Universität Passau

Fakultät für Informatik und Mathematik

Masterarbeit

Feature-orientierte Entwicklung von Programmiersprachwerkzeugen: Eine Fallstudie

Verfasser:

Rolf Daniel

28. Februar 2012

1. Gutachter:

Dr.-Ing. Sven Apel

2. Gutachter:

Prof. Dr. Christian Lengauer

Universität Passau
Fakultät für Informatik und Mathematik
Innstraße 33, D–94032 Passau

Daniel, Rolf:
Feature-orientierte Entwicklung von Programmiersprachwerkzeugen: Eine Fallstudie
Masterarbeit, Universität Passau, 2011.

Zusammenfassung

Diese Masterarbeit befasst sich mit dem Thema einer Feature-orientierten Entwicklung von Programmiersprachen. Ausgangspunkt dafür ist die Sprachdefinition einer Programmiersprache, die sich aus Syntax-, Typ- und Semantikregeln zusammensetzt.

Diese Sprachdefinition wird auf mögliche Features hin untersucht. Die sich daraus ergebenden Features werden modularisiert, um eine möglichst hohe Wiederverwendbarkeit zu erreichen, mit dem Ziel, eine Familie von Programmiersprachen mit entsprechenden Werkzeugen, wie z.B. Parser oder Pretty Printer, erstellen zu können.

Zwei Fallstudien, bestehend aus einer Sprache für Ausdrücke und der frei verfügbaren Sprache MoBL, sollen die Anwendbarkeit dieses Ansatzes demonstrieren, einen näheren Einblick in die Feature-orientierte Entwicklung von Programmiersprachwerkzeugen geben und die Grundlage für weiterführende Forschungen auf diesem Gebiet bilden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Kapitel 1

Einleitung

Softwareprogrammiersprachen sind im Bereich der Softwareentwicklung weit verbreitet und umfassen neben Universalsprachen, wie Java oder C, auch domänenspezifische Sprachen, Modellierungs- und Spezifikationssprachen und Application Programming Interfaces (APIs). Das Software-Language Engineering befasst sich mit dem Design, der Implementierung, der Wartung und der Weiterentwicklung dieser Softwareprogrammiersprachen.

Die meisten Softwareprogrammiersprachen sind einer schrittweisen Entwicklung unterworfen, was zur Folge hat, dass sie fortlaufend erweitert und verfeinert werden. Dadurch entstehen eine Vielzahl an Sprachvarianten und -versionen, z.B. Java 1.4 und Java 1.5. Um die Entwicklung dieser Varianten und Versionen systematisch managen zu können, wird der Ansatz der **Produktfamilie** bzw. **Produktlinie** – wird in Abschnitt 2.2.1 genauer definiert – verfolgt.

Eine Produktfamilie besteht aus einer Menge von Sprachen, die alle eine grundlegende Anzahl an Sprachfeatures gemeinsamen haben und sich, darüber hinaus, in bestimmten, zusätzlichen Sprachfeatures unterscheiden. Produktfamilien und Produktlinien zielen

- auf eine hohe Wiederverwendbarkeit der Sprachfeatures,
- eine hohe Variabilität und
- eine automatisierte Spezifikation einer Sprache, abhängig von der Auswahl ihrer Features durch den Benutzer, ab.

Meist wird eine Sprachenfamilie für eine bestimmte Domäne entwickelt, wobei die Features der Sprachenfamilie Abstraktionen der Domäne widerspiegeln. In dieser Arbeit wird der Ansatz der Produktfamilie zur praktischen Entwicklung von unterschiedlichen Sprachdefinitionen genutzt. Dies wird durch mehrere Werkzeuge, wie z.B. FeatureHouse und die Spoofax Workbench, unterstützt und anhand zweier Fallstudien untersucht.

Im Gegensatz zur bisherigen Komposition bzw. Dekomposition von Sprachen, die sich nur mit der Syntax der Sprache beschäftigt hat, werden hier alle Aspekte einer Sprachdefinition berücksichtigt. Dazu gehören, neben Syntaxregeln, auch Typ- und Semantikregeln. Hierfür werden eine Reihe von Sprachen und Werkzeugen benötigt, die diese Regeln anhand der Features der Sprache zerlegen:

- SDF und Stratego für die Spezifikation einer Sprache

- Spoofax für die Generierung von Sprachwerkzeugen, wie z.B. Parser, Typsystem, Codegenerator und Editor
- FeatureHouse für die Komposition von Sprachfeatures

In zwei Fallstudien werden diese Werkzeuge dazu genutzt, bestehende Sprachen in Features zu zerlegen und diese dann in unterschiedlichen Kombinationen zu komponieren, um verschiedene Varianten von Sprachen zu erstellen.

1.1 Motivation

Der Ansatz der Produktfamilie, der in dieser Arbeit verfolgt wird, ist durch mehrere Aspekte motiviert. Heutige Programmiersprachen besitzen häufig eine sehr hohe Komplexität und beinhalten eine Vielzahl an Features, die in vielen Anwendungsbereichen gar nicht benötigt werden.

Sie können sogar einen negativen Effekt haben, was die Performanz oder ein Fehlverhalten betrifft. Daher ist es manchmal sinnvoll, dass ein Benutzer für seinen Anwendungsbereich nicht benötigte Features aus der Sprachdefinition entfernen kann. Bei einem solchen Downsizing kann die Komplexität durch eine Reduktion der Sprachmächtigkeit erheblich verringert werden.

Ein weiterer Aspekt ist das Experimentieren mit unterschiedlichen Sprachfeatures. Mit den generierten Sprachvarianten können unterschiedliche Szenarien getestet werden und so kann eine optimale Auswahl an Features für eine spezielle Anwendung gefunden werden. Desweiteren können die verschiedenen Varianten in diversen Bereichen eingesetzt werden, um maßgeschneiderte Anwendungen zu entwickeln.

Durch eine schrittweise Entwicklung von Sprachvarianten können immer neue Features hinzugefügt werden, die die bestehende Sprachfamilie erweitern und verbessern. Es kann auch sein, dass Features, die sich nicht bewährt haben, wieder entfernt werden.

Ein letzter Gesichtspunkt ist die Komposition von Domain Specific Languages (DSLs). DSLs werden für spezielle Domänen entwickelt und haben u.a. die Vorteile einer geringeren Redundanz und einer besseren Lesbarkeit. Features unterschiedlicher DSLs können miteinander komponiert werden und somit neue DSLs bilden, die in verschiedenen Domänen genutzt werden können.

1.2 Zielsetzung

Im Rahmen dieser Masterarbeit soll eine Feature-orientierte Entwicklung von Programmiersprachwerkzeugen anhand zweier Fallstudien näher untersucht werden. Das Hauptziel ist, die unterschiedlichen Dokumente, die eine Sprachdefinition ausmachen – dazu gehören Syntaxregeln, Typregeln und Semantikregeln, auf eine deklarative Art und Weise auszudrücken.

Dadurch soll eine Erweiterung und Verfeinerung der Sprachdefinition, bedingt durch individuelle Features, ermöglicht und vereinfacht werden. Werkzeuge wie

Parser, Typsystem, Codegenerator und Editor werden dann nach Bedarf automatisch generiert.

1.3 Gliederung der Arbeit

Nach einer kurzen Einleitung durch Motivation und Zielstellung dieser Masterarbeit werden in Kapitel 2 grundlegende Konzepte vorgestellt. Es erfolgt unter anderem eine kurze Einführung in den Compilerbau, eine Übersicht über die benötigten Bestandteile einer Sprachdefinition und ein Überblick zur Feature-Orientierung. Es werden wichtige Begriffe wie Sprachprozessoren, Syntax, Semantik und Typsystem, sowie Produktlinie und Feature vorgestellt. Desweiteren werden verschiedene Vorgehensmodelle zur Erstellung einer Produktlinie beschrieben und das Framework FeatureHouse[1] genauer betrachtet.

Darauf folgt Kapitel 3, in dem der Ansatz des Feature-Oriented Software Engineering (FOSE) vorgestellt wird, der aus einem schrittweisen Vorgehen besteht. Dazu gehören z.B. die Refaktorisierung der Sprachdefinition in Features und die Abbildung der Features auf entsprechende Implementierungseinheiten, um nur zwei von den benötigten Schritten zu nennen. Zusätzlich werden einige Gründe präsentiert, die für einen solchen Feature-orientierten Ansatz sprechen.

