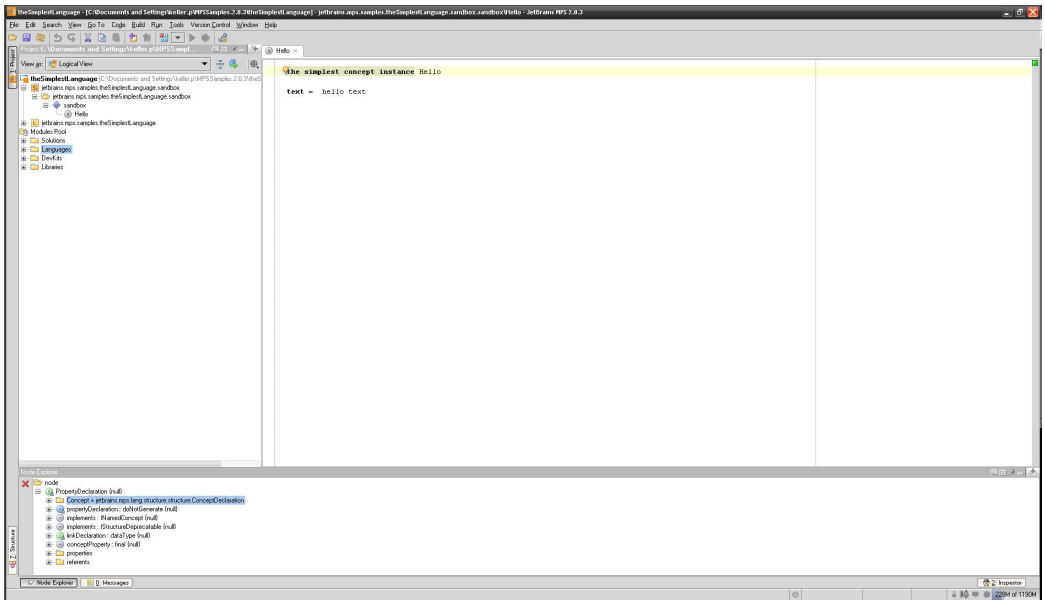


Abbildung 4.30: MPS MetaProgramming System

guages. Hierbei wird eine *Solution* auf Basis einer *Language* vorgenommen. Eine *Language* kann dabei wieder von anderen *Languages* erben, wodurch Sprachen kombinierbar und erweiterbar werden. Abbildung 4.30 zeigt den Aufbau des Editors. Im Folgenden soll ein Beispiel von der Produktseite des Editors aufgegriffen und erweitert werden. Abbildung 4.31 zeigt die Struktur eines Projekts im Editor.

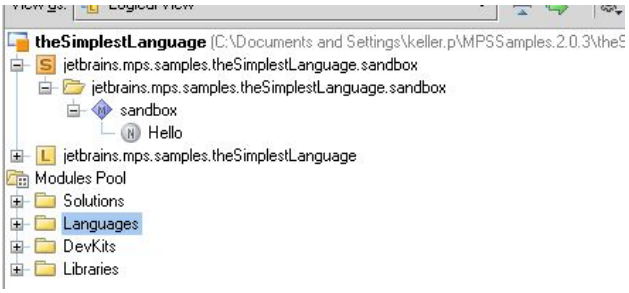


Abbildung 4.31: Project Struktur MPS

Unterhalb des Elements, welches mit dem Zeichen *L* gekennzeichnet ist findet sich die Definition der eigentlichen Sprache. In dem Beispiel auf der Produktseite von MPS [Jet] handelt es sich im Prinzip um eine *Hello World* Sprache. Diese Sprache besteht aus einer Definition eines Wurzelelements,

welches ein Textelement beinhaltet. Dies ist die eigentliche Grammatik der Sprache. Sprachen werden in MPS nicht auf herkömmliche Art definiert, sondern auf Basis eines objektorientierten Konzepts erstellt. Hierbei erben alle Sprachelemente wie in der Objektorientierung von einem gemeinsamen Basisobjekt, welches in MPS durch das Objekt *BaseConcept* dargestellt wird.

Das Objekt *BaseConcept* kann als generelle Mutterklasse verstanden werden, von der jedes Sprachelement automatisch erben muss. In Abbildung 4.31 ist auch zu sehen, dass das definierte Sprachelement *TheSimplestConcept* von dieser Klasse erbt und eine Eigenschaft von Typ *Text* besitzt. Diese Informationen zur Definition der Sprache können innerhalb MPS über die Strukturansicht für die Sprachdefinition angezeigt werden. Neben der Sicht auf die Definition der Sprache bietet MPS auch die Möglichkeit einer Definition der späteren Ansicht im Editor. Hier kann der Entwickler ein Template beschreiben, welches bei der Programmierung auf Basis der Sprache im Editor angezeigt wird, wenn das Sprachelement verwendet wird. Durch dieses Template wird also definiert wie die Ansicht der Syntax im AST-basierten Editor dargestellt wird. Für die einzelnen Elemente einer Sprache werden so Abstände, Absätze, Leerzeichen und Einrückungen einheitlich definiert, was im späteren Verlauf dazu führt dass Programme einer Sprache einheitlich aussehen. Dieser Menüpunkt wird ebenfalls unter dem Element der Sprachdefinition im Untermenü Editor angezeigt. Abbildung 4.32 zeigt die Definition des Layout der Sprache in MPS.

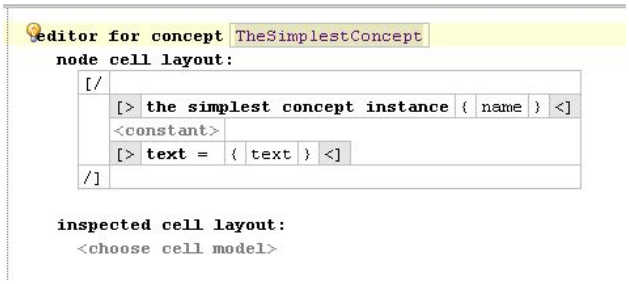


Abbildung 4.32: Definition Layout Editor

Während bei einer herkömmlichen Programmiersprache die Grammatik fest definiert ist, bietet MPS den Ansatz, dass die Sprache um eigene Konstrukte erweitert oder mit anderen Sprachen kombiniert werden kann. Abbildung 4.33 verdeutlicht nun, dass das definierte Element *TheSimplestConcept* durch ein Kindelement erweitert werden kann. Über die Funktion *CTRL Space* lassen sich die Elemente der Sprache anzeigen, die an einer bestimmten Stelle

im Editor eingebunden werden können. Hierbei wird direkt auf der in MPS erstellten Grammatik der Sprache gesucht.

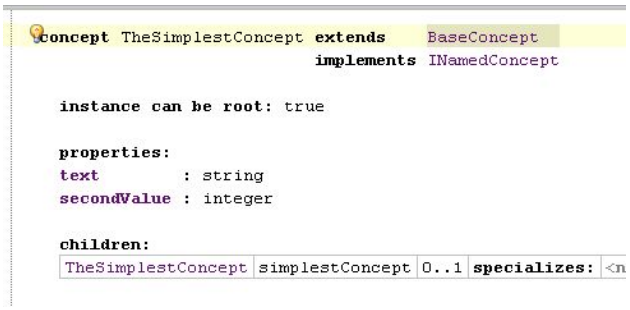


Abbildung 4.33: Erweiterung der Sprache in MPS

Nachdem ein neues Element in die Sprache eingebunden wurde, muss über den Template Ansatz auch definiert werden, an welcher Stelle in der textuellen Repräsentation des AST das entsprechende Element angezeigt werden soll. Ohne diese Anpassung hätte der Entwickler innerhalb des Editors keine Möglichkeit das neue Element hinzuzufügen. Abbildung 4.34 zeigt die Erweiterung des Templates durch das neue Kindelement.

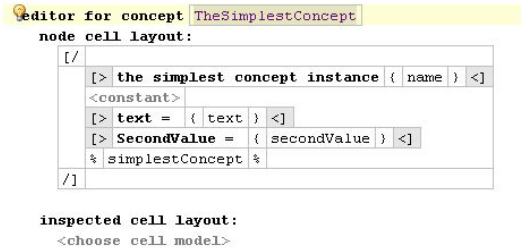


Abbildung 4.34: Erweiterung des Templates

In diesem Zusammenhang ergibt sich ein Vorteil, den MPS gegenüber der herkömmlichen Entwicklung mit Java bietet. MPS bietet die Möglichkeit eigene Sprachen zu definieren, die von anderen Sprachen erben können. Da eine Basisimplementierung von Java in MPS vorhanden ist, kann diese Sprache um eigene Konstrukte erweitert werden. Somit besteht die Möglichkeit, die Sprache selbst an die eigenen Bedürfnisse anzupassen und in MPS auf Basis des Syntaxbaums den Code zu editieren. Neu definierte Konstrukte verhalten sich wie die bestehenden Basiskonstrukte. Über den Shortcut *CTRL - Space* wird innerhalb der Kontrollfragmente ein Kontextmenü angezeigt, welches

auf Basis des aktuellen Kontext mögliche Aufrufe, Variablen oder Statements anbietet. Die Konzepte der eigenen Sprache können so z.B. von Elementen der Basissprache wie Java abgeleitet werden. Dies geschieht, in dem die definierte Sprache von einer anderen Sprache erben kann. Abbildung 4.35 zeigt auf, wie von einer bestehenden Klasse in Java eine Vererbung durchgeführt werden kann.

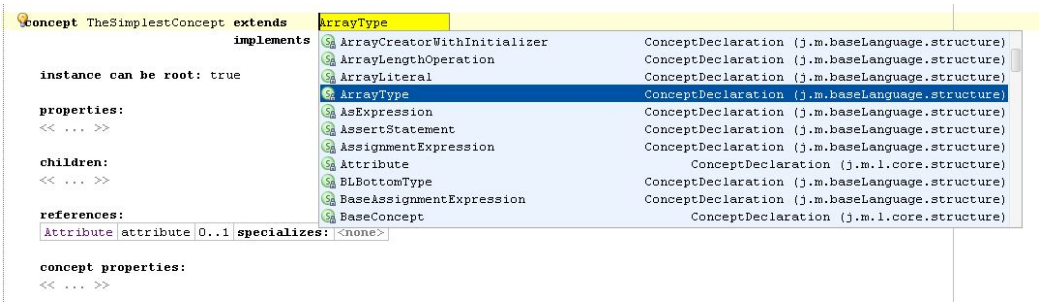


Abbildung 4.35: Java Spracherweiterung

Insgesamt ist die Programmierung in MPS sehr gut dargestellt, obwohl die Freiheitsgrade eines normalen textuellen Editors verloren gegangen sind. Alles was in MPS editiert wird, wird nicht direkt in Java Byte Code umgewandelt, sondern als XML serialisiert. Hinsichtlich der Editierung von Code auf einem Syntaxbaum stellt der Editor einen großen Fortschritt dar. Jedoch ist die Lernkurve zur Erstellung von Programmen sehr hoch. Aktuell wird die Programmierung erschwert, da das Konzept des Editors und das Erben von Grammatiken zuerst vom Anwender verinnerlicht werden muss. Der Editor ist nicht für Einsteiger in die Programmierung geeignet, sondern lediglich für erfahrene Programmierer, die die Grenzen bestehender Sprachen durch eigene DSL oder Erweiterungen erweitern wollen [Jet].

Zwar ist die Benutzerführung des Editors teilweise gewöhnungsbedürftig, jedoch wird der Entwickler durch eine Vielzahl an Menüs und Kontextmenüs bei der Erstellung des Quellcode unterstützt. Der Editor unterstützt den Entwickler auch bei der Durchführung von Refaktorisierungen, dem Verschieben und auch dem Löschen von erstellten Konzepten innerhalb des Editors. Bei Änderungen an der eigentlichen Definition der Sprache muss sowohl die Sprache als auch die Solution die auf ihr basieren neu erstellt werden. Jet Brains hat sich dazu entschlossen, MPS als Open Source Version öffentlich zur Verfügung zu stellen, so dass durch eine Community auch eigene Erweiterungen geschrieben werden können. Die Speicherung des Code wird auf

Basis eines serialisierten Syntaxbaums generiert, was wiederum dazu führt, dass man nicht mit anderen textuellen Editoren arbeiten kann, da sonst die Darstellung in MPS nicht unbedingt wiederhergestellt werden kann. Jedoch ist auch Teamarbeit über Source Code Management Systeme möglich, da *Diff and Merge* Operationen auch auf dem in XML serialisierten Syntaxbaum durchgeführt werden können. Wie bereits weiter oben angesprochen ist die in MPS realisierte Zielsprache Java. Aktuell existiert noch keine andere definierte Zielsprache. Dies müsste ein Entwickler zuerst vornehmen, bevor es möglich wäre Erweiterungen für diese Sprachen zu schreiben.