Anschließend wird in Kapitel 4 näher auf die Implementierung eingegangen und wie mit Hilfe von FeatureHouse und der Spoofax Workbench neue Sprachen Feature-orientiert entwickelt werden können. Dabei wird für die Beschreibung der Sprachgrammatik SDF genutzt und für die Analyse und Transformation von Programmen wird die Sprache Stratego verwendet.

In Kapitel 5 werden zwei Fallstudien und die damit erzielten Ergebnisse vorgestellt. Bei der ersten Fallstudie handelt es sich um eine Sprache für arithmetische Ausdrücke, die neu definiert wurde. Die Sprache MoBL bildet die zweite Fallstudie. Sie ist eine frei verfügbare Programmiersprache für eine einfache und schnelle Entwicklung von Applikationen für mobile Geräte, wie z.B. Handys oder Tablet-PCs. Für beide Sprachen wird eine Feature-Lokalisation und -Dekomposition durchgeführt, um die jeweilige Sprachdefinition in verschiedene Features aufzuteilen. Anschließend werden die Features mit Hilfe von FeatureHouse zu unterschiedlichen Varianten von Sprachdefinitionen komponiert.

Am Ende der Arbeit werden in Kapitel 6 noch einmal die wichtigsten Ergebnisse, vor allem im Hinblick auf die beiden Fallstudien, zusammengefasst.

[1] FeatureHouse: http://www.fosd.de/fh

Kapitel 2

Grundlagen

In diesem Kapitel werden einige grundlegende Konzepte vorgestellt, die dem Leser einen besseren Einblick in die Thematik dieser Masterarbeit verschaffen sollen.

Der folgende Abschnitt befasst sich mit Softwaresystemen, die für die Übersetzung von Programmen zuständig sind, so dass diese auf einem Computer ausgeführt werden können. Diese Softwaresysteme werden auch als **Compiler** bezeichnet. Die hier vorgestellten Informationen sind aus dem 1. Kapitel von [ALSU08] übernommen.

2.1 Compilerbau

2.1.1 Sprachprozessoren

Ein Compiler lässt sich vereinfacht als

> „ein Programm, das ein Programm in einer Sprache – der **Quellsprache** – lesen und in ein gleichwertiges Programm einer anderen Sprache – der **Zielsprache** – übersetzen kann [ALSU08]“

definieren (siehe Abb. 2.1).

Abbildung 2.1: Ein Compiler

Oft wird als Zielsprache **Assemblersprache**, **Bytecode** oder **Maschinensprache** verwendet. Das Zielprogramm kann anschließend aufgerufen werden, um Eingaben entgegen zu nehmen und daraus Ausgaben zu erstellen, wie in Abb. 2.2 dargestellt ist.

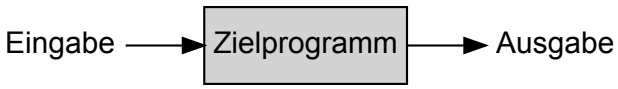

Abbildung 2.2: Ausführen des Zielprogramms

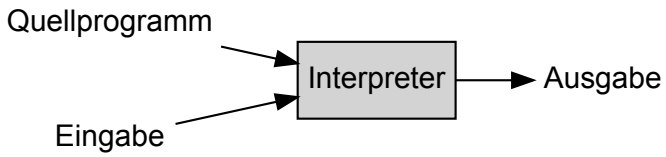

Abbildung 2.3: Ein Interpreter

Neben dem Compiler gibt es noch andere Formen von Sprachprozessoren, z.B. den **Interpreter**. Dieser führt die Operationen des Quellprogramms direkt auf den Eingaben des Benutzers aus und erstellt somit eine Ausgabe (siehe Abb. 2.3).

Ein Vorteil des Compilers ist es, dass das in Maschinensprache erstellte Zielprogramm die Eingaben viel schneller verarbeiten kann, als der Interpreter. Im Gegensatz dazu ist der Interpreter – was die Fehlerdiagnose während der Programmausführung anbelangt – dem Compiler gegenüber meist im Vorteil.

Nachdem in diesem Abschnitt die Funktionsweise eines Compilers nur als ein Ganzes betrachtet wurde, wird im nächsten Abschnitt genauer auf dessen Struktur eingegangen.

2.1.2 Struktur eines Compilers

Die Übersetzung von Quellsprachen in Zielsprachen erfolgt in zwei Hauptschritten, die wiederum in kleinere Teilschritte aufgeteilt sind:

1. **Analyse** und
2. **Synthese**.

Analyse: Während der Analyse des Programms, wird dieses in seine Bestandteile zerlegt und erhält eine grammatikalische Struktur. Diese Struktur wird für die Erstellung einer Zwischendarstellung des Programmcodes genutzt. Werden dabei syntaktische oder semantische Fehler im Quellprogramm erkannt, müssen diese dem Benutzer – in Form von Nachrichten – kenntlich gemacht werden, damit der Benutzer darauf reagieren kann.

Die Symboltabelle ist eine Datenstruktur, in der Informationen über den Programmcode während der Analyse gesammelt und gespeichert werden. Diese Tabelle wird zusammen mit der Zwischendarstellung an die Synthese weitergegeben.

Synthese: Die Synthese hat die Aufgabe aus der Zwischendarstellung und der Symboltabelle das Zielprogramm zu erstellen.

Oft wird der Teil des Compilers, der sich mit der Analyse beschäftigt, als **Front-End** bezeichnet, und der andere Teil, der sich mit der Synthese befasst, als **Back-End**. In Abb. 2.4 kann man gut erkennen, dass ein Compiler während des Kompilierungsvorgangs mehrere Phasen durchläuft und für welche Phasen das

```
1  position = initial + rate * 60
```

Listing 2.1: Quellprogramm

```
1  <id, 1> <=> <id, 2> <+> <id, 3> <*> <60>
```

Listing 2.2: Tokenstream des Quellprogramms

Front- bzw. Back-End zuständig sind. Auf die Informationen der Symboltabelle greifen sowohl das Front- als auch das Back-End zu.

Im Folgenden wird anhand eines kleinen Beispiels[1] näher auf die einzelnen Phasen eingegangen. Ausgangspunkt ist das Quellprogramm, das aus der Anweisung besteht, die in Listing 2.1 zu sehen ist.

In der ersten Phase, der sogenannten **lexikalischen Analyse**, liest der lexikalische Analysator die einzelnen Buchstaben der Anweisung und gruppiert diese in Lexeme. Ein Lexem wird durch einen Token der Form

$$< Tokenname, Attributwert >$$

repräsentiert. Dabei ist der Tokenname ein abstraktes Symbol und der Attributwert ist eine Referenz auf den Eintrag für dieses Token in der Symboltabelle. Die Anweisung kann in Lexeme gruppiert und auf die entsprechenden Token abgebildet werden, was in Tabelle 2.1 abgebildet ist. Dabei werden Leerzeichen vom lexikalischen Analysator verworfen.

Lexem	**Token**
position	<**id**, 1>
=	<**=**>
initial	<**id**, 2>
+	<**+**>
rate	<**id**, 3>
*	<*****>
60	<**60**>

Tabelle 2.1: Symboltabelle: Zuordnung von Lexemen zu Token

Die aus dem Quellprogramm erstellten Token werden dann als **Tokenstream**, wie in Listing 2.2 dargestellt, an die nächste Phase, die **Syntaxanalyse**, übergeben.

Das Hauptziel dieser Phase ist die Konvertierung des Quellprogramms in eine strukturierte Zwischendarstellung. Eine typische Darstellung hierfür ist ein **Syntaxbaum**, bei dem jeder innere Knoten für eine Operation und seine Kindknoten für die Argumente dieser Operation stehen [ALSU08].