Die allgemeine Einführung in diese Entwicklungsumgebung dient der Verinnerlichung des implementierten Konzepts. Auf dieser Basis können nun die vorhandenen Möglichkeiten zur Durchführung von Refaktorisierungen untersucht werden. Zu diesem Zweck werden die definierten Refaktorisierungen auf Basis einer Klasse in Java, die in MPS erstellt wurde, implementiert. Abbildung 4.36 zeigt eine einfache Klasse, auf der die einzelnen Refaktorisierungen durchgeführt werden sollen.

```
root template
input <unspecified>
public class MyDemoClass extends Object implements Cloneable {
  <<static fields>>

  <<static initializer>>
  private boolean bISOK;
  private string helloWorld;
  <<properties>>
  <<initializer>>
  public MyDemoClass() {
    helloWorld = "Hello World";

    final boolean bIsNotOk = false;

    System.out.print(helloWorld);
    doPrintOut(helloWorld);
  }

  public void doPrintOut(string text) {
    System.out.print(text);
  }

  public int doReadIn() throws IOException {
    int i  = System.in.read();
    System.out.print(i);
    return i;
  }

  <<static methods>>

  <<nested classifiers>>
}
```

Abbildung 4.36: Refaktorisierungen in MPS auf Basis von Java

4.9.2 Refaktorisierungen in MPS

MPS unterstützt das Durchführen von Namensänderungen auf Klassen, Variablen und Methoden. Das Umbenennen wird direkt auf dem Syntaxbaum ausgeführt und die Referenzen werden daraufhin geändert. Auf Basis einer Methode kann auch die Signatur geändert werden. Hierfür wird ein Dialog geöffnet, der die Signatur der Methode anzeigt und diese durch den Entwickler editierbar macht. Nach Abschluss der Änderungen wird der Entwickler durch eine Konsole darauf aufmerksam gemacht, welche Anpassungen am AST des Programms durchgeführt werden. Durch eine Bestätigung dieser Aktion wird die Änderung auf alle Referenzen angewandt. Abbildung 4.37 zeigt die hervorgerufenen Änderungen durch das Ändern der Signatur einer Methode.

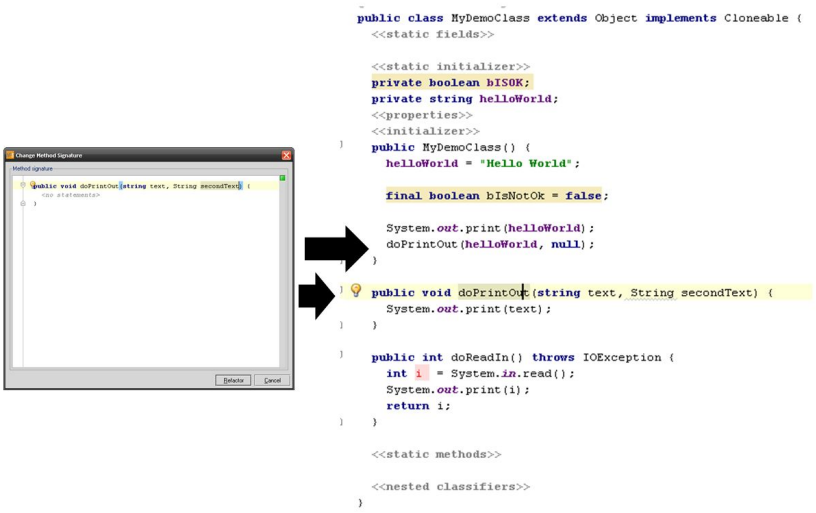


Abbildung 4.37: Refaktorisierung der Signatur einer Methode

Die Refaktorisierung *Temp to Field*, wird nicht direkt durch die Entwicklungsumgebung unterstützt. Diese Refaktorisierung lässt sich jedoch auch manuell durchführen. Hierbei wird zuerst ein neues Feld mit der gleichen Benennung eingeführt anschließend wird die Definition der lokalen Variable entfernt. Dies führt ähnlich wie in LavaPE zu einem Fehler bei der Auflösung der Referenzen, welche dem Entwickler direkt im Editor angezeigt werden. Dieser Umstand lässt sich jedoch durch eine Revalidierung der lokalen Va-

riable auflösen, da durch diesen Vorgang eine neue Referenz auf das nun vorhandene Feld gelegt wird. Abbildung 4.38 zeigt die entsprechende Refaktorisierung.

Abbildung 4.38: Refaktorisierung Temp to Field

Wenn nun das Feld gelöscht wird, so werden die Referenzen angezeigt, die im aktuellen AST noch auf diese Definition bestehen. MPS führt eine Analyse durch und zeigt in einer Ansicht am unteren Rand an, dass Konflikte bei dieser Änderung auftreten können. Der Entwickler hat jedoch die Möglichkeit diese Refaktorisierung durchzuführen, was zu einem inkonsistenten AST führt. Abbildung 4.39 zeigt die durch die Entwicklungsumgebung gefundenen Konflikte.

Somit sind die Refaktorisierungen innerhalb einer Klasse ähnlich den Automatismen in einem textuellen Editor durchführbar. Der Entwickler wird vor der Durchführung gewarnt und bekommt Hinweise zur Änderung. Im Folgenden wird das bestehende Beispiel nun erweitert, um zu prüfen, inwiefern auch Refaktorisierungen in abhängigen Klassen durchgeführt werden können. Hierbei wird untersucht, welche Refaktorisierungen MPS bei dem Verschieben von Methoden und Feldern unterstützt und wie sich diese auf die Konsistenz des AST auswirken. MPS bietet keine direkte Unterstützung um eine Methode aus einer abstrakten Superklasse in eine Subklasse zu verschieben.

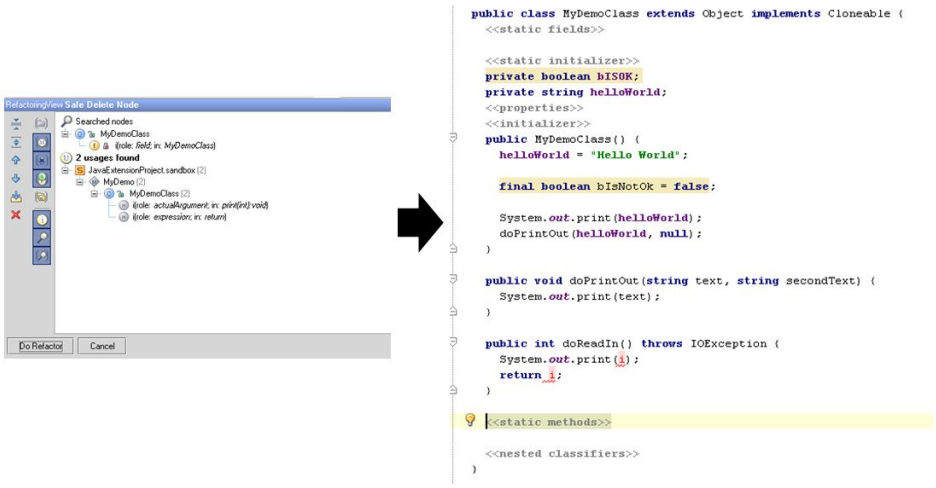


Abbildung 4.39: Eliminieren eines definierten Feldes

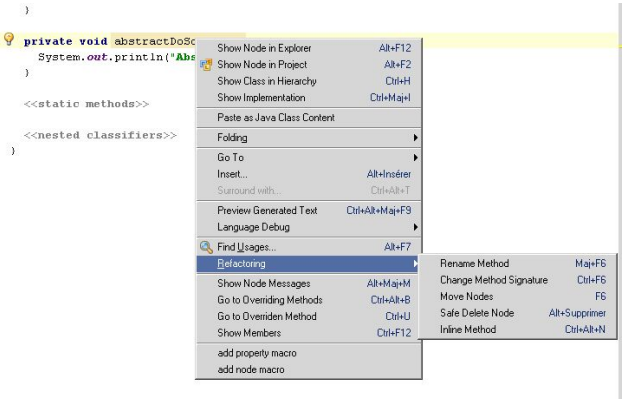


Abbildung 4.40: Refaktorisierung Move Method

Die Refaktorisierung lässt sich auf dem Baum durch eine *Cut & Paste* Operation realisieren. Dies führt wiederum zu einem kurzfristigen Fehler beim Auflösen der Referenzen aus der Subklasse, da die Methode der Superklasse nicht mehr vorhanden ist, jedoch lässt sich dieser Zustand durch ein Update auf den angezeigten Referenzen wieder auflösen. Abbildung 4.40 und 4.41 zeigen die Durchführung dieser Refaktorisierungen.

Abbildung 4.41: Nach der Refaktorisierung Move Method

Insgesamt ist MPS sehr gut geeignet um die definierten Refaktorisierungen auf dem AST des Programms durchzuführen. MPS bietet zwar weniger automatisierte Refaktorisierungen wie in *Eclipse* und *Visual Studio*, jedoch lassen sich diese auch direkt nachstellen und werden durch die Möglichkeit von *Cut & Paste* Operationen auf einer Baumansicht des aktuellen Codes unterstützt. Durch ein Update der Referenzen auf ihre Definitionen lassen sich auch kurzzeitige Inkonsistenzen auflösen.

4.10 Weitere Implementierungen

Die in diesem Abschnitt vorgestellten Implementierungen konnten nicht eindeutig als AST-basierte Editoren identifiziert werden, jedoch können Eigenschaften der AST-basierten Programmierung erkannt werden. Somit werden sie am Rande in dieser Arbeit vorgestellt.

Editor von Kirill Osenkow

Kirill Osenkow hat in seiner Diplomarbeit im Jahr 2007 das Thema der strukturierten Editoren behandelt. In diesem Kontext ist eine prototypische Implementierung entstanden, bei der auf Basis einer Untermenge von C ein strukturierter Editor entstanden ist. Ziel seiner Arbeit war es einen Syntaxbaum Editor direkt in einer Entwicklungsumgebung zu realisieren. Die Programme innerhalb des Editors werden durch hierarchische Blöcke dargestellt, welche wiederum andere Blöcke enthalten können [Ose07, Seite 36]. So können zum Beispiel Schleifen und Bedingungen wiederum eine Vielzahl anderer Blöcke enthalten. Jedoch kann eine einfache Anweisung keine eigenen Blöcke beinhalten. Der Entwickler kann auf Basis leerer Blockstrukturen diesen einen Verwendungszweck zuführen. Abbildung 4.42 veranschaulicht, welche Elemente dem Entwickler auf Basis eines leeren Blocks vorgeschlagen werden.

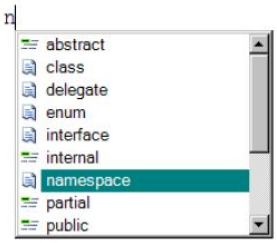


Abbildung 4.42: Leerer Block im Editor (entnommen aus [Ose07, Seite 38])

In Abbildung 4.42 wird nun eine *Namespace Declaration* vorgenommen. Auf Basis einer Code Vervollständigung werden die im aktuellen Kontext möglichen Elemente eingeblendet, die der Programmierer verwenden kann. Nach dem Erstellen der *Namespace Declaration* werden unterhalb leere Blöcke erzeugt, die die Kindelemente dieser Deklaration aufnehmen können.

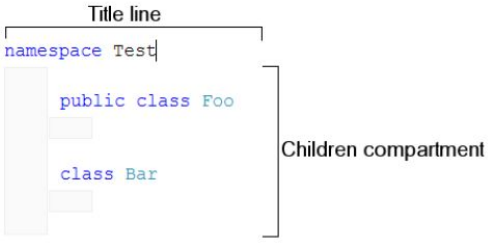


Abbildung 4.43: Kindelemente (entnommen aus [Ose07, Seite 42])

Der Editor basiert jedoch nicht vollständig auf einem AST. Nach Osenkov [Ose07, Seite 37] sind lediglich die höheren Typen im Rahmen des Prototyps umgesetzt. Methoden, Parameter und Statements sind auf Basis von Text realisiert worden. Dieser Text wird dann mit den strukturierten Elementen des Editors verknüpft. Es handelt sich also hierbei lediglich um einen partiellen AST-basierten Editor, da die Methoden der Objekte in textueller Form erstellt und somit bei der Programmausführung weiterhin in einen Syntaxbaum geparst werden müssen. Da dieser experimentelle Editor nicht zu Verfügung steht, sind weitere Untersuchungen im Kontext von Refaktorisierungen nicht möglich. Es ist jedoch anzunehmen, dass der Editor die im Rahmen dieser Arbeit betrachteten Refaktorisierungen nicht unterstützt, da es sich um einen hybriden Editor handelt, der nur teilweise die Konzepte der AST-basierten Programmierung umsetzt. Abbildung 4.43 zeigt auf Basis des Namespace die Klassen als Kindelemente an.

App Inventor for Android

Der App Inventor für Android ist eine neue webbasierte Entwicklungsplattform von Google zur schnellen Anwendungserstellung auf Basis eines Syntaxbaums. Abbildung 4.44 zeigt eine schematische Darstellung des Prozesses der Anwendungsentwicklung.

Ziel der Plattform ist die schnelle Applikationsentwicklung für Enduser. Der Editor bietet sowohl einen Designmode für User Interfaces als auch eine Komponente, bei der durch einen Syntaxbaum Editor die eigentliche Ablauflogik für die User Interfaces erstellt werden kann. Der Syntaxbaum Editor besitzt dabei ein Menü durch welches die Kontrollelemente direkt in den Editor per *Drag & Drop* gezogen werden können. Die Konstrukte können sich dabei auf die Ereignisse der *UI* Oberflächenobjekte registrieren, wodurch ereigni-

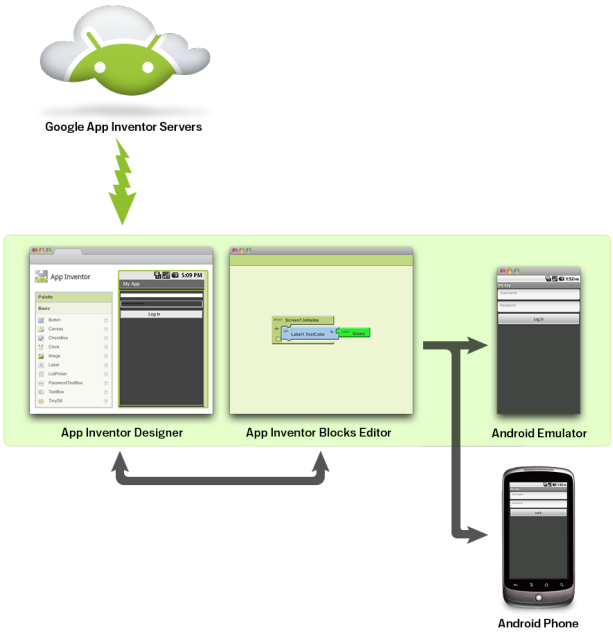


Abbildung 4.44: AppInventor-Doc-Diagram (entnommen aus [Goo09])

sorientierte Programme erstellt werden können. Abbildung 4.45 zeigt einen Ausschnitt der block-orientierten Anwendungsentwicklung, die Ähnlichkeiten mit Scratch und Elements for Smalltalk aufweist.

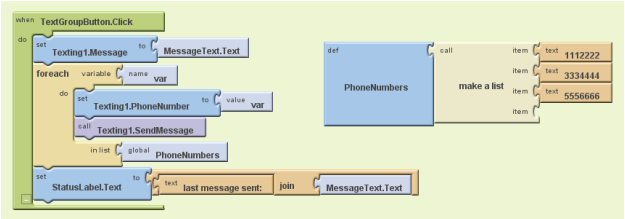


Abbildung 4.45: Syntax-Baum Editor (entnommen aus [Goo09])

Das Projekt von Google befindet sich aktuell noch in der Beta Phase. Die Projektseite ist ausführlich dokumentiert und zielt durch die Einfachheit und Klarheit der Beschreibung auf technisch versierte Enduser ab, die kleinere Applikationen gerne selbst erstellen wollen. Der Programmierer ist allerdings durch die Funktionen der webbasierten Entwicklungsplattform begrenzt. Es ist nicht möglich die erstellten Applikationen in Java Code zu konvertieren, so dass mögliche Erweiterungen auf Basis der kompletten API von Android

entwickelt werden können. Somit ist die Plattform eher zur Erstellung von Features für Enduser als zur professionellen Anwendungsentwicklung gedacht [Goo09].

Am 31.12.2011 hat Google den Support für den AppInventor aufgegeben. Mitte 2012 wird erwartet, dass das MIT Center for mobile Learning einen neuen Service auf Basis des App Inventors bereitstellen wird. Aus diesem Grund ist an dieser Stelle das Verhalten des App Inventors bei Refaktorisierungen nicht weiter zu untersuchen.

Microsoft SharePoint Designer

Der SharePoint Designer in der Version 2010 kann nicht direkt als ein AST-basierter Editor bezeichnet werden, jedoch verwendet er den sogenannten Workflow Designer, der es auf einfache Weise erlaubt, auf dem Objektmodell der Windows Workflow Foundation aufbauend wiederverwendbare Workflows zu erzeugen, die auf den SharePoint Server publiziert werden können. Hierbei werden die im Editor erzeugten graphischen Elemente beim Speichern in XOML transformiert. XOML ist ein auf XML basierendes Entwicklungsframework, welches auf Basis von .NET realisiert ist. Die generierten Artefakte lassen sich auch nach Visual Studio importieren, wo Erweiterungen vorgenommen werden können, die die Grenzen des graphischen Workflow Designers erreichen. Der Designer ist also eine vollständige Komponente zur schnellen und einfachen Erstellung von wiederverwendbaren Applikationen. In diesem Zusammenhang hat Microsoft ein Tool entwickelt welches nicht primär auf die Entwickler, sondern eher auf Business Analysten und Power User ausgerichtet ist. Das Tool bietet dem Enduser die Möglichkeit die Ablaufstrukturen eines Workflows mit Verzweigungen, Vorgangszuordnungen und If-Anweisungen auf eine graphische Weise zu erstellen. Der Editor bietet dem Anwender also einen vereinfachten Zugriff auf das Objektmodell und damit die Möglichkeit, ohne Code sequenzielle Prozesse zu erstellen. Der Code wird auf Basis des bereits beschriebenen XML Dialekts gespeichert und kann direkt über eine Assembly, eine Bibliothek in Microsoft, auf den verbundenen Server publiziert werden. Abbildung 4.46 kann man erkennen, auf welche graphische Art und Weise die Ablaufstrukturen eingegeben werden können [Mic10]). Abbildung 4.46 zeigt eine Workflow Aktivität, auf der verschiedene Aktionen ausgeführt werden können die entweder sequentiell oder parallel ausgeführt werden können.

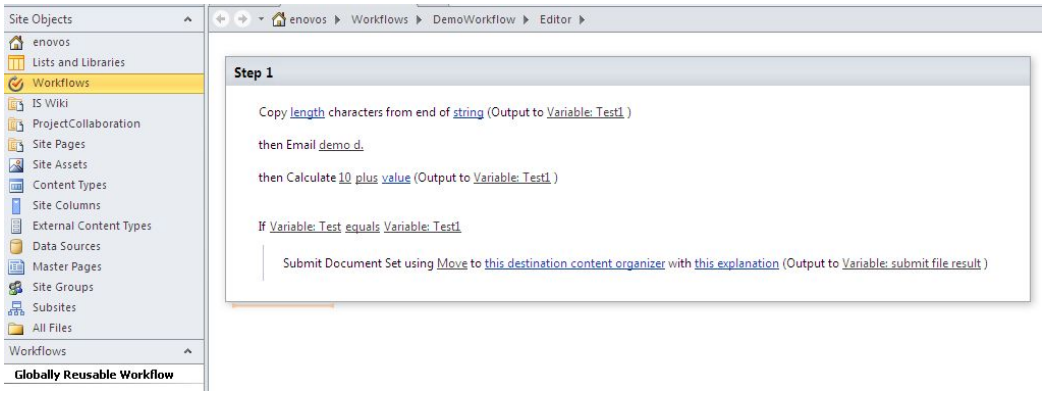


Abbildung 4.46: SharePoint Designer

Die Elemente, die der Benutzer zur Erstellung der Workflows benutzen kann, werden über die Symbolleiste zur Verfügung gestellt. Der Benutzer kann durch die Menüführung Bedingungen, vordefinierte Aktionen und weitere Prozessschritte einführen. Innerhalb eines Prozessschritts können Aktionen sequentiell oder parallel ausgeführt werden. Anzumerken ist, dass es sich hierbei nicht um ein konsistentes Modell handelt, da semantische Fehler bei der Bearbeitung nicht erkannt werden. Des Weiteren findet bei der Erstellung von Berechnungen auch keine syntaktische Prüfung statt, ob es sich bei den Operanden um Zahlen oder Texte handelt. Das transiente Modell kann jedoch zu jedem Zeitpunkt während der Bearbeitung gespeichert werden. Da durch die entsprechenden syntaktischen und semantischen Freiheitsgrade keine direkte Prüfung auf dem Modell vorgenommen wird, existiert eine Funktion zum Validieren des aktuellen Modells. Diese Validierung zeigt in der graphischen Repräsentation des Workflows die möglichen Fehlerquellen an. Über die Funktion *Publish* kann der erstellte Workflow auf den verknüpften SharePoint publiziert und somit den Anwendern zur Verfügung gestellt werden. Die nicht existente syntaktische und semantische Prüfung steht somit für das Fehlen eines konsistenten Modells. Einfache Refaktorisierungen wie das Umbenennen einer Variable sind möglich, jedoch wirkt sich eine Änderung nicht auf die vorhandenen Referenzen innerhalb des Modells aus. Auf Basis dieser Erkenntnis wird von einer weiteren Analyse des SharePoint Designers im Rahmen dieser Arbeit abgesehen, da es sich nicht um einen AST-basierten Editor handelt.

Rathereasy

Rathereasy ist ein experimenteller Syntaxbaum-Editor, der für die Programmiersprache Eastwest entwickelt wurde. Im Rahmen dieses Forschungsprojekts soll geprüft werden, wie die funktionale Programmierung durch die Verwendung von strukturierten Editoren beschleunigt werden kann. Ein Vorteil dieses Editors ist, dass auf Basis eines Ausdrucks ein Kontextmenü aufgeklappt wird, welches daraufhin die entsprechenden sinnvollen Vorschläge liefert. Diese Vorschläge können nicht nur durch eine Maussteuerung, sondern auch durch die Funktionstasten ausgewählt werden. Somit kann der Editor vollständig über die Tastatur gesteuert werden, was die Geschwindigkeit der Programmerstellung erhöht. Negativ anzumerken ist, dass der Editor lediglich in einem experimentellen Zustand ist und das User Interface trotz schneller Bedienbarkeit nicht sehr übersichtlich gestaltet ist [LeN]. Der Editor liegt als Quellcode vor und lässt sich downloaden. Jedoch traten Fehler im Rahmen des Installationsprogramms auf, die ein weitergehendes Testen im Rahmen dieser Arbeit nicht ermöglichten. Aus diesem Grund wird an dieser Stelle nicht weiter auf den Editor eingegangen.

Subtext

Bei Subtext handelt es sich um einen strukturierten Editor, der sich vor allem um die Darstellung von Bedingungen kümmert. Hierbei nutzt Subtext den Ansatz von Entscheidungstabellen, um die verschiedenen Bedingungen in verschachtelten Konstrukten lesbar zu machen. Der Editor ist in einem experimentellen Status und unterstützt lediglich einfache Variablen und verschachtelte Bedingungen. Es gibt keine experimentelle Version von Subtext die zum Download zur Verfügung steht. Aus diesem Grund können auch mögliche Refaktorisierungen von Referenzen in diesem Kontext nicht geprüft werden. Auch die verfügbaren Quellen zu diesem Thema lassen keine eindeutige Schlussfolgerung zu, ob der Editor die hier zu analysierenden Verhaltensweisen unterstützt [Edw07]. Abbildung 4.47 zeigt einen Ausschnitt der Entwicklungsumgebung mit einer Entscheidungstabelle aus der Arbeit von Edwards [Edw07].

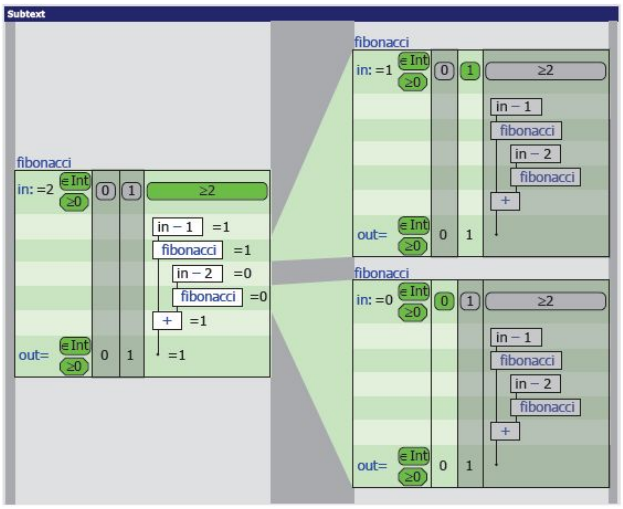


Abbildung 4.47: Bedingungstabellen in Subtext ([Edw07, Seite 14])

Intentional

Bei der Intentional Workbench handelt es sich um proprietäres Produkt zur Erstellung von domänenspezifischen Sprachen. Ähnlich wie MPS der Firma Jet Brains wird bei der Erstellung einer domänenspezifischen Sprache mit einem Editor gearbeitet, der auf Basis des Syntaxbaums der zu erstellenden Sprache arbeitet. Auf Basis des AST in der definierten Sprache, wird dann Code in der eigentlichen Zielsprache erstellt. Intentional kann dabei Code in Java, C und Ruby erstellen. Somit findet bei der Erstellung des Codes eine *Model to Model* Transformation statt, die das Modell der definierten Sprache in den AST einer Zielsprache überführt. Abbildung 4.49 stammt aus der Präsentation von Christerson [Chr10] und zeigt auf, dass der Entwickler direkt auf dem AST seiner domänenspezifischen Sprache arbeitet, die wiederum anschließend als Ausgangspunkt für die Generation des Quellcode benutzt wird. Das Schema der Entwicklung mit Intentional ist in Abbildung 4.49 dargestellt, welche ebenfalls aus der Präsentation von Christerson ([Chr10]) stammt.