[1] Übernommen aus [ALSU08]

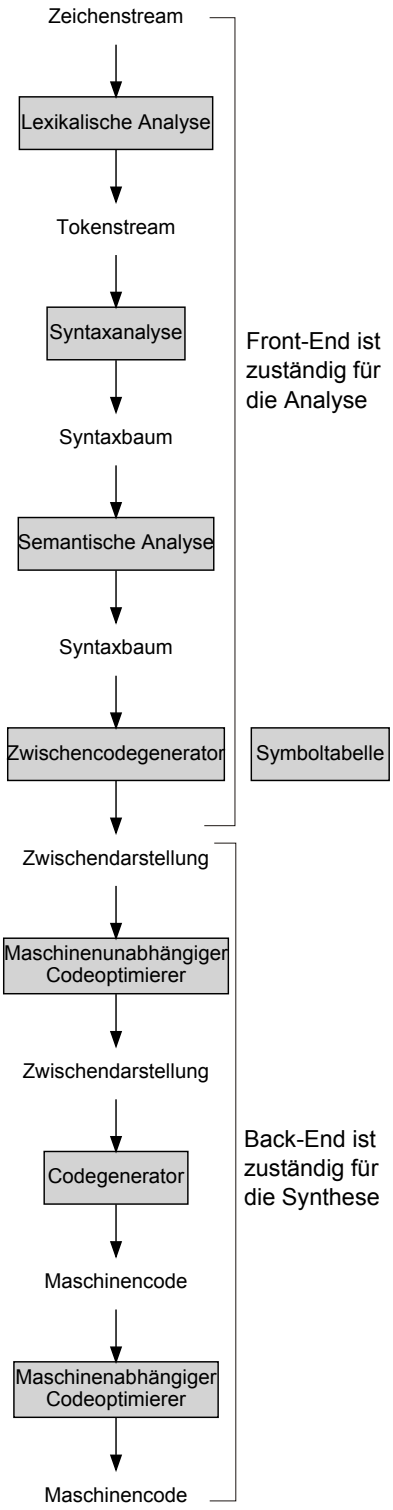

Abbildung 2.4: Phasen eines Compilers

In Abb. 2.5 ist der Tokenstream als Syntaxbaum dargestellt. Die nachfolgenden Phasen (vgl. Abb. 2.4) nutzen diese strukturierte Form, um das Quellprogramm weiter zu analysieren und daraus das Zielprogramm zu generieren.

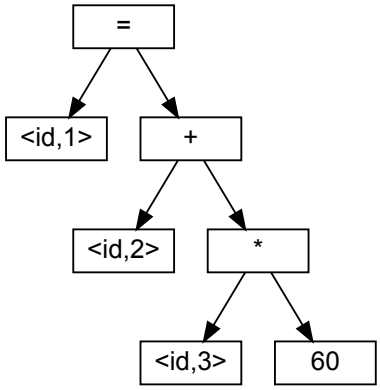

Abbildung 2.5: Tokenstream als Syntaxbaum repräsentiert

Die nächste Phase ist die **semantische Analyse**. Der semantische Analysator hat die Aufgabe, das Quellprogramm auf seine Korrektheit zu überprüfen. Dabei greift er auf Informationen in der Symboltabelle zurück und fügt zusätzliche Informationen hinzu. Dieser Vorgang wird auch als **Typüberprüfung** bezeichnet. Haben beispielsweise zwei Operanden einer Operation nicht die passenden Typen, muss eine Fehlermeldung ausgegeben werden.

Der **Zwischencodegenerator** bildet die letzte Phase des Front-Ends. Nach der syntaktischen und semantischen Analyse des Quellprogramms wird der Syntaxbaum in eine maschinennahe Darstellung (z.B. Drei-Adress-Code) umgewandelt. Diese Darstellung hat den Vorteil, dass sie sich

- einfach erstellen und
- leicht in die Zielmaschine übersetzen lässt.

Zwischen dem Front-End und Back-End eines Compilers kann optional eine **maschinenunabhängige Optimierungsphase** stattfinden. Dabei wird die bereits maschinennahe Zwischendarstellung so transformiert, dass das Back-End daraus ein besseres und schnelleres Zielprogramm erstellen kann.

Besitzt ein Compiler keine **maschinenabhängige Optimierungsphase**, die ebenfalls optional ist, bildet der **Codegenerator** den letzten Schritt des Back-Ends. Dieser erhält als Eingabe die Zwischendarstellung des Quellprogramms und bildet diese auf das Zielprogramm ab.

Der nachfolgende Abschnitt soll einen kleinen Einblick in die Definition neuer Programmiersprachen geben:

- was es bedeutet, eine neue Programmiersprache zu entwickeln und
- was alles dafür benötigt wird.

2.1.3 Sprachdefinition

Eine Programmiersprache ist ein **formales System**, das wie folgt definiert werden kann:

> „Ein formales System ist ein mathematisch präzises System, das ein Phänomen von Interesse modelliert [Pie02]."

Formale Systeme besitzen **Regeln** oder **Axiome**, die es erlauben mit dem System zu arbeiten bzw. Berechnungen durchzuführen. Sie sollten möglichst einfach sein, um Untersuchungen und Beweisführungen zu erleichtern und eine möglichst große Aussagekraft besitzen, um die relevanten Eigenschaften des Phänomens exakt widerzuspiegeln.

Für die Definition einer Programmiersprache werden drei Bestandteile benötigt (siehe Abb. 2.6):

1. Formale Definition der **Syntax**
2. Definition des **Typsystems**
3. Formale Definition der **Semantik**

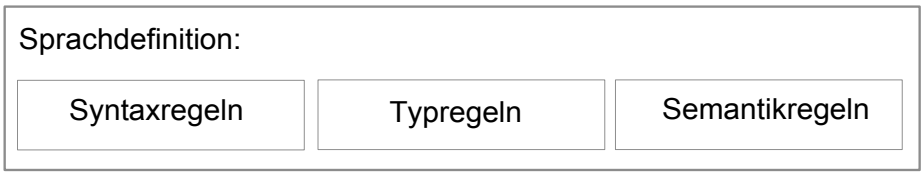

Abbildung 2.6: Bestandteile einer Sprachdefinition

Syntax: Unter der Syntax einer Programmiersprache versteht man ein System von Regeln, nach denen erlaubte Konstruktionen bzw. wohlgeformte Ausdrücke, wie z.B. die Verschachtelung von Klammern oder Steuerstrukturen, aus einem grundlegenden Zeichenvorrat gebildet werden — wobei von der inhaltlichen Bedeutung der Zeichen abgesehen wird. Meist wird die syntaktische Struktur einer Programmiersprache durch eine **kontextfreie Grammatik** beschrieben.

Das folgende Beispiel beschreibt die Grammatik einer einfachen Sprache für arithmetische Ausdrücke (vgl. Abschnitt 5.1). Die Sprache besteht aus Anweisungen der Form (nur ein Auszug der gesamten Grammatik):

- Zuweisung: $VAR := EXP$;
- Variablendeklaration: $var\ ID : TYPE$;
- Funktion: $eval(\ EXP\ )$;

Dabei ist EXP, wie in List. 2.3 dargestellt, definiert.

```
1   EXP :                Ausdrücke
2     INT                  Konstante
3     STRING               Konstante
4     -EXP                 Unäres Minus
5     (EXP)                Klammerung
6     EXP + EXP            Operation Addition
7     EXP - EXP            Operation Subtraktion
8     EXP * EXP            Operation Multiplikation
9     EXP / EXP            Operation Division
10    EXP % EXP            Operation Modulo
```

Listing 2.3: Definition für Ausdrücke

Typsystem: Einen weiteren Teil der Programmiersprache bildet das **Typsystem**, das definiert wird als

> „eine **syntaktische** Methode zur **automatischen** Überprüfung von Programmen mit dem Ziel, **fehlerhaftes** Programmverhalten auszuschließen [Pie02]."

Zu den Aufgaben eines Typsystems gehören u.a.:

- das frühe Auffinden von Fehlern: ein Typsystem kann Fehler aufgrund von Typinformationen erkennen. So können Operationen mit inkompatiblen Typen oder nicht deklarierten Variablen bereits vor dem Programmstart, also bei der Analyse des Programms durch den Compiler, angemahnt werden;
- die Verbesserung der Code-Lesbarkeit und der Effizienz: durch statische Überprüfungen können viele dynamische Checks, die sehr zeitaufwändig sein können, eliminiert werden;
- Sicherheitsgarantien: eine sichere Sprache schützt ihre Abstraktionen, z.B. kann der Gültigkeitsbereich von Variablen nicht verletzt werden oder der Zugriff auf Instanzvariablen wird durch Zugriffsmodifikatoren gesteuert und kann somit nicht umgangen werden;
- die Erhöhung des Abstraktionsniveaus.

Semantik: Die Semantik einer Programmiersprache lässt sich wie folgt beschreiben:

> „Die **operationale** Semantik spezifiziert das Verhalten einer Programmiersprache mittels der Definition einer **abstrakten Maschine**. Abstrakt bedeutet, dass die Maschine die Terme der Sprache als Maschinencode benutzt [Pie02]."

Neben der hier verwendeten operationalen Semantik gibt es auch noch weitere semantische Stile. Dazu gehören die **denotationelle** und die **axiomatische** Semantik, auf die hier aber nicht näher eingegangen wird.

Für einfache Sprachen wird ein Zustand der Maschine durch einen Term repräsentiert und das Verhalten der Maschine wird durch eine **Transitionsfunktion** definiert [Ape10]. Das heißt, entweder hält die Maschine, was

```
1  VALUE :              Werte
2    INT                Integer Konstante
3    STRING             String Konstante
```

Listing 2.4: Semantik von arithmetischen Ausdrücken: Werte

einem Laufzeitfehler entspricht, oder es gibt einen Übergang in den nächsten Zustand. Für komplexere Sprachen muss z.B. auch der Programmspeicher berücksichtigt werden.

In der Beispielsprache für arithmetische Ausdrücke ist der Zustand der Maschine das Programm selbst und eine Berechnung durchführen bedeutet, das vereinfachende Umschreiben des Programms. Hierfür müssen

- **Endzustände**, die als **Werte** bezeichnet werden (siehe List. 2.4) und
- **Evaluierungsfunktionen/-relationen** (siehe Abb. 2.7, Beispiel für Addition)

definiert werden.

Die Regeln *E-PlusID* und *E-PlusSTRING* werden auch als **Berechnungsregeln** bezeichnet und die Regel *E-Plus* als **Kongruenzregel**. Das Zusammenspiel dieser Regeln definiert eine **Evaluierungsstrategie**. Mit Hilfe von Evaluierungsstrategien wird überprüft, ob sich ein Programm in eine **Normalform**, die zugleich auch ein Wert ist, umformen lässt. Ist dies nicht möglich, so ist das Programm festgefahren – was einem Laufzeitfehler entspricht.

$$\mathrm{ID} + \mathrm{EXP} \longrightarrow \mathrm{EXP} \quad \text{(E-PlusID)}$$

$$\mathrm{STRING} + \mathrm{EXP} \longrightarrow \mathrm{EXP} \quad \text{(E-PlusSTRING)}$$

$$\frac{\mathrm{EXP}_1 \longrightarrow \mathrm{EXP}_1'}{\mathrm{EXP}_1 + \mathrm{EXP}_2 \longrightarrow \mathrm{EXP}_1' + \mathrm{EXP}_2} \quad \text{(E-Plus)}$$

Abbildung 2.7: Evaluierungsfunktion für Addition

2.2 Feature-Oriented Programming

Altbewährte Programmierparadigmen, wie z.B. die Objekt-orientierte Programmierung, weisen noch einige Schwächen auf, die z.B. in [Ape08] genauer beschrieben werden. Um diese Probleme lösen zu können, werden modernere Programmierparadigmen genutzt. Dazu gehört u.a. das Feature-Oriented Programming (FOP) Paradigma, das im folgenden Kapitel näher betrachtet wird.

2.2.1 Produktlinie

Vom Software Engineering Institute (SEI) der *Carnegie Mellon University* in Pittsburgh wird der Begriff der **Produktline** wie folgt definiert:

> „A Software Product Line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.“[SEI12]

Der in dieser Definition neu eingeführte Begriff des **Features**, spielt in der Entwicklung von Produktlinien eine zentrale Rolle und wird im Abschnitt 2.2.2 genauer beschrieben.

Die Grundidee, die hinter Produktlinien steckt, ist die Entwicklung von maßgeschneiderter Software, die genau die benötigte Funktionalität für verschiedene Anwendungsfälle, in Form von **Varianten**, enthält [Ape08]. Neue Varianten sollen schnell entwickelt und leicht hinzugefügt werden können, gleichzeitig soll eine bewährte Funktionalität wiederverwendbar sein und die Software soll auf spezielle Kundenwünsche abgestimmt sein sowie sich an die verfügbaren Ressourcen anpassen [Ape08].

Ein sehr anschauliches Beispiel für Produktlinien sind Notebooks. Hier hat der Kunde die Möglichkeit, sich ein Notebook genau nach seinen Wünschen zusammen zu stellen. Ausgehend vom Prozessor, dem Betriebssystem und der Farbe des Notebooks über die Größe des Arbeitsspeichers und der Festplatte bis hin zu zusätzlicher Hardware, wie z.B. einem Blu-Ray Laufwerk, kann sich der Kunde für die einzelnen Komponenten seines Notebooks entscheiden (vgl. Abb. 2.8). Diese individuelle Auswahl von Bauteilen führt zu einer sehr großen Variantenvielfalt.

Einer der größten Vorteile von Produktlinien ist die hohe Wiederverwendbarkeit der einzelnen Funktionalitäten innerhalb der Produktfamilie. So ist es beispielsweise möglich, die Implementation einer bestimmten Funktionalität in der Regel für alle Produkte wieder zu verwenden. Durch die Wiederverwendbarkeit solcher Komponenten, wird eine Senkung der Entwicklungskosten angestrebt, was eines der Hauptziele bei der Entwicklung von SPLs ist.

2.2.2 Feature

Features werden dazu genutzt, zwischen verschiedenen Varianten eines Programms oder einer Software zu unterscheiden. Eine allgemeine Definition findet sich in [ALMK08]:

> „A feature is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder‘s requirement, to implement a design decision, and to offer a configuration option.“

Ein Feature [Ape08]:

- ist eine semantisch zusammenhängende Einheit, die einer Anforderung an ein Programm (Produkt) entspricht.

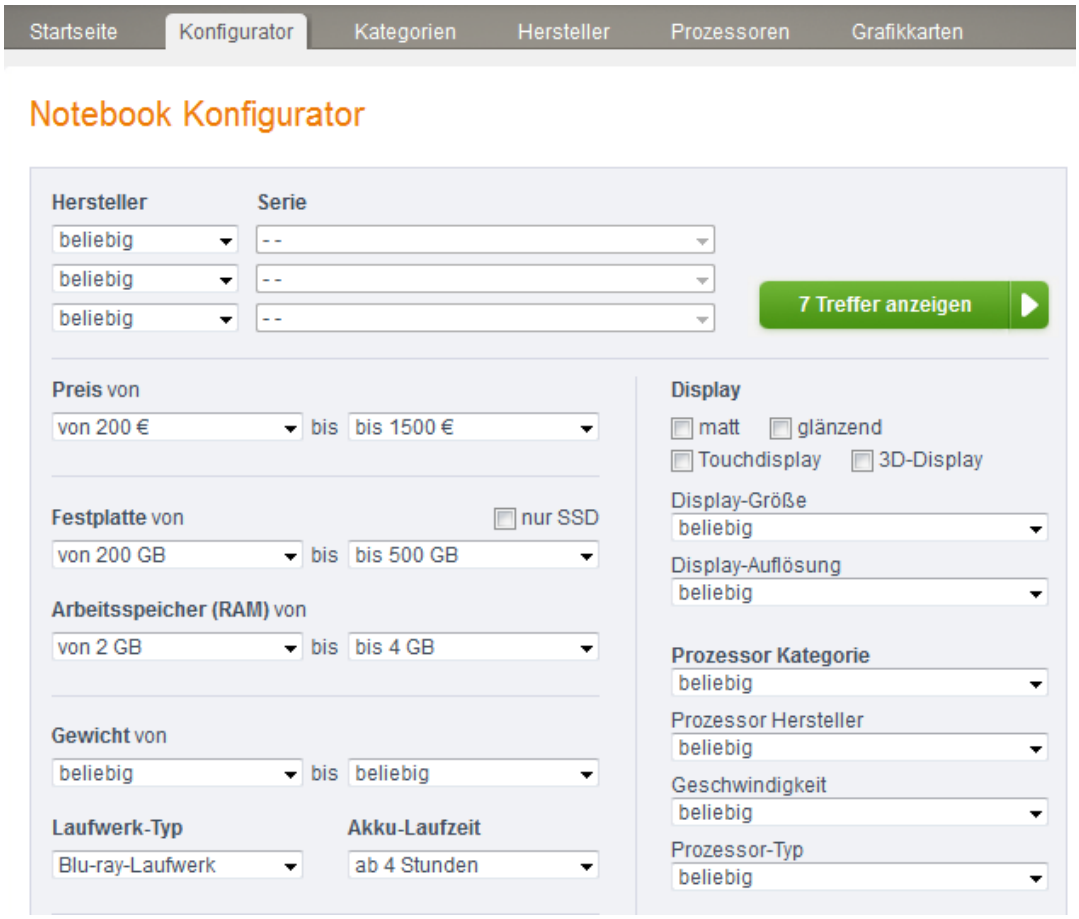

Abbildung 2.8: Produktlinie – Notebook
[Quelle: http://www.notebookinfo.de/notebook-konfigurator/]

- ist eine Abstraktion einer Funktionalität.
- dient zur Spezifikation von Programmen (Produkten).
- repräsentiert Gemeinsamkeiten oder Unterschiede eines Programms (Produkts).
- bietet eine Konfigurationsmöglichkeit (vgl. Beispiel zu Produktlinien in Abb. 2.8).