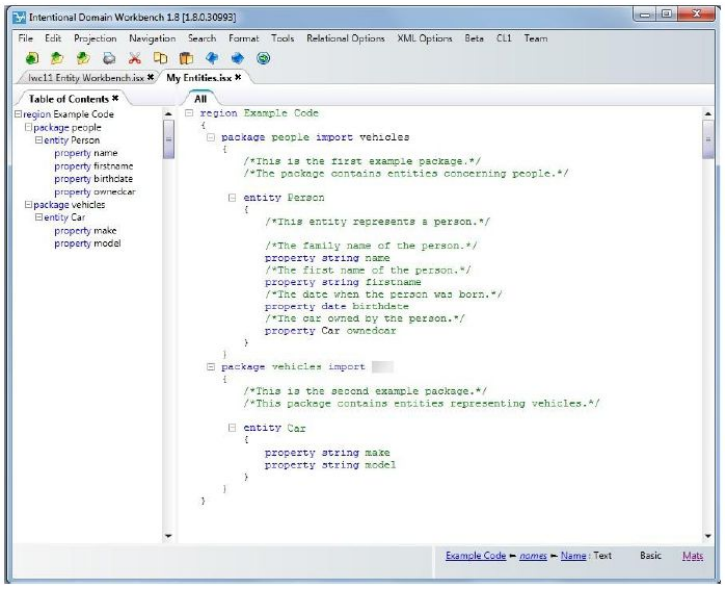


Abbildung 4.48: Intentional Language Workbench (entnommen aus[Chr10])

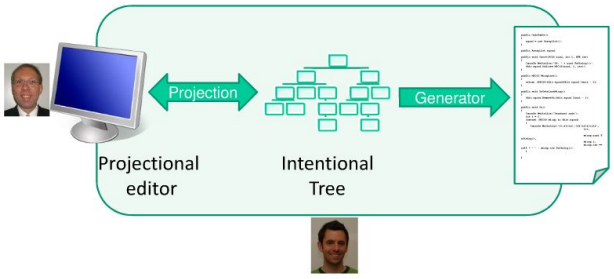


Abbildung 4.49: Syntaxbaum in der Intentional Workbench (entnommen aus [Chr10])

Intentional bietet also die Möglichkeit, das Schema einer Sprache zu erstellen, auf welcher dann der Domänenexperte entsprechende Programme realisieren kann. Diese Programme werden dann auf dem Syntaxbaum der definierten Sprache erstellt. Abbildung 4.50 aus der Präsentation von [Chr10] zeigt die unterschiedlichen Projektionen, die auf Basis der Grammatik der Sprache realisiert werden können.

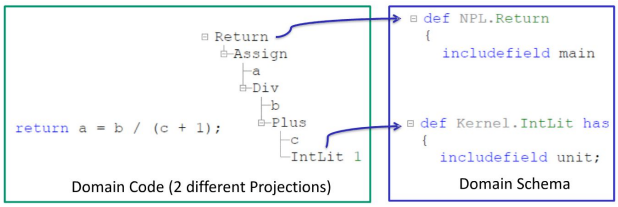


Abbildung 4.50: Darstellung der Projektionen in Intentional (entnommen aus [Chr10])

Eine weitergehende Analyse dieses Produktes kann an dieser Stelle nicht vorgenommen werden, da keine freie Version der vorgestellten Software zum Download zur Verfügung steht.

PureBuilder

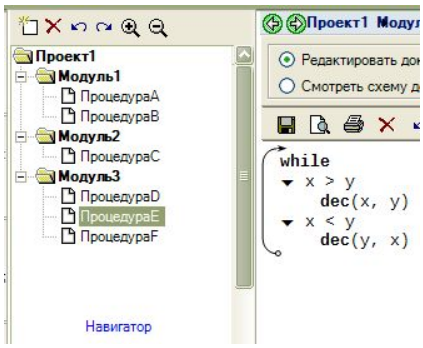


Abbildung 4.51: Pure Builder (entnommen aus [PL09])

Der exemplarische Editor PureBuilder stellt nach Sergey Prokhorenko und Valery Laptev [PL09] eine vollständig AST-basierte Implementierung dar, der es möglich sein soll Code in verschiedenen Sprachen zu erzeugen. Der Quellcode wird also bei der Erstellung von Software vollständig eliminiert und die Fehleranfälligkeit soll reduziert werden. Der Editor steht nicht zum

Download zur Verfügung und seine Projektpage ist in russisch gehalten. Aus diesem Grund ist keine weitergehende Analyse dieses Editors möglich. In Abbildung 4.51 wird die block-orientierte Erstellung von Code dargestellt.

4.11 Kategorisierung der Implementierungen

Der folgende Abschnitt dient dazu, die gewonnen Erkenntnisse aus den Untersuchungen der Entwicklungsumgebungen auf Basis von Kategorien zusammenzufassen und zu bewerten.

Bedienbarkeit des Editors

LavaPE stellt einen AST-basierten Editor dar, der auf der Programmiersprache Lava erarbeitet wurde. Er enthält einen strukturierten Editor, der eine textuelle Eingabemöglichkeit bietet. Diese Form der Erstellung des Quellcode kann mit der Editierung in MPS verglichen werden. In diesem Zusammenhang bietet MPS dahingehend Vorteile, dass das Objektmodell auch über die kontextsensitive Codevervollständigung verfügt. Dies bedeutet, dass MPS über *CTRL Space* alle möglichen Strukturelemente auf einem bestehenden Kontext anbietet. Dies funktioniert nicht in LavaPE, da dort die Elemente über eine Palette ausgewählt werden müssen, was den Entwicklungsprozess verlangsamt und den Entwickler zwingt, Code durch das Selektieren von Ausdrücken zu erstellen. Ähnliche Eigenschaften wie MPS bietet auch das Framework Barista. Ein Vorteil gegenüber MPS ist hier jedoch die Einblendung von Beispielen und Graphiken in das Kontextmenü, so dass der Entwickler direkt ein Beispiel zum gewählten Statement sieht. Diese Fähigkeit des Editors ließ sich über die Seite des Forschungsprojekts identifizieren. Inwiefern Barista wirklich diese Möglichkeiten bietet, kann im Rahmen dieser Arbeit nicht überprüft werden. Der Editor VFPE bietet eine graphische Editierung, die die Struktur des AST widerspiegelt. Eine Ansicht der eigentlichen Struktur des zugrundeliegenden Codes wird über den Tooltip auf einem selektierten Element angezeigt. Scratch bietet graphische Elemente die an *Lego* Bausteine erinnern. Es existiert keine Codevervollständigung und die Elemente können nur via *Drag & Drop* auf die Bearbeitungsoberfläche gezogen werden. Elements for Smalltalks basiert auf Scratch und verwendet die Erstellung von Code durch die Kombination der Bausteine miteinander. Somit stimmt das Bedienkonzept mit dem von Scratch überein. Ein weiterer Vertreter der blockorientierten Erstellung von Code ist Star Logo. Ähnlich wie

in Alice und Scratch werden hier graphische Blöcke durch ihre geometrischen Formen zu Code zusammengestellt. LavaPE, VFPE und MPS kennzeichnen sich dadurch aus, dass sie ungültige Eingaben direkt verhindern. Bei LavaPE fällt auf, dass Variablen gelöscht werden können, jedoch aber noch Referenzen auf diese Variablen existieren. MPS führt Prüfungen durch und zeigt durch Löschungen verursachte Fehler an. Scratch zeigt prinzipiell keine Fehler an, wenn eine Variable gelöscht wird. Dies gilt auch für Elements for Smalltalk. Barista und VFPE konnten in diesem Zusammenhang nicht analysiert werden. AST-basierte Editoren sollten sich dahingehend auszeichnen, dass Referenzen wie zum Beispiel Variablen oder Prozeduren durch eine Umbenennung automatisch angepasst werden. LavaPE ändert direkt alle Referenzen auf eine Variable oder Prozedur, wenn die Definition angepasst wird. Dies gilt auch für MPS. Scratch bietet keine Möglichkeit den Namen einer Variable nach ihrer Erstellung wieder zu ändern. In diesem Fall muss die Variable gelöscht und eine neue erstellt werden. Die Bedienung eines AST-basierten Editors sollte durch die Vermeidung von Syntaxfehlern zu einer leichteren Bedienbarkeit führen und somit die Produktivität des Entwicklers steigern. LavaPE ist in seiner Bedienungsführung gewöhnungsbedürftig, da der Entwickler die möglichen Konstrukte immer von einer Palette in den eigentlichen Editor ziehen muss. Dies ist gerade bei umfangreichen Entwicklungen weniger vorteilhaft, da der Entwickler immer zwischen der Maus und dem Keyboard wechseln muss. Code Vervollständigung oder die Anzeige eines Kontextmenüs existieren in dieser Implementierung nicht. VFPE basiert auf dem gleichen Prinzip und besteht wie bereits beschrieben aus einer rein graphischen Oberfläche. Zudem bietet VFPE im Kontextmenü eine Codeansicht an, über die sich der Entwickler informieren kann. Scratch stellt durch sein kindgerechtes User Interface einen sehr schnellen Einstieg in die Entwicklung dar, jedoch sind auch in diesem Editor Codevervollständigung und Kontextmenüs nicht vorhanden. Elements bietet hier einen kleinen Vorteil, da hier Kontextmenüs vorgesehen sind, die das Löschen und Umbenennen als einfache Refaktorisierungen anbieten. MPS bietet seitens der Usability den gelungensten und vollständigsten Ansatz AST-basierter Editoren, da sich Code unmittelbar über die Tastatur erzeugen lässt und der Entwickler kaum *Drag & Drop* oder Klick Operationen durchführen muss um seine Arbeit am Code durchzuführen.

Die MPS Entwicklungsumgebung ist besonders für sehr erfahrene Entwickler gedacht, da man mit ihr die Grenzen von bestehenden Sprachen erweitern oder eigene domänenspezifische Sprachen erstellen kann. Die Implementierung von Rathereasy verfolgt einen ähnlichen Ansatz wie MPS. Die Editierung findet auf dem Syntaxbaum statt, jedoch kann über Codever-

vollständigung oder Vorschläge auf Basis eines Kontextmenüs die komplette Entwicklung eines Programms über die Tastatur vorgenommen werden. Es handelt sich hier jedoch um einen experimentellen Editor für die Sprache OCaml, bei der es sich um eine Variante der Sprache Caml handelt. Die vorhandenen Erkenntnisse basieren auf den existierenden Hinweisen der entsprechenden Projektseite.

Darstellung von semantischen Referenzen

Die Sichtbarkeit von Referenzen, die bei der textuellen Erstellung von Quellcode erst durch die semantische Analyse erstellt werden, ist ein weiterer Vorteil der AST-basierten Programmierung. MPS und LavaPE bieten Funktionen um auf Basis einer Referenz direkt zu der eigentlichen Definition zu gelangen. Dies stellt einen enormen Vorteil im Rahmen der Übersichtlichkeit bei der Editierung dar. Es ist jedoch auch anzumerken, dass moderne Entwicklungsumgebungen, die auf Basis eines textuellen Editors arbeiten, ähnliche Sprünge innerhalb des Codes anbieten. Die getesteten Editoren bieten darüber hinaus jedoch keine besondere Ansicht zur graphischen Visualisierung dieser Referenzen durch gesonderte Diagramme oder spezielle graphische Komponenten.

Unterstützung von Refaktorisierungen

Unter den hier aufgeführten Implementierungen bietet vor allem MPS Unterstützung von Refaktorisierungen an. Es wird nicht die Masse an Refaktorisierungen unterstützt wie zum Beispiel in Eclipse, jedoch lassen sich die im Rahmen dieser Arbeit analysierten Refaktorisierungen innerhalb des Editors durchführen. Diese Fähigkeit wird besonders durch die *Cut & Paste* Operationen unterstützt.

Aktuelle Marktverbreitung und Chancen

Die hier aufgeführten Entwicklungsumgebungen haben mehr oder weniger eine gewisse Marktreife erreicht. Barista, VFPE und Rathereasy sind experimentelle Editoren, die sich bislang aus ihrem Stand der Forschung heraus nicht weiterentwickeln konnten. LavaPE hat diesen Zustand leicht erweitert und steht zur Verfügung, jedoch machen Recherchen zu diesem Thema auch deutlich, dass der Editor keine besonders große Verwendung gefunden hat. Dies liegt auch vor allem daran, dass LavaPE auf einer eige-

nen Programmiersprache basiert, die in sich jedoch nicht erweitert werden kann. Des Weiteren ist seine Bedienbarkeit auch nach längerem Gebrauch sehr gewöhnungsbedürftig. Scratch zeichnet sich durch eine große Community und durch seine verbreitete Verfügbarkeit in über 50 Sprachen aus. Der Vorteil von Scratch besteht darin, dass die Entwicklungsumgebung nicht versucht AST-basierte Programmierung im Allgemeinen durchzusetzen, sondern lediglich auf das spielerische Erlernen der Programmierung fokussiert ist. Diese Tatsache ist de facto ein Alleinstellungsmerkmal, welches dazu beigetragen hat, dass diese Entwicklungsumgebung eine weite Verbreitung gefunden hat. In ihr liegt auch ein interessantes Prinzip, dass durch das leichtere Erstellen von Code mehr Anwender für die Entwicklung von Software begeistert werden können. Dieses Prinzip der schnellen Anwendungsentwicklung wurde auch in den Produkten Microsoft SharePoint Designer und im App Inventor for Android eingesetzt. Diese Editoren sind gekennzeichnet durch eine besondere Einfachheit der Programmierung, so dass auch kaum versierte Benutzer ohne großen Aufwand schnell Ergebnisse erzielen können. Diese Editoren bieten jedoch in ihrer Einfachheit auch Limitierungen, da nicht alle Möglichkeiten der Sprache im entsprechenden Editor realisiert sind. Den umfassendsten Ansatz bietet MPS. Der Vorteil liegt klar in der Vollständigkeit der Implementierung, da dieser Editor nur wenige Einschränkungen gegenüber bestehenden Entwicklungsumgebungen wie Eclipse oder Visual Studio aufweisen. In Abbildung 4.52 werden die Eigenschaften tabellarisch dargestellt.