Meistens sind die Programme einer Produktlinie auf einen bestimmten Markt, auch **Domäne** genannt, zugeschnitten, damit sie nicht zu unterschiedlich werden [Ape08]. Die **Feature-Modellierung** dient zur Darstellung der Features einer solchen Domäne [Ape08]. Durch ein **Feature-Modell** werden die elementaren Abstraktionen einer Domäne und deren Beziehungen sowie die Menge der Programme einer Produktlinie beschrieben [Ape08]. Feature-Modelle sind hierarchisch angeordnete Mengen von Features. Die Beziehungen zwischen **Eltern-Features** und **Kind-Features** lassen sich wie folgt definieren [Bat05]:

- *and*: alle Kind-Features müssen ausgewählt werden
- *alternative*: genau eines der Kind-Features kann ausgewählt werden
- *or*: mindestens eines der Kind-Features muss ausgewählt werden
- *mandatory*: Kind-Feature muss ausgewählt werden
- *optional*: Kind-Feature kann ausgewählt werden

Mit Hilfe eines **Feature-Diagramms** werden Features und deren Beziehungen graphisch dargestellt [Bat05]. Hierbei handelt es sich um einen Baum, der für eine hierarchische Darstellung besonders gut geeignet ist:

- Die inneren Knoten entsprechen dabei den zusammengesetzten Features (Eltern-Features).
- Die Blätter entsprechen primitiven, d.h. atomaren Features [Bat05].
- Die Kanten des Baumes tragen die Information über die Art der o.g. **Eltern-Kind-Beziehungen**, wie in Abb. 2.9 zu sehen ist.

Abb. 2.10 zeigt ein Beispiel eines Feature-Diagramms.

Features sind, wie oben beschrieben, semantisch zusammenhängende Einheiten, deren Implementierung jedoch im Quelltext verstreut vorliegen kann. Grund hierfür ist eine mangelnde **Feature-Kohäsion**, die in [Ape08] definiert ist als

> „Eigenschaft eines Programms alle Implementierungseinheiten eines Features an einer Stelle im Code zu lokalisieren“

und die das **Feature Traceability Problem** zur Folge hat. In [Ape08] wird es definiert als

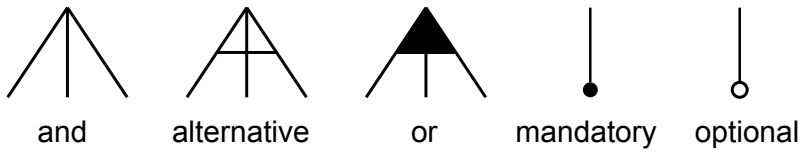

Abbildung 2.9: Kantenkennzeichnungen
[Quelle: [Bat05]]

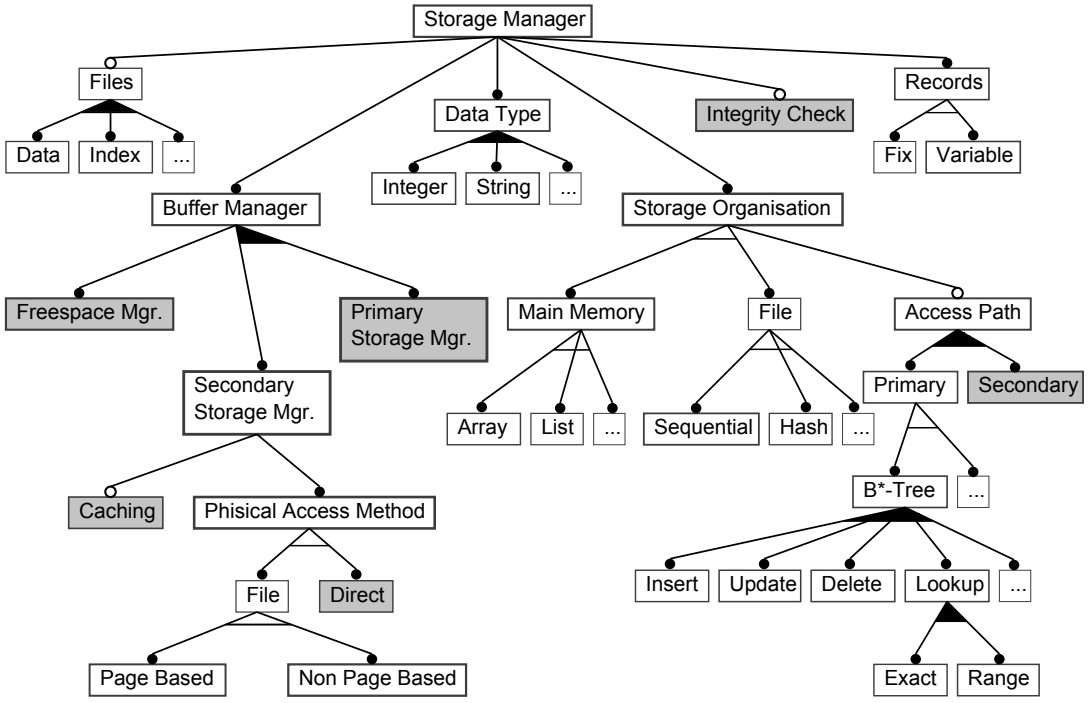

Abbildung 2.10: Feature-Diagramm eines Speichermanagers
[Quelle: [Ape08]]

> „Problem die Features einer Domäne bzw. eines konkreten Programms im Code wiederzufinden."

Die **Feature-orientierte Programmierung** ist in der Lage dieses Problem zu lösen, indem jedes Feature durch ein **Feature-Modul** implementiert wird [Ape08]. Dadurch wird eine Trennung und Modularisierung von Features erreicht und die Komposition von Features vereinfacht. Diese Modularität erlaubt eine Programmerstellung mit Hilfe einer **Feature-Selektion** [Ape08]. Dabei wird aus allen vorhandenen Features eine Menge von Features ausgewählt, die dann das Programm bildet.

Häufig ergibt sich das Problem, dass die Anzahl der möglichen Features meist sehr groß ist – mehr als 100. Ein möglicher Lösungsansatz hierfür ist eine **partielle Selektion** [Ape08]. Die Grundidee ist folgende:

Ein Anwender wählt nur die Features aus, die für ihn am wichtigsten sind und daraufhin werden *passende* Features automatisch ausgewählt. Welches passende Features sind, wird durch Einschränkungen und Regeln der Domäne oder durch Expertenwissen festgelegt [Ape08]. Durch diese automatische Selektion wird eine sehr gute Optimierung erreicht.

Es gibt drei übliche Ansätze eine SPL (vgl. Abschnitt 2.2.1) zu erstellen bzw. einzuführen [Ape08]:

- **Proaktives Vorgehensmodell**
- **Reaktives Vorgehensmodell**
- **Extraktives Vorgehensmodell**

Der nachfolgende Abschnitt beschreibt diese Vorgehensmodelle genauer.

2.2.3 Vorgehensmodelle

Proaktiv: Bei diesem Vorgehensmodell wird eine SPL **neu** entworfen und implementiert. Dafür muss eine komplette **Domänenanalyse** zu Beginn durchgeführt werden und es müssen zunächst alle möglichen Features identifiziert und geplant werden – ähnlich dem **Wasserfallmodell** [Ape08].

Mit diesem Vorgehen sind hohe Kosten und ein hohes Risiko verbunden, da die Analyse der Domäne und der Features sehr zeitaufwändig ist. Das Vorgehensmodell ist besonders dann gut geeignet, wenn die Anforderungen wohldefiniert sind [Ape08]. Bei unklaren Anforderungen hingegen ist dieses Vorgehensmodell nicht zu empfehlen.

Reaktiv: Im Gegensatz zum proaktiven werden beim reaktiven Vorgehensmodell zunächst nur einige wenige Variationen implementiert und später inkrementell weitere hinzugefügt [Ape08]. Dadurch ergibt sich eine kleinere Projektgröße und es fallen geringere Anfangskosten an, da weniger Ressourcen benötigt und schnellere Ergebnisse erzielt werden.

Dieses Vorgehen ist vor allem dann geeignet, wenn die benötigten Varianten nicht komplett im Voraus bekannt sind, und für unvorhergesehene Änderungen. Ein Nachteil ist jedoch, dass später eventuell Umstrukturierungen nötig sind [Ape08].

Extraktiv: Dieses dritte und letzte Vorgehensmodell verfolgt einen anderen Ansatz, wie die ersten beiden. Es nutzt eine oder mehrere bestehende Anwendungen als Basis, extrahiert daraus Features und erzeugt so verschiedene Varianten [Ape08].

Dieses Vorgehen ist gut geeignet für den schnellen Wechsel von traditionellen Anwendungen zu SPLs. Dabei werden relativ wenig Ressourcen benötigt und das Risiko ist geringer als bei den anderen zwei Vorgehensmodellen.

2.2.4 FeatureHouse

Das Framework **FeatureHouse** bietet die Möglichkeit der Komposition von Features, basierend auf einem sprachenunabhängigen Modell für Softwareartefakte. Die hier vorgestellten Inhalte sind aus [AKL09] übernommen und sollen einen Überblick über die Funktionsweise des Frameworks verschaffen.

Softwarekomposition bedeutet die Konstruktion von Softwaresystemen aus einer Menge von Softwareartefakten. Ein **Artefakt** kann jegliche Art von Information sein, die Teil einer Sprache ist. Dazu gehören Codestücke, wie z.B. Pakete, Klassen und Methoden, oder unterstützende Dokumente, wie z.B. Modelle und Dokumentationen. Ein sehr beliebter Ansatz für die Softwarekomposition ist die **Superimposition** [AL08].

Bei der Superimposition werden Softwareartefakte komponiert, indem korrespondierende Teilstrukturen der Artefakte miteinander verschmolzen werden. Superimposition wurde bereits vielfach erfolgreich eingesetzt:

- Komposition von Klassenhierarchien in Multi-Team Softwareentwicklung [OH92]
- Erweiterung von verteilten Programmen [BF88]
- Feature-Orientierte Programmierung [BSR04, Pre97]
- Multi-Dimensionale Separierung von Belangen [TOHS99]
- Aspekt-Orientierte Programmierung [MH03, MO]

Obwohl diese Anwendungen sehr verschieden sind, nutzen sie alle die Superimposition von hierarchisch organisierten Programmkonstrukten, basierend auf nominellen und strukturellen Gemeinsamkeiten. Es gibt bereits eine Vielzahl an Tools, die eine Superimposition von Softwareartefakten unterstützen, aber diese sind meist nur für eine bestimmte Sprache definiert.

Ein genereller Ansatz für die Komposition, basierend auf Superimposition, von Softwareartefakten, die in verschiedenen Sprachen geschrieben sind, wird durch

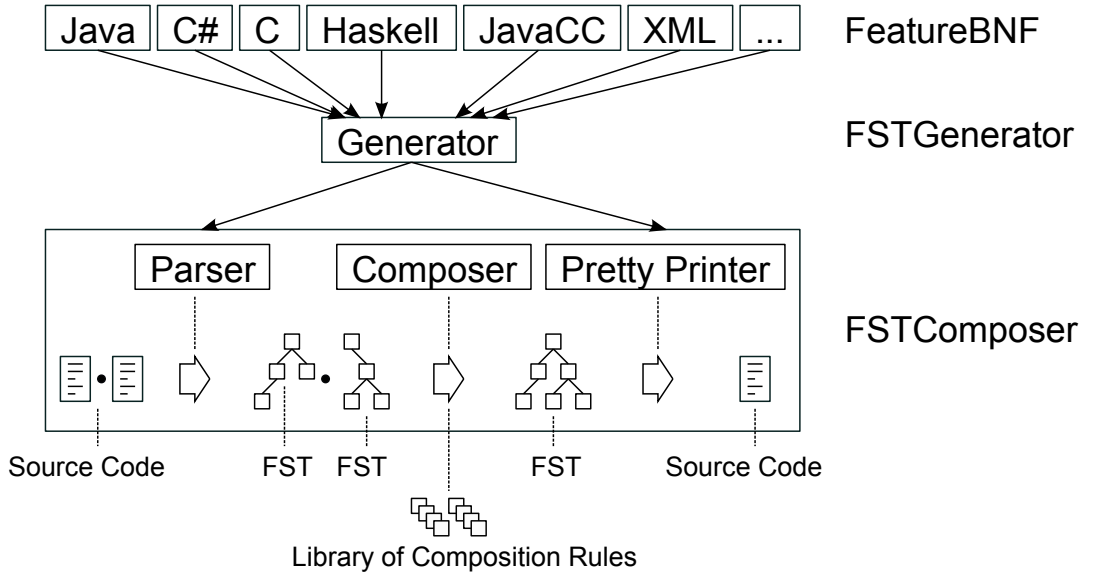

Abbildung 2.11: FeatureHouse Architektur
[Quelle: [AKL09]]

FeatureHouse beschrieben. Technisch gesehen besteht das Framework aus den folgenden drei Teilen:

1. Sprachenunabhängiges Modell für Softwareartefakte

2. Superimposition als ein sprachenunabhängiges Kompositionsparadigma

3. Spezifikation für Artefakte einer Sprache basierend auf einer **Attributgrammatik**

In Abb. 2.11 ist die Architektur von FeatureHouse dargestellt. Der **FSTComposer** basiert auf einem allgemeinen Modell für die Struktur von Softwareartefakten, dem sogenannten Feature Structure Tree (FST)-Modell. FSTs können jegliche Art von Artefakte, die eine hierarchische Struktur besitzen, repräsentieren und abstrahieren dabei von sprachenspezifischen Details.

Jeder Knoten in einem FST hat einen Namen und einen Typ, der dem strukturellen Element des Softwareartefakts entspricht. Die inneren Knoten (**Nichtterminale**) spezifizieren Module und die Blätter (**Terminale**) den dazugehörigen Inhalt. Welche strukturellen Elemente als innere Knoten oder als Blätter repräsentiert werden, hängt von der Granularität ab, mit der die Softwareartefakte komponiert werden sollen.

Alle strukturellen Elemente, die nicht im FST dargestellt werden, sind in Form von Text als Inhalt der Blätter gespeichert. Abb. 2.12 zeigt, wie ein Java Softwareartefakt als FST dargestellt werden kann.

Die Komposition von Softwareartefakten erfolgt durch die Superimposition der entsprechenden FSTs, gekennzeichnet durch den • Operator. Zwei FSTs werden komponiert, indem ihre Knoten, beginnend bei der Wurzel, absteigend vereint

```
1  package com.sleepycat;
2  public class Database {
3    private DbState state;
4    private List triggerList;
5    protected void notifyTriggers(Locker locker, DatabaseEntry priKey,
6      DatabaseEntry oldData, DatabaseEntry newData) throws DatabaseException {
7        for (int i=0; i<triggerList.size(); i+=1) {
8          DatabaseTrigger trigger = (DatabaseTrigger) triggerList.get(i);
9          trigger.databaseUpdated( this , locker, priKey, oldData, newData);
10       }
11   } // over 650 further lines of code...
12 }
```

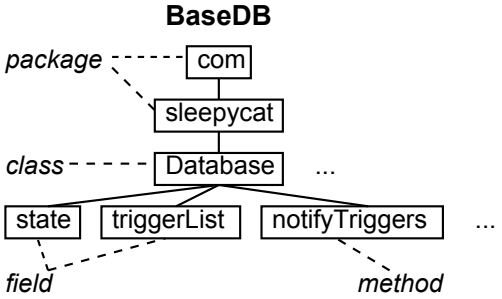

Abbildung 2.12: Java Code und FST für das Softwareartefakt BaseDB [Quelle: [AKL09]]

werden (siehe Abb. 2.13). Bei der Komposition zweier Blätter muss berücksichtigt werden, dass der Inhalt nicht als Unterbaum repräsentiert wird, sondern als Text.