Entwicklungsumgebung	Bedienbarkeit	Marktverbreitung	Zielgruppe	Implementierte Refaktorisierungen
LavaPE	aufwendig	gering	Entwickler	Rename
Barista	nicht feststellbar	keine	Entwickler	nicht feststellbar
Alice	einfach	mäßig	Schüler	Rename
VFPE	nicht feststellbar	keine	Entwickler	nicht feststellbar
Scratch	Einfach	mäßig	Schüler	Rename
Elements f. Smalltalk	aufwendig	keine	Entwickler	Rename
Star Logo und Open Blocks	einfach	mäßig	Schüler	Rename
Jet Brains MPS	aufwendig	mäßig	Entwickler	Rename, Delete, Move, Inline Variable, Introduce Field, Constant, Variable

Abbildung 4.52: Eigenschaften der AST-basierten Editoren

4.12 Zusammenfassung

Vorteile der AST-basierten Programmierung liegen in der schnelleren Erlernbarkeit einer neuen Sprache. Durch die Verwendung von vordefinierten Schablonen im Editor können Elemente der zugrundeliegenden Sprache schneller zusammengefügt werden, ohne dass syntaktische Fehler den Arbeitsprozess

behindern. Diese Form der Entwicklung kann somit einen gewissen Geschwindigkeitsvorteil bieten, wenn es darum geht Lösungen schneller zu erstellen. Jedoch ist auch anzumerken, dass Entwicklungsumgebungen wie *Eclipse* oder *Visual Studio*, die aus textuellen Editoren bestehen, ebenfalls Ansätze bieten um den Entwicklungsprozess durch Code Vervollständigung oder durch Syntax und Semantik Prüfungen sowie Lösungsvorschläge zu optimieren. Dies wird vor allem deutlich, wenn man die Entwicklung von Software in Java mit den Plattformen Eclipse und MPS der Firma Jet Brains vergleicht. Beide Implementierungen bieten nach einer gewissen Einarbeitung eine Vielzahl von Funktionalitäten zur Unterstützung des Software Entwicklungsprozesses. Andere AST-basierte Implementierungen, die im Rahmen dieser Arbeit untersucht wurden, sind oft für andere Anwendungsbereiche entwickelt worden und lassen sich eher schlecht in diesem Kontext vergleichen. Festzustellen ist jedoch, dass die AST-basierten Editoren potentiell mehr Möglichkeiten bieten als bislang in den bisherigen Realisierungen implementiert wurden. Diese Aussage trifft vor allem auf den Bereich der Refaktorisierungen zu. Das konsistente Umbenennen und Löschen auf den Syntaxbaum wird jedoch von einer Vielzahl der Realisierungen unterstützt. Diese Änderungen lassen sich leicht auf dem Syntaxbaum durchführen. Komplexe Refaktorisierungen wie zum Beispiel *Move Up / Down Method* sind aufwendiger zu realisieren und bislang in den untersuchten Implementierungen nicht umgesetzt. Um komplexe Refaktorisierungen zu implementieren ist es notwendig, dass diese in Einzelschritte zerlegbar sind, die die Konsistenz des Syntaxbaums nicht beeinflussen. Dies ist vor allem bei komplexen manuellen Refaktorisierungen schwierig zu realisieren, da durch die AST-basierte Implementierung Freiheitsgrade gegenüber der textuellen Programmierung verschwinden. Alle AST-basierten Editoren stehen noch in einem sehr frühen Entwicklungsstadium. Vor allem bei professionellen Entwicklern haben sich die existierenden Implementierungen nicht durchgesetzt. Des Weiteren bietet diese Form der Erstellung von Software noch eine Vielzahl an Möglichkeiten, die bislang nicht realisiert wurden. Die Akzeptanz professioneller Entwickler wird den Ausschlag geben, ob sich diese Form der Programmierung durchsetzt.

Kapitel 5

Implementierung des AST-basierten Editors

In diesem Kapitel wird der ASTOP Editor vorgestellt, der auf Basis einer einfachen Sprache, eingebettet in Eclipse, eine einfache Variante eines AST-basierten Editors darstellt. Der Editor dient der Vorstellung der Implementierung von Refaktorisierungen auf einem konsistenten Objektmodell. Die Refaktorisierungen sollen so implementiert werden, dass auftretende Inkonsistenzen erkannt und verhindert werden. Er soll Vor- und Nachteile von Refaktorisierungen auf einem AST untersuchen. Bei möglichen Konflikten soll der Editor dem Entwickler verschiedene Lösungsstrategien anbieten.

5.1 Grammatik des Objektmodells

Grundlage des Editors ist eine einfache, funktionale Sprache, mit der typische Programmierkonstrukte wie Schleifen, Anweisungen und Bedingungen erstellt werden können. Die Abbildung 5.1 zeigt das Klassendiagramm des ASTOP Editors.

Zugriff auf den AST eines Programms wird über den *ASTStructureController* gewährt, der die Schnittstelle zur Erstellung von Elementen innerhalb im AST darstellt. Dieser Controller bietet daher Erstellungs-, Update- und Löschmethoden an, die benötigt werden damit der Programmierer innerhalb des Editors Operationen auf dem AST des Programms ausführen kann. Zu diesen Operationen gehört das Bearbeiten des Programms über das User Interface.

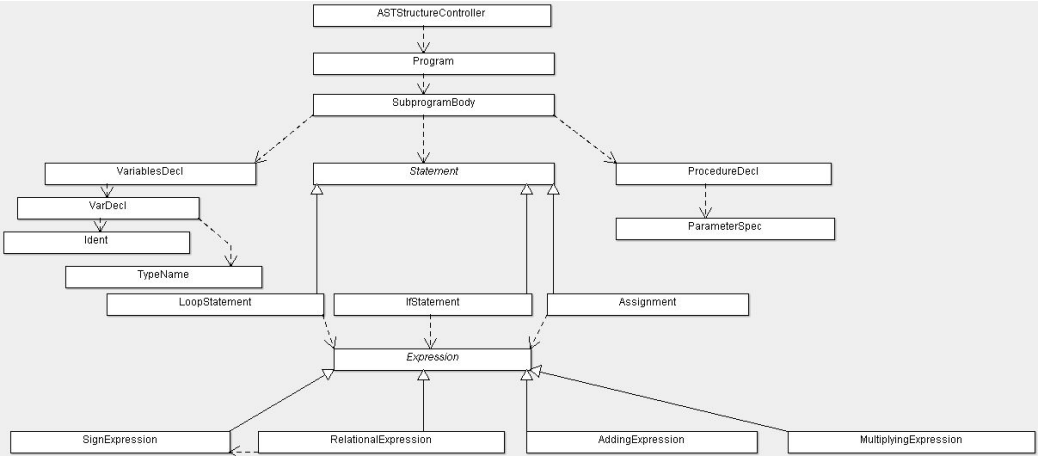


Abbildung 5.1: ASTOP Editor Klassendiagramm

5.2 Implementierung

5.2.1 Implementierung des Objektmodells

Der Editor wurde auf Basis des Eclipse Framework realisiert, da dieses Framework eine Vielzahl von Möglichkeiten bietet, Editoren für unterschiedliche Sprachen zu entwickeln. Im ersten Schritt wurde der ASTOP Editor auf Basis eines *JFace Tree Viewers* realisiert, der über seine Knotenstruktur auf die einzelnen Elemente des Syntaxbaums zugreifen soll. Der Tree Viewer stellt dabei die zentrale Komponente dar, die die Struktur des aktuellen Programms aufzeigt. Neben dieser Ansicht sieht der User eine Detailansicht die, je nach selektiertem Knoten auf dem der Programmierer steht, eine andere Ansicht bietet. Die Detailansicht zeigt die Inhalte der einzelnen Statements an und ermöglicht dem Programmierer diese auch zu ändern. Jeder Knoten innerhalb der Baumstruktur besitzt ein objektspezifisches Kontextmenü, welches dem Anwender verschiedene Operationen anbietet. Hierbei handelt es sich um Bearbeitungsfunktionen des aktuellen Knotens. Es gibt jedoch auch Operationen, die zur Refaktorisierung des gewählten Elements dienen.

Der ASTOP Editor ist auf Basis des Model View Controller (MVC) Muster entwickelt worden. Die Klasse *ASTStructureContoller* stellt die zentrale Schnittstelle zwischen dem Editor und dem Objektmodell der Sprache dar. Der Controller nimmt die Aktionen, die aus dem Editor ausgelöst werden auf, validiert diese und nimmt dann die neuen Elemente in den AST auf. Der Syntaxbaum des aktuellen Programms wird also durch den Controller gehalten. Der Controller besitzt eine Vielzahl von Methoden zum Erstellen, Löschen und Verändern von einzelnen Elementen innerhalb des Baums.

5.2.2 Implementierung des graphischen Editors

Der Editor wurde auf Basis des Eclipse Frameworks realisiert. Die Grafikkomponenten, die zur Erstellung der User Interfaces verwendet wurden, stammen aus der *JFace Bibliothek*. *JFace* bietet bei der Implementierung den Vorteil, dass die einzelnen Grafikkomponenten direkt Objekte aus dem Syntaxbaum der Sprache aufnehmen können. Zum Design der User Interfaces wurde das *Jigloo Plugin* verwendet, welches Programmierer beim Design von User Interfaces unterstützt indem der notwendige Code direkt im Hintergrund generiert wird. Abbildung 5.2 zeigt die Komponenten des User Interfaces.

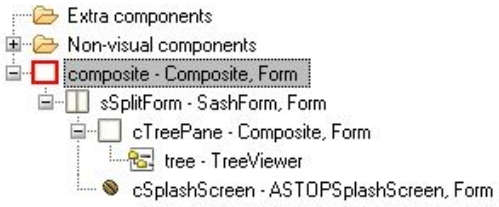


Abbildung 5.2: Erstellen der User Interfaces

Der Editor ist so strukturiert, dass es eine zentrale Ansicht gibt, die auf der linken Seite einen *Tree Viewer* aufnehmen kann, der die aktuelle Struktur des Programms darstellt. Auf der rechten Seite werden die verschiedenen Bearbeitungsdialoge angezeigt, die dem User das Erstellen und Verändern von Elementen anbieten. Bei diesen Bearbeitungsdialogen handelt es sich um eigenständige Klassen, die je nach selektiertem Kontext auf dem AST in der rechten Seite der zentralen View eingeblendet werden.

5.3 Bedienkonzept bei der Durchführung von Refaktorisierungen

Änderung von Elementen

Durch die Auswahl eines Elements werden die Inhalte automatisch in den Dialog geladen und der Anwender kann das Statement ändern. Auch hier gilt, dass nur Änderungen möglich sind, die die Konsistenz des AST nicht gefährden. Elemente können auch gelöscht werden, indem auf dem Kontextmenü des entsprechenden Elements die entsprechende Operation durchgeführt wird. Falls eine Variable gelöscht werden soll, so kann diese nur dann gelöscht werden, wenn sie nicht mehr in einem Statement oder einer Prozedur referenziert wird.

Drag & Drop

Innerhalb der Baumstruktur ist es dem Anwender möglich durch *Drag & Drop* Operationen Änderungen an der Struktur vorzunehmen. Statements oder Definitionen von Variablen können auf entsprechende andere Knoten verschoben werden. *Drag & Drop* prüft zudem, ob die Verschiebung zulässig ist und ob die Rahmenbedingungen für den Wechsel eines Statements stimmen. Hierbei wird die Konsistenz des AST geprüft. Nur wenn diese nicht gefährdet wird, wird die Operation durch den Controller vorgenommen.

Die Laufzeitumgebung

Die Laufzeitumgebung bietet die Möglichkeit, den generierten AST der Sprache zur Ausführung zu bringen. Auf dem Wurzelknoten des Programms kann der Anwender über das Kontextmenü die Funktion *Execute Program* auswählen, die dann die Abarbeitung des Baums startet. Bei der Bearbeitung des Programms wird ein Stack für jede Variable mit ihrem aktuellen Wert angelegt. Durch den sequentiellen Ablauf bei der Programmausführung werden die Werte innerhalb des Stacks immer wieder neu berechnet und gemeinsam in dem Ausgabefenster Console View ausgegeben, welches in Abbildung 5.3 dargestellt wird.

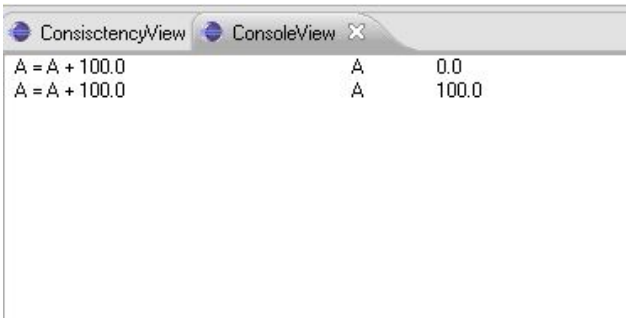


Abbildung 5.3: Console View im ASTOP Editor

5.4 Refaktorisierungen im ASTOP Editor

Die Konsistenz im Editor wird durch den Grundsatz gewährleistet, dass nur modellkonforme Elemente in den AST überführt werden können. Dieser Grundsatz wird auch durch die implementierten Refaktorisierungen befolgt. Ziel ist es, dass das Programm nach jeder Änderung ausführbar bleibt. Im folgenden Abschnitt werden die Refaktorisierungen vorgestellt, die im Rahmen des Editors implementiert sind.

5.4.1 Rename: Umbenennung von Elementen

Rename von Variablen

In einem textuellen Editor müssen bei dem Rename einer Variablen alle Referenzen, die auf diese Definition verweisen geändert werden. Die AST-basierte Programmierung bietet hier einen Vorteil, da der AST der Sprache die einzelnen Referenzen innerhalb des Codes mit der Definition verknüpft. Durch die Veränderung der Definition werden alle Referenzen, die auf diese Definition verweisen, ebenfalls geändert. Abbildung 5.4 veranschaulicht auf Basis des Objektmodells die Definition einer Variablen und deren Referenz in Zuweisungen und Ausdrücken. Nach der Umbenennung sind auch alle Referenzen angepasst. Abbildung 5.5 verdeutlicht diesen Zustand.