Zum Beispiel werden Methodenrümpfe anders komponiert, als Felder oder Extensible Markup Language (XML)-Elemente. Hierfür müssen bestimmte Regeln berücksichtigt werden, die abhängig vom Typ des Softwareartefakts sind und von der Sprache, in der die Softwareartefakte implementiert sind. Unter anderem gehören **Ersetzung**, **Konkatenation**, **Spezialisierung** oder **Überschreibung** zu diesen Regeln.

Für die Integration neuer Sprachen in FeatureHouse wird der **FSTGenerator** (vgl. Abb. 2.11) verwendet. Hierfür wird die Grammatik, der zu integrierenden Sprache, im **FeatureBNF**-Format [Fea] benötigt. Ausgehend von dieser Grammatik können dann mit Hilfe des FSTGenerators andere Tools, wie **LL(k)-Parser** [Kun08], **Adapter** und **Pretty Printer**, für die neue Sprache automatisch generiert werden.

Bisher wurden u.a. die Sprachen **Java**, **C#**, **C**, **Haskell**, **JavaCC** und **XML** in FeatureHouse integriert. Im Rahmen dieser Masterarbeit werden SDF und Stratego durch die Erstellung zweier Grammatiken für die beiden Sprachen (siehe Abschnitt 4.2) in FeatureHouse eingebunden.

Nach dem Generierungsschritt kann die Komposition von Softwareartefakten der neuen Sprache erfolgen (siehe auch Abb. 2.11):

1. Der generierte Parser erhält als Eingabe Softwareartefakte, die als Quelltext vorliegen, und erstellt pro Artefakt einen FST. Um festzulegen, wie Arte-

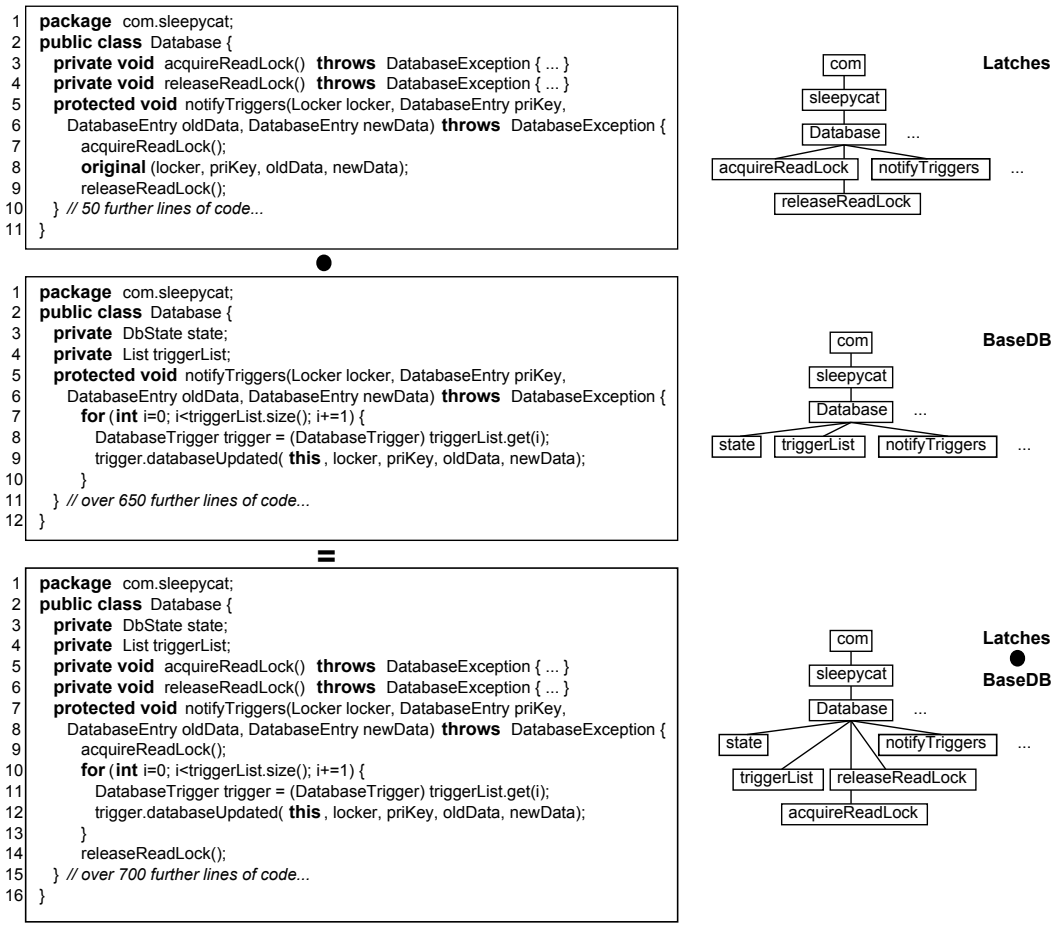

Abbildung 2.13: Komposition von Softwareartefakten
[Quelle: [AKL09]]

```
1  ClassDecl : "class" Type "extends" ExtType "{"
2    (VarDecl)* (ClassConstr)* (MethodDecl)*
3  "}";
4  VarDecl : Type <IDENTIFIER> ";";
5  MethodDecl :
6      Type <IDENTIFIER> "(" (FormalParamList)? ")" "{"
7        "return" Expression ";"
8      "}";
```

Abbildung 2.14: Auszug einer vereinfachten Java Grammatik
[Quelle: [AKL09]]

```
1  @FSTNonTerminal(name="{Type}")
2  ClassDecl : "class" Type "extends" ExtType "{"
3    (VarDecl)* (ClassConstr)* (MethodDecl)*
4  "}";
```

Abbildung 2.15: Auszug einer vereinfachten Java Grammatik mit Annotationen
[Quelle: [AKL09]]

fakte als FST repräsentiert werden sollen, muss die Grammatik der Sprache mit Annotationen in Form von **Attributen** versehen werden.

2. Der FSTComposer führt die Komposition durch. Dabei greift er auf eine Bibliothek von **Kompositionsregeln** zu, die für die Komposition des Inhalts von terminalen Knoten benötigt werden.

3. Der generierte Pretty Printer speichert dann das komponierte Artefakt wieder als Quelltext ab.

Attribute : Mit Hilfe von Attributen kann festgelegt werden, wie Softwareartefakte als FST repräsentiert werden sollen. Die Nutzung dieser Attribute ist in Abb. 2.15 zu sehen. Abb. 2.14 zeigt einen Auszug aus einer Grammatik in FeatureBNF, die Klassen- und Methodendeklarationen beschreibt. Dabei definiert z.B. die Regel **ClassDecl**, die Struktur von Klassen. Klassen können Felder, Konstruktoren und Methoden beinhalten, die durch die Regeln **VarDecl**, **ClassConstr** und **MethodDecl** definiert sind.

Ohne die Angabe von Attributen generiert der FSTGenerator standardmäßig für jede Produktionsregel einen entsprechenden terminalen Knoten im FST. Es werden aber nur die Top-Level Terminale im generierten FST angezeigt, d.h. in diesem Fall wird pro Klasse ein terminaler Knoten im FST dargestellt. Durch die Verwendung von Attributen kann eine feinere Granularität des FSTs erreicht werden.

Hierzu werden bestimmte Produktionsregeln mit Annotationen, wie **@FSTNonTerminal** versehen (siehe Abb. 2.15). Dieses Attribut zeigt an,

```
1  @FSTTerminal(name="{<IDENTIFIER>}({FormalParamList})",
2    composer="JavaMethodOverriding")
3  MethodDecl :
4      Type <IDENTIFIER> "(" (FormalParamList)? ")" "{"
5        "return" Expression ";"
6      "}";
```

Abbildung 2.16: Annotation für Kompositionsregeln
[Quelle: [AKL09]]

dass Klassen nichtterminale Knoten sind und weitere Elemente beinhalten können. Mit dem Attributparameter **name** wird dem Knoten, der die Klasse im FST repräsentiert, der Name der Klasse zugewiesen.

Mit Hilfe des Attributparameters **compose** wird die Kompositionsregel für terminale Knoten spezifiziert. Abb. 2.16 zeigt z.B. die Verwendung der Kompositionsregel **JavaMethodOverriding**. Dabei werden zwei Methoden, die in verschiedenen FSTs als Terminale repräsentiert werden und den gleichen Namen besitzen, bei der Komposition der FSTs zu einem terminalen Knoten verbunden, indem die Methodenrümpfe miteinander verschmolzen werden.