Diese Möglichkeit der direkten Umbenennung aller Referenzen innerhalb des AST wird im ASTOP Editor durch eine gemeinsame Klasse gewährleistet. Jede Definition einer Variable im ASTOP Editor besitzt eine Eigenschaft *Bezeichnung*. Diese wird in allen Referenzen innerhalb des AST referenziert. Da es nur eine Definition der Bezeichnung innerhalb des AST gibt, werden bei deren Änderung auch alle Referenzen innerhalb des AST geändert. Durch

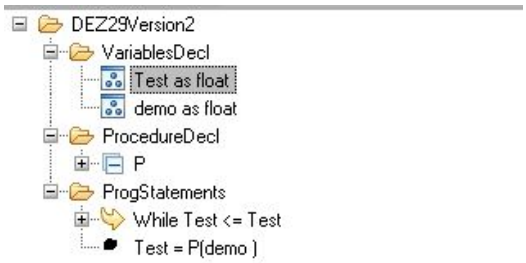


Abbildung 5.4: Umbenennen einer Variable

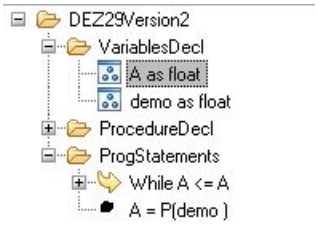


Abbildung 5.5: Update der Referenzen

das Update des grafischen *Tree Viewers* werden auch die geänderten Knoten dem Entwickler zur Anzeige gebracht.

Rename von Prozeduren

Nicht nur Variablen können innerhalb des ASTOP Editors umbenannt werden. Dies gilt auch für Prozeduren, wie in Abbildung 5.6 dargestellt. Nach der Umbenennung sind auch alle Referenzen umbenannt. Abbildung 5.7 verdeutlicht diesen Zustand.

Hierbei kommt das gleiche Prinzip wie bei der Umbenennung von Variablen zum Tragen. Auch Prozeduren besitzen eine Eigenschaft von Typ Bezeichnung, die auch in allen auftretenden Referenzen innerhalb des Baums realisiert werden.

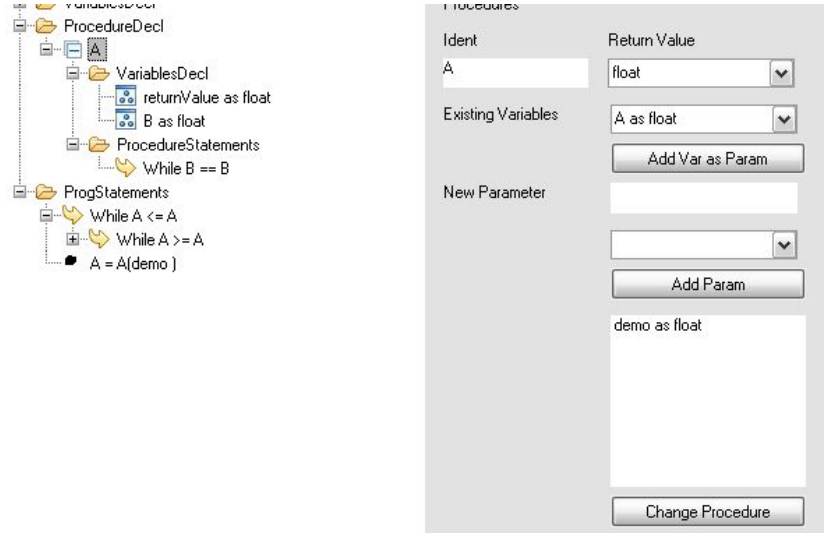


Abbildung 5.6: Umbenennen einer Prozedur

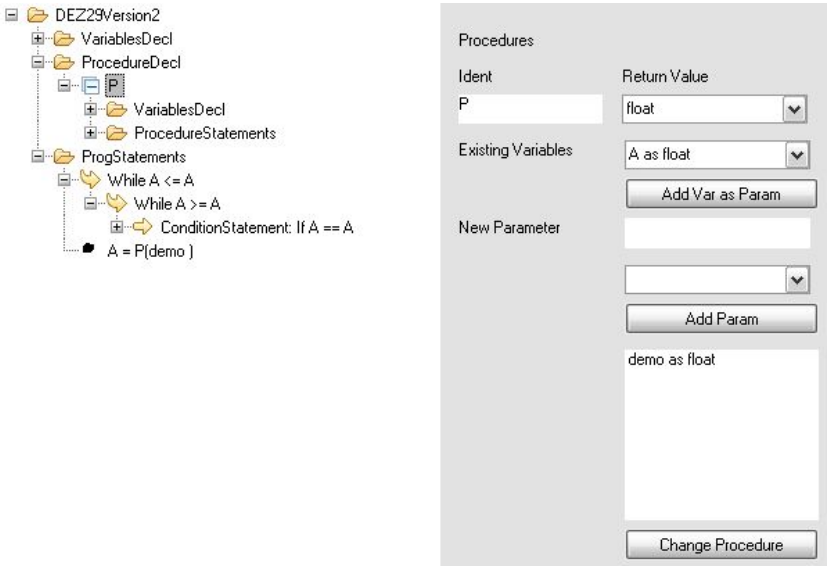


Abbildung 5.7: Umbenennung aller Referenzen

5.4.2 Bearbeiten von Parametern

Bei der Umbenennung von Prozeduren müssen jedoch auch die Anzahl und die Typen der übergebenen Parameter berücksichtigt werden. Der ASTOP Editor deckt dabei folgende Szenarien ab:

- Löschen von übergebenen Parametern
- Hinzufügen von neuen Parametern
- Änderung des Parametertyps.

Löschen von Parametern

Das Löschen von übergebenen Parametern ist in den folgenden Szenarien ohne Benutzerinteraktion möglich. Wenn ein übergebener Parameter innerhalb der Prozedur nicht verwendet wird, dann kann dieser ohne das Einschreiten des Entwicklers gelöscht werden, da die Ausführbarkeit des Syntaxbaums nicht eingeschränkt wird. Wird der Parameter jedoch in der Prozedur verwendet, kann es zwei Möglichkeiten geben um die weitere Ausführbarkeit zu gewährleisten. Zum einen kann verhindert werden, dass der Parameter gelöscht wird und der Anwender wird auf die existierenden Referenzen innerhalb der Struktur hingewiesen. Eine weitere Möglichkeit wäre, dass der Anwender über einen modalen Dialog aufgefordert wird die entsprechenden Stellen innerhalb der Prozedur zu ändern. Der Anwender hätte somit die Möglichkeit, die erzwungenen Änderungen durchzuführen und erst nach einer erneuten Validierung, welche die Ausführbarkeit des Syntaxbaums sicherstellt, würde der übergebene Parameter aus dem Kopf der Prozedur herausgenommen werden. Die zweite Variante stellt eine komplexere Form der Editierung dar, da sichergestellt werden muss, dass der Anwender nur die gefundenen Stellen bearbeiten kann, die in Folge der Konsistenzprüfung angezeigt wurden. Alle weiteren Stellen dürfen nicht bearbeitet werden. Des Weiteren muss die Abhängigkeit der gefundenen Statements nach jeder Bearbeitung erneut geprüft werden. Erst nach einer sauberen Löschung aller Referenzen kann der eigentliche Parameter im Anschluss gelöscht werden. Abbildung 5.8 zeigt die Realisierung der Löschrestriktion im ASTOP Editor.

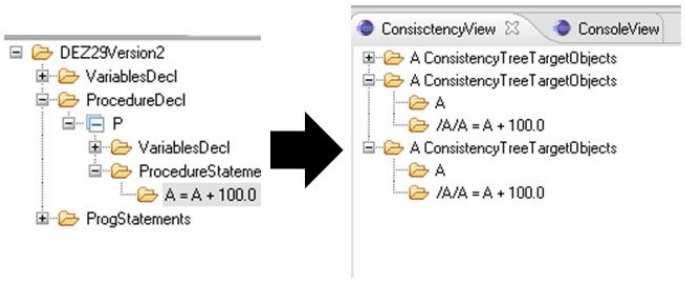


Abbildung 5.8: Löschrestriktion eines Parameters

Hinzufügen von Parametern

Beim Hinzufügen neuer Parameter zu einer Prozedur hat dies keine Auswirkungen auf den Inhalt der existierenden Prozedur. Auswirkungen bei denen der Programmierer eingreifen oder die Konsistenzprüfung von Annahmen ausgehen muss treten bei Prozeduraufrufen auf. Jede Stelle des Programmcodes, die einen Prozeduraufruf beinhaltet, muss um den neuen Parameter erweitert werden. Hier gibt es verschiedene Möglichkeiten dies zu realisieren. Zum einen können die einzelnen Prozeduraufrufe durch Konstanten erweitert werden. Hierbei kann zum Beispiel eine 0 für *Integer* und 0.0 für *Float* Variablen oder auch true / false und leere *Strings* übergeben werden. Diese Operation kann ohne das Eingreifen des Programmierers realisiert werden. Abbildung 5.9 veranschaulicht wie Parameter zu einer bestehenden Prozedur im ASTOP Editor hinzugefügt werden können.

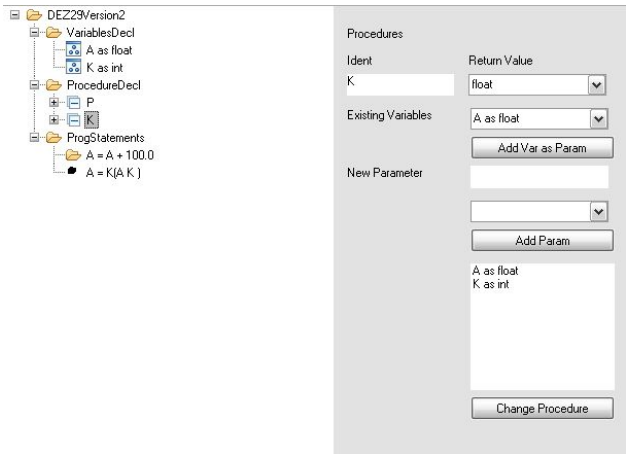


Abbildung 5.9: Hinzufügen eines Parameters zu einer Prozedur

In Abbildung 5.9 wird ein neuer Parameter mit dem Bezeichner K vom Typ *int* der Prozedur hinzugefügt. Damit diese Refaktorisierung durchgeführt werden kann, muss auch der Procedure Call durch einen neuen Parameter erweitert werden. Der ASTOP Editor realisiert diese Refaktorisierung, indem er auf der Ebene des Procedure Calls eine neue Variable vom Typ des neuen Parameters in der Prozedur erzeugt.

Typänderung von Parametern

Eine Änderung des Parametertyps bewirkt Änderungen innerhalb der Prozedur aber auch in den Prozeduraufrufen. Die einfachste Form der Realisierung ist auch hier das Unterbinden, falls festgestellt werden kann, dass diese Änderung die Konsistenz anderer Anweisungen verletzen würde. Eine mögliche Implementierung ist auch hier, dass durch einen entsprechenden Report angezeigt wird warum der Typ nicht geändert werden kann.

Der ASTOP Editor realisiert dies durch eine Verweigerung der Anforderung und durch ein Anzeigen der auftretenden Konflikte in der *Consistency View*. Das Problem könnte auch gelöst werden, indem alle betroffenen Stellen in ihrer Ablaufreihenfolge über modale Dialoge dem Anwender präsentiert werden, so dass dieser die Konflikte Schritt für Schritt beheben kann. Alle diese Operationen auf dem Modell müssten jedoch wie in einer Transaktion behandelt werden, da ein Abbruch inmitten der Bearbeitung die Struktur in ihrer Ausführbarkeit gefährden würde.

5.4.3 Delete: Löschen von Elementen

Delete von Variablen

Das Löschen einer Variable kann dazu führen, dass der Baum in einen inkonsistenten Zustand gerät. Aus diesem Grund muss beim Löschen geprüft werden ob es noch Referenzen gibt. Der ASTOP Editor realisiert diesen Sachverhalt durch das Auflisten der möglichen Konflikte in der *Consistency View*. Abbildung 5.10 zeigt die Ergebnisse einer Löschrestriktion durch die Ausgabe der Ergebnisse in der *Consistency View*.

Auch bei diesem Anwendungsfall besteht wie bereits weiter oben beschrieben die Möglichkeit sich alle Statements vorzunehmen, die auf diese Variable verweisen, um somit die Konflikte manuell durch den Programmierer

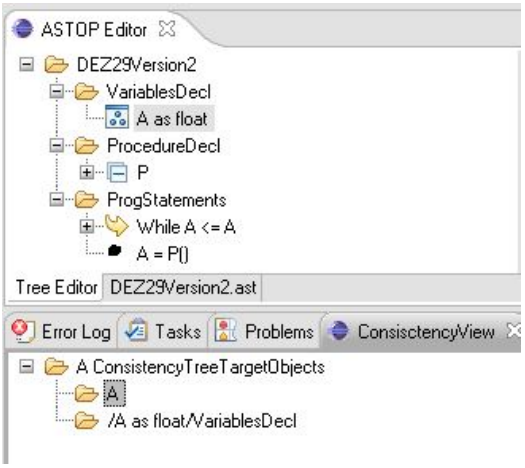


Abbildung 5.10: Delete einer Variable mit Referenzen

vor dem eigentlichen Löschvorgang zu lösen. Auch hier gilt natürlich, dass die Operationen in einer Transaktion geklammert werden müssten, so dass der ursprüngliche Zustand bei Abbruch oder Fehler wiederhergestellt werden könnte.

Delete von Prozeduren und Prozeduraufrufen

Das Löschen von Prozeduraufrufen kann auf zwei verschiedene Arten realisiert werden. Das Löschen einer Prozedur kann auch bewirken, dass alle Referenzen auf diese Prozedur gelöscht werden, da sie durch das Löschen der Quelle nicht mehr verwendet werden. Da innerhalb des ASTOP Editors Prozeduraufrufe als einzelne Statements dargestellt werden, kann dies ohne die Konsistenz des Baums zu gefährden durchgeführt werden. Der ASTOP Editor prüft an dieser Stelle jedoch wie in Abbildung 5.11 zu sehen die existierenden Abhängigkeiten und zeigt die Konflikte beim Löschen einer Prozedur auf, wenn noch referenzierte Prozeduraufrufe existieren. Prozeduraufrufe können direkt aus dem Editor gelöscht werden, da an dieser Stelle keine Konflikte in der Konsistenz des Syntaxbaums auftreten können.

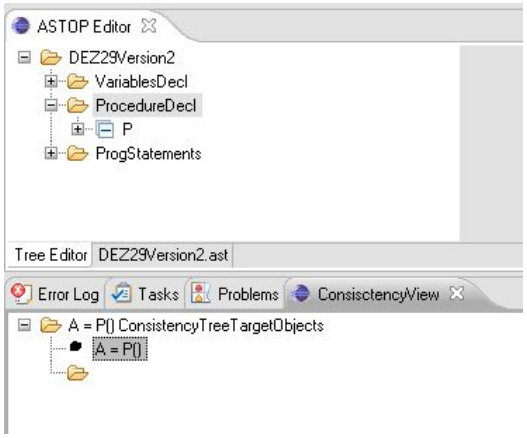


Abbildung 5.11: Validierung von Referenzen auf eine Prozedur

5.4.4 Move: Verschieben von Elementen

Move steht für das Verschieben von Variablen, Statements oder Prozeduraufrufen in einen anderen Kontext. Der ASTOP Editor bietet diese Möglichkeiten über *Drag & Drop* Funktionalitäten. Es kann also ein beliebiges Element ausgewählt werden, welches dann an eine andere Stelle verschoben werden kann. Im folgenden werden mögliche Operationen und darauf basierende Konsistenzprüfungen beschrieben.

Move von Variablen

Variablen können zwischen der Hauptprogrammebene und den Prozedurebenen verschoben werden. Dies ist vor allem dann sinnvoll, wenn die Sichtbarkeit der Variablen vergrößert oder verkleinert werden soll. Beispielsweise kann die Sichtbarkeit auf die Prozedurebene eingeschränkt werden, falls diese nicht im globalen Kontext des Programms benötigt wird. Des Weiteren kann aber durch ein *Move Up* der Kontext auch erweitert werden, so dass eine Variable globale Sichtbarkeit auf Programmebene erlangt. Abbildung 5.12 und 5.13 zeigen das Verschieben einer Variablen zwischen Prozedur- und Programmebene.

Bei der Reduzierung der Sichtbarkeit durch eine Verschiebung auf Programmebene muss natürlich der aktuelle Kontext der Variablen geprüft werden, bevor die Verschiebung durchgeführt wird. Falls Abhängigkeiten gefunden werden, so muss das Verschieben verboten werden und der Anwender muss über diesen Vorgang informiert werden. Der ASTOP Editor erkennt die

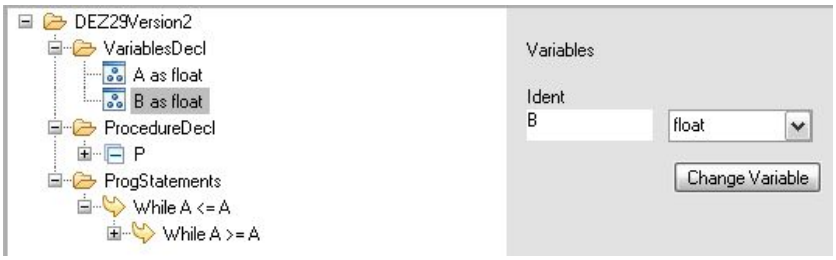


Abbildung 5.12: Verschieben einer Variable

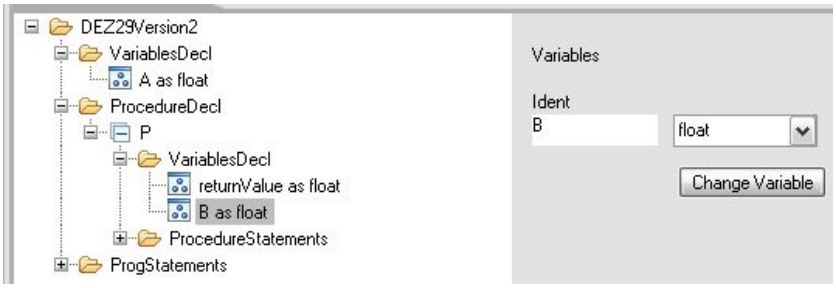


Abbildung 5.13: Platzieren einer Variable an einer neuen Stelle

Abhängigkeiten des aktuellen Kontext und zeigt die gefundenen Referenzen innerhalb der *Consistency View* an.

Move von Anweisungen und Statements

Anweisungen können ohne Prüfung aus dem aktuellen Kontext herausgelöst werden. Jedoch muss vor dem Überführen in den neuen Kontext geprüft werden, ob die verwendeten Variablen innerhalb der selektierten Statements auch im Zielkontext zur Verfügung stehen. Nur falls dies der Fall ist, kann eine Anweisung von einem Kontext ohne Interaktion in einen anderen Kontext überführt werden. Dieser Sachverhalt könnte auch optimiert werden, indem zeitgleich der Operation des Verschiebens auch der Gültigkeitsbereich der Variablen angepasst wird, so dass der neue Kontext in den die Anweisung überführt wird auch die Rahmenbedingungen aufweist, die zur Ausführung benötigt werden. Der ASTOP Editor verhindert das Verschieben und blendet die auftretenden Konflikte in der *Consistency View* ein. Abbildung 5.14 zeigt die Durchführung dieser Operation in dem Editor. Anschließend kann das Statement an einer neuen Stelle wieder eingefügt werden. Abbildung 5.15 zeigt das Einfügen des Statements in einen neuen Kontext.

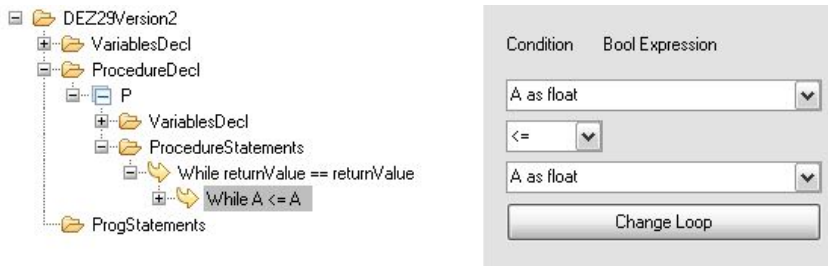


Abbildung 5.14: Verschieben eines Statements

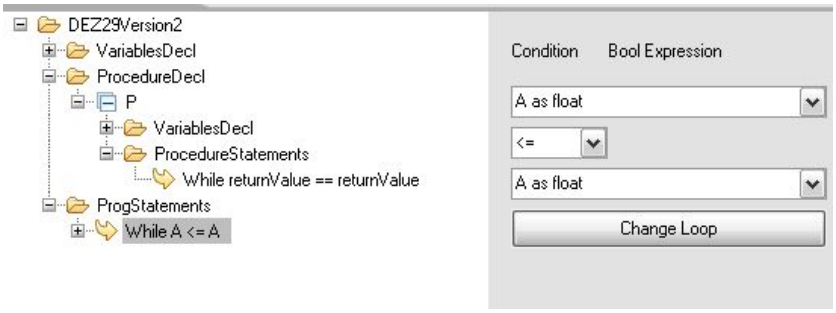


Abbildung 5.15: Erstellen des Statements

Bedingungen und Schleifen können eine Vielzahl von Kindelementen besitzen. Bei der Verschiebung des Kontextes müssen diese auch in den neuen Kontext verschoben werden. Somit müssen die beschrieben Prüfungen ebenfalls für jedes Kindelement durchgeführt werden. Falls lediglich ein Element inkonsistent wäre muss die Operation verhindert werden oder der Gültigkeitsbereich für alle enthaltenen Referenzen ebenfalls verschoben oder erweitert werden.

Move von Prozeduraufrufen

Das Verschieben von Prozeduraufrufen erfordert die Prüfung sowohl auf die für den Rückgabewert angesprochene Variable als auch die Parameter, die innerhalb des Statements mitgegeben werden. Wie bereits oben beschrieben müssen für Variablen und Parameter die neuen Kontexte vorbereitet sein, ansonsten muss das Verschieben unterbunden werden und eine entsprechende Warnmeldung muss in der *Consistency View* ausgegeben werden.

5.4.5 Extract to Procedure

Der ASTOP Editor realisiert auch die Refaktorisierung *Extract to Procedure*. Im Rahmen dieser Refaktorisierung wird ein Statement oder eine Menge von Statements selektiert und in eine neue Prozedur überführt. Hierbei muss natürlich berücksichtigt werden, dass die Variablen und Konstanten der selektierten Statements auch in der neu zu erstellenden Prozedur vorhanden sind. Der ASTOP Editor geht von einer Annahme aus, dass auch die Konstanten bei der Erstellung der *Procedure Calls* in die neue Prozedur mitgegeben werden. Im folgenden Beispiel wird *1000* im Prozeduraufruf als Konstante mitgegeben, aber in der Prozedur als Parameter *V1000* verwendet. Hier wäre aber auch möglich, dass die Konstante nicht mitgegeben wird und nur die Zahl innerhalb der neuen Prozedur verwendet wird. Die Refaktorisierung trifft also eine Annahme, so dass der Baum nach der Operation in einem konsistenten Zustand bleibt. Eine weitere Möglichkeit wäre, eine Variable in dem zu extrahierenden Teilabschnitt erstmalig und einzig zu verwenden. Darüber hinaus kann diese Entscheidung dem Programmierer als Konfliktlösung vorgeschlagen werden. Abbildung 5.16 zeigt die Realisierung dieser Refaktorisierung im ASTOP Editor. Hierbei wird der User durch einen Dialog dazu aufgefordert, dass er eine neue Prozedur definieren muss.

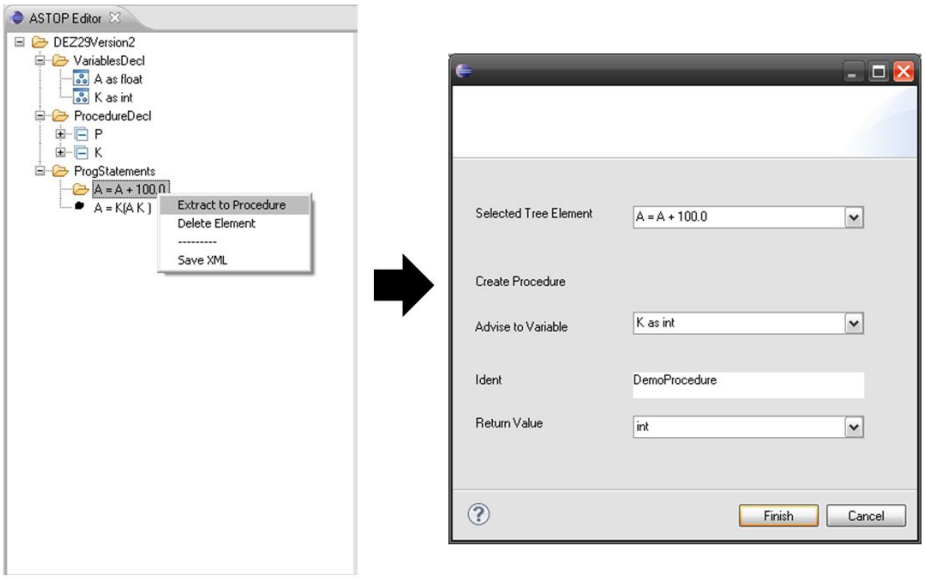


Abbildung 5.16: Refaktorisierung Extract to Procedure

5.5 Zusammenfassung

Der ASTOP Editor zeigt auf, wie Refaktorisierungen von multiplen Referenzen innerhalb eines Programms auf Basis eines AST gelöst werden. Wie bereits festgestellt, müssen in diesem Zusammenhang auch auftretende Konflikte analysiert werden. Hier können unterschiedliche Konfliktlösungstechniken in Betracht gezogen werden. Zum einen kann der Benutzer durch die Vorstellung der Konflikte aufmerksam gemacht werden und die Aktion verhindert werden, um die Konsistenz des Syntaxbaums nicht zu gefährden. Zum anderen können die Konflikte Schritt für Schritt im Rahmen einer Transaktion bearbeitet werden, um somit zu gewährleisten dass auch nach der Auflösung der Konflikte ein konsistenter Zustand beibehalten wird. Es muss allerdings auch bei Bearbeitungsfehlern und bei Systemabbrüchen sichergestellt werden dass der Ursprungszustand wiederhergestellt werden kann.

AST-basierte Programmierung bietet vor allem beim Update von Referenzen einen wesentlichen Bearbeitungsvorteil. Abhängige Stellen im Code können mit der Veränderung der Definition direkt angepasst werden. Somit werden versehentliche Fehler reduziert, die durch anpassen der existierenden Referenzen verursacht werden können. Besitzen jedoch die Referenzen bereits weitere Abhängigkeiten, so müssen diese wiederum analysiert und ebenfalls im Rahmen der Refaktorisierung verändert werden. Auf Basis der durchgeführten Analyse ist anzumerken, dass mit steigender Komplexität die Bearbeitung der Refaktorisierungen erschwert wird, da der Aufwand zur Analyse der bestehenden Abhängigkeiten steigt. Ein Beispiel hierfür ist das Löschen eines Parameters in einer Prozedur, der innerhalb der Prozedur bereits verwendet wird. Dies betrifft auch die Änderung eines Variablentyps, der innerhalb des Programms an verschiedenen Stellen verwendet wird. Um diese Refaktorisierungen durchzuführen bedarf es einer Möglichkeit im Rahmen einer Transaktion, die für den Entwickler durch modale Dialoge gekennzeichnet sein kann, alle auftretenden Konflikte nacheinander zu lösen, so dass schlussendlich nach Abschluss der Transaktion der Syntaxbaum wieder in einem konsistenten Zustand vorliegt. Bei größeren Refaktorisierungen, bei denen zum Beispiel komplette Strukturen des Programms auf ein Entwicklungsmuster umgestellt werden, kann man sich demnach vorstellen, dass der Aufwand zur Prüfung dieser Refaktorisierungen mehr und mehr steigt. Somit stellen Refaktorisierungen in AST-basierten Editoren nur bis zu einem gewissen Komplexitätsgrad einen Bearbeitungsvorteil dar.

Kapitel 6

Ausblick und Zusammenfassung

Zusammenfassung

Wie bereits im Kapitel zum Thema AST-basierte Implementierungen untersucht wurde, sind lediglich in einigen wenigen Editoren Funktionen zur Durchführung von Refaktorisierungen enthalten. Hauptsächlich beschränkten sich diese auf das Umbenennen und Löschen von Konstrukten in der Sprache. Refaktorisierungen in einem AST-basierten Editor müssen konsistent sein, dass bedeutet, dass der Code nach der Refaktorisierung ausführbar bleibt und seine Funktionsweise sich nicht geändert hat. Der prototypische ASTOP Editor dient der Vorstellung der Implementierung dieser Refaktorisierungen auf einem konsistenten Modell, da dies auf Basis der bestehenden Implementierungen nicht nachvollziehbar ist. Ziel war es, auf einem AST-basierten Editor vollständige Refaktorisierungen zu implementieren um die Möglichkeiten, aber auch die Schwierigkeiten der Implementierung auf einem AST herausstellen zu können. Der ASTOP Editor realisiert zu diesem Zweck eine einfache AST-basierte Entwicklungsumgebung, bei dem die Refaktorisierungen über das User Interface aufgerufen werden können, ein Controller führt die Validierung der angestrebten Refaktorisierung durch und Warnungen werden dem Programmierer über eine spezielle Ansicht mitgeteilt. Die Refaktorisierungen werden direkt auf dem Modell ausgeführt, so dass keine Inkonsistenzen entstehen. Das Kapitel stellt dabei nicht nur die realisierten Refaktorisierungen dar, sondern spricht auch mögliche Erweiterungen oder andere Formen der Realisierung an. Hierbei entsteht die Erkenntnis, dass komplexere Refaktorisierungen in einzelne, in sich konsistente Teilschritte zerlegt werden müssen, bei der jeder einzelne Schritt die Konsistenz des Syntaxbaums nicht verletzt. Es muss möglich sein, diese Teilschritte jederzeit rückgängig zu machen um den Ausgangszustand wieder erreichen zu können. Ein wesentlicher Vorteil der Zerlegung der Refaktorisierungen in einzelne Teilschritte ist die damit

verbundene konsistente Durchführung derselben auf dem AST. Im Rahmen der Untersuchung der Implementierung dieser Refaktorisierungen ist aber auch aufgefallen, dass es schwieriger ist komplexe Refaktorisierungen zu implementieren. Dies ist dem Umstand geschuldet, dass keine Freiheitsgrade bei der Durchführung von Refaktorisierungen vorhanden sind. Es lassen sich also nur Schritte im Rahmen des Refaktorisierungsprozesses durchführen, die die Konsistenz des AST nicht beeinflussen. Dies stellt gerade bei der Durchführung von großen Refaktorisierungen einen erheblichen Nachteil dar, da in diesem Zusammenhang oft nicht alle Teilschritte eindeutig bekannt sind und dies zu Inkonsistenzen führen kann. AST-basierte Refaktorisierungen sind also bei der Durchführung kleinerer Refaktorisierungen möglich und nützlich, stellen jedoch bei größeren Refaktorisierungen ein Hindernis dar.

Ausblick

AST-basierte Editoren öffnen eine Vielzahl von Möglichkeiten, um die schnelle und leichtere Anwendungsentwicklung voranzutreiben. Gerade bei einfachen und schnell zu lösenden Problemstellungen wirken sich die positiven Effekte aus, die mit diesem Ansatz der Programmierung gewonnen werden können. Rechnersysteme sind heute so stark hinsichtlich ihrer Rechenleistung, dass die Konstruktion des AST und die auf ihm durchgeführten Operationen sehr schnell und leicht im Hauptspeicher vorgenommen werden können. Die Entwicklungen im Rahmen dieser Arbeit haben gezeigt, welcher Aufwand erzeugt werden muss um Editoren zu erstellen, die auf einem AST arbeiten. Dies bedeutet, je komplexer die Applikationen und die verwendeten Technologien werden, desto schwieriger ist es, diese durch einen AST-basierten Editor zu erstellen, da die Komplexität durch die entsprechenden Prüfungen im Editor aufgefangen werden muss. Zwangsläufig kommt es zu einem erhöhten Aufwand bei der Erstellung derartiger Editoren. Änderungen in der Sprache müssen unmittelbar nach ihrem Erscheinen in die AST-basierten Editoren implementiert werden. Schaut man sich die Ergebnisse dieser Untersuchungen an, so fällt auf, dass die Editoren und Sprachen, die für eine bestimmte Anwendungsdomäne realisiert wurden, die meisten Chancen hatten sich am Markt durchzusetzen. Scratch hat sich als Entwicklungsumgebung zum spielerischen Erlernen der Programmierung herausgestellt und in diesem Bereich eine recht hohe Verbreitung erreicht. SharePoint Designer kann nicht als echter AST-basierter Editor gesehen werden, da er lediglich Konfigurationsfiles für die Workflow Engine in SharePoint generiert, doch seine Einfachheit und seine Resistenz gegenüber Syntaxfehlern helfen versierten Benutzern schnell Applikationen auf Basis von Workflows zu erstel-

len. MPS ist die erste vollständige Realisierung eines AST-basierten Editors, der die Erstellung von Software in einer gängigen Sprache unterstützt. Die Kernsprache muss als Sprachpaket vorliegen, so dass ein Programmierer auf dieser Basis Code in der Sprache erzeugen kann. Das bedeutet, dass eine bekannte Sprache erst nach MPS portiert werden muss bevor mit ihr entwickelt werden kann. Der Hauptzweck von MPS liegt aber nicht in den Anwendungsentwicklung auf Basis einer existierenden Sprache, sondern auf der Erstellung von domänenspezifischen Sprachen. Die AST-basierte Programmierung stellt ihre Vorteile besonders bei der Hervorhebung von semantischen Referenzen heraus, die im Rahmen der Visualisierung von Zusammenhängen und bei Refaktorisierungen gebraucht werden. Dies ist leicht zu realisieren und stellt für den Programmierer einen großen Nutzen dar. Bei der Erstellung von Ablauflogik im Detail ist die textuelle Programmierung sehr ausdrucksstark und das Parsen von kleineren Blöcken auch nicht besonders aufwendig. In diesem Zusammenhang ist eine Kombination aus AST-basierter Programmierung zur Erstellung der Strukturen komplexer Systeme und der textuellen Erstellung der Ablauflogik im Detail ein mögliches Szenario zur Kombination der Stärken beider Formen der Programmierung.

Abbildungsverzeichnis

Literatur

[AB06] J.K. Andrew und A.M. Brad. *Enabling Innovation in Code Editors*. 2006. URL: http://faculty.washington.edu/ajko.

[Aho+08] V. Aho u. a. *Compiler Prinzipien, Techniken und Werkzeuge*. Bd. 2. Pearson Studium, 2008.

[Bei10] F. Beister. *Literate Programming für mehrsprachige Spezifikationen in einer Eclipse Entwicklungsumgebung*. 2010. URL: http://www.mendeley.com/research/literate-programming-fr-mehrsprachige-spezifikationen-einer-eclipseentwicklungsumgebung/.

[Ber11] E. Bernstein. *Entwurf und Implementierung der Konfigurations und Präsentationskomponente eines entscheidungsunterstützenden Systems für die computergestützte Evaluation der kindlichen Spontanmotorik*. 2011. URL: http://opus.bsz-bw.de/hshn/volltexte/2011/6/pdf/Diplomarbeit_EugenBerenstein.pdf.

[Chr10] M Christerson. *Intentional Sofware at Work*. 2010. URL: http://www.infoq.com/presentations/Intentional-Software-at-Work.

[Edw07] J. Edwards. *Uncovering the Simplicity of Conditionals*. 2007. URL: http://subtextual.org.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Bd. 1. 1999.

[GG00] K. Guenther und I. Guenther. *Lava An Experimental Object-Oriented RAD Language with Virtual Types and Component Integration Support*. 2000. URL: http://lavape.sourceforge.net.

[Goo09] Google. *App Inventor for Android*. http://info.appinventor.mit.edu/. 2009.

[Gün00] K. D. Günther. *Lava: Programmieren im Lego-Stil*. 2000. URL: http://ftp.heanet.ie/mirrors/sourceforge/l/project/la/lavape/Lava%20Papers/LavaLego.pdf.

[Huh08] Dr.M. Huhn. *Compiler Vorlesung*. 2008. URL: http://www.ips.tu-braunschweig.de/wwwalt/images/ss08/compiler_I/VL/cp2.pdf.

[Jet] JetBrains. *Jet Brains Meta Programming System*.

[Kel02] J. Kelso. *A Visual Programming Environment for Functional Languages*. 2002. URL: http://www.csse.uwa.edu.au/~joel/vfpe/index.html.

[Lau05] R. Laue. *Code Konvetionen*. 2005. URL: http://ebus.informatik.uni-leipzig.de/www/media/lehre/seminar-javatools05/semtools05-plushnikov-text.pdf.

[Lei10] T. Leinbach. *Entwicklung und Anwendung von Informations- und Kommunikationstechnologie zwischen den 1950ern und heute*. 2010. URL: http://edoc.ub.uni-muenchen.de/12436/1/Leimbach_Timo.pdf.

[LeN] J. LeNormand. *Rathereasy*. http://sites.google.com/site/rathereasy/.

[Log06] Star Logo. *User Guide Star Logo*. 2006. URL: http://education.mit.edu/webdav/Quick%20Start%20Guide/SLTNG%20Quick%20Start%20Guide%20102308.pdf.

[Mal10] J. et al. Malony. *The Scratch Programming Languag and Environment*. 2010. URL: http://web.media.mit.edu/~jmaloney/papers/ScratchLangAndEnvironment.pdf.

[Mic10] Microsoft. *Evaluierungshandbuch Microsoft SharePoint Designer 2010*. 2010. URL: http://www.microsoft.com/downloads/de-de/details.aspx?FamilyID=cffb14e8-88a9-43bd-87aa-4792ab60d320.

[MIT] MIT. *Scratch Imagine program share*.

[Mön09] J. Mönig. *Syntax Elements for Smalltalk*. 2009. URL: http://www.chirp.scratchr.org/blog/wp-content/uploads/2009/03/elements090224.pdf.

[Ort96] A. Ortmann. *Modellierung von Abhängigkeitsgraphen*. 1996. URL: http://www-sfb288.math.tu-berlin.de/~armin/study/stud.html.

[Ose07] K. Osenkov. *Designing, implementing and integrating a structured C code editor*. 2007. URL: http://www.osenkov.com/diplom.

[Pau+95] R Pausch u. a. *A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality*. 1995. URL: http://www.cs.cmu.edu/~stage3/publications/95/journals/IEEEcomputer/CGandA/paper.html.

[Pin00] S. Pingel. *Abstrakte Syntaxbäume*. 2000. URL: http://steffenpingel.de/files/papers/ast.pdf.

[PL09] S Prokhorenko und V Laptev. *Visual Development Environment without source code, based on the dialect of Oberon*. 2009.

[Pol07] S. Poll. *Lexikalische Analyse*. 2007. URL: http://www.wi.uni-muenster.de/pi/lehre/ws0607/seminarCB/ausarbeitungen/02-LexikalischeAnalyse.pdf.

[Res06] H. Restel. *Automatisches Erkennen von Trial-and-Error Episoden beim Programmieren*. 2006. URL: http://www.inf.fu-berlin.de/inst/ag-se/theses/Restel06-trial-error.pdf.

[Roq07] R.V Roque. *Open Blocks: An Extendable Framework for Graphical Block Programming Systems*. 2007. URL: http://dspace.mit.edu/bitstream/handle/1721.1/41550/220927290.pdf.

[Sd10] M. Schäfer und O. deMoor. *Specifiying and Implementing Refactorings*. 2010. URL: http://researcher.ibm.com/researcher/files/us-mschaefer/specifying.pdf.

[Voe12] R. Voeller. *Compiler im Überblick*. 2012.

[Völ10] R. Völler. *Formale Sprachen und Compiler*. 2010. URL: http://users.informatik.haw-hamburg.de/~voeller/fc/comp/node3.html.