Anhand der in diesem Grundlagenkapitel vorgestellten Konzepte, sollen in den nachfolgenden Kapiteln der Ansatz des Feature-Oriented Language Engineerings, eine programmiertechnische Umsetzung und zwei Fallstudien zur Untersuchung der Praktikabilität dieses Ansatzes, näher betrachtet werden.

Kapitel 3

Ansatz – Feature-Oriented Language Engineering

Im Rahmen dieser Masterarbeit wird das **extraktive Vorgehensmodell** (vgl. Abschnitt 2.2.3) zur Erstellung einer Produktlinie – es handelt sich um eine Produktlinie für Sprachdefinitionen – genutzt. Ausgangspunkt hierfür ist eine bereits bestehende Sprachdefinition (vgl. Fallstudien in Kapitel 5).

Im extraktiven Vorgehensmodell wird eine **Refaktorisierung** in Features genutzt [Ape08]. Dabei wird Quellcode aus der zugrundeliegenden Sprachdefinition, die aus Syntax-, Typ- und Semantikregeln besteht (siehe 2.1.3), in Features verschoben, ohne das sich dabei das Verhalten der Anwendung ändert. Das heißt, mit aktivierten Features muss sich die Produktlinie wie die originale Anwendung verhalten [Ape08].

Anschließend können mit Hilfe der gefundenen und extrahierten Features unterschiedliche Varianten von Sprachdefinitionen erstellt werden (siehe Fallstudien in Kapitel 5). Dies erfolgt durch die Komposition der ausgewählten Features in FeatureHouse. Der nachfolgende Abschnitt erläutert dieses Vorgehen genauer.

3.1 Überblick

Der Ansatz des Feature-Oriented Language Engineerings lässt sich durch die folgenden Schritte beschreiben (siehe Abb. 3.1):

1. Refaktorisierung der Sprachdefinition in Features:
   Hierfür müssen der Quellcode untersucht und mögliche Features identifiziert werden. Dazu wird der Quellcode zunächst mit Kommentaren versehen, die kennzeichnen, zu welchem Feature der Teil des Quellcodes gehört. Dadurch wird die spätere Abbildung der Features auf Implementierungseinheiten erleichtert.

2. Erstellung eines Feature-Modells anhand der gefundenen Features, z.B. mit Hilfe des **Eclipse**[1] Plug-Ins **FeatureIDE**[2]:
   Die graphische Repräsentation der Features durch ein Feature-Modell hat vor allem den Vorteil, dass sehr leicht zu erkennen ist, ob ein Feature optional oder von einem anderen Feature abhängig ist. Desweiteren können

[1] http://www.eclipse.org
[2] http://www.fose.de/featureide

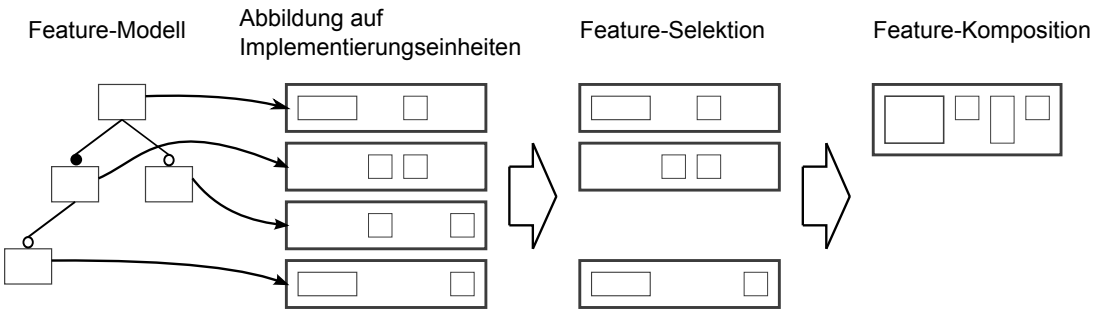

Abbildung 3.1: Überblick der Feature-orientierten Vorgehensweise

zusätzliche Bedingungen – sogenannte **Constraints** – im Feature-Modell beschrieben werden, wie z.B.:

$$FeatureA \wedge FeatureB \Rightarrow FeatureC$$

Wenn **FeatureA** und **FeatureB** ausgewählt sind, dann muss auch **FeatureC** gewählt werden.

3. Abbildung der Features auf entsprechende Implementierungseinheiten:
   Die Sprachdefinition besteht aus mehreren **.sdf-** bzw. **.str-Modulen** (in Abb. 3.2 senkrecht dargestellt). Features (in Abb. 3.2 waagerecht dargestellt) werden oft von mehreren dieser Module implementiert und ein Modul der Sprachdefinition implementiert oft mehr als nur ein Feature [Ape08].

   Die grundlegende Idee ist, die Module der Sprachdefinition prinzipiell beizubehalten, sie jedoch in Feature-Module – auch Kollaborationen genannt – aufzuteilen (siehe Abb. 3.2).

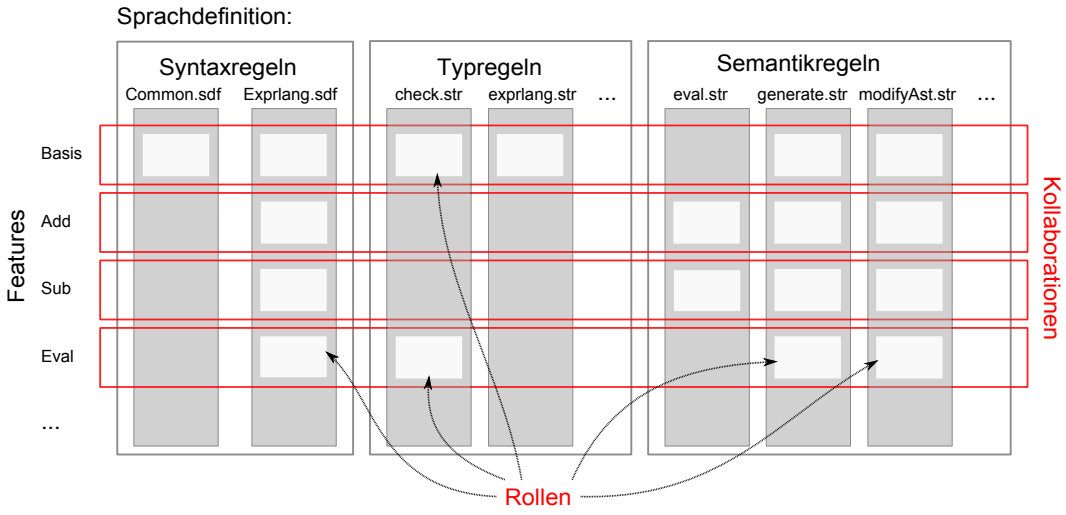

Abbildung 3.2: Sprachdefinition: Zerlegung in Features

Eine **Kollaboration** ist eine Menge von Modulen, die miteinander interagieren, um ein Feature zu implementieren. Dabei spielen verschiedene Mo-

dule verschiedene **Rollen** innerhalb einer Kollaboration und ein Modul spielt verschiedene Rollen in verschiedenen Kollaborationen (siehe Abb. 3.2). Eine Rolle kapselt das Verhalten bzw. die Funktionalität eines Moduls, welches bzw. welche für eine Kollaboration relevant ist [Ape08].

4. Feature-Selektion zur Erstellung neuer Varianten:
   Verschiedene Varianten von Sprachdefinitionen können durch die Auswahl einiger Features aus der Menge aller vorhandenen Features erstellt werden. Dabei müssen gegebenenfalls Abhängigkeiten zwischen den Features berücksichtigt oder, wie oben beschrieben, Bedingungen eingehalten werden.

5. Feature-Komposition mit Hilfe von FeatureHouse (vgl. 2.2.4):
   Die Features, die im vorherigen Schritt ausgewählt wurden, dienen den Kompositions-Algorithmen von FeatureHouse als Eingabe und diese Algorithmen erstellen daraus die komponierte, fertige Sprachdefinition.

Eine der Herausforderungen bei diesem Vorgehen sind **optionale** Features. Die Extraktion eines Features entspricht einer neuen Variante innerhalb der Produktlinie und alle so erstellten Varianten müssen korrekte Anwendungen sein. Das heißt, die Extraktion eines Features muss vollständig sein [Ape08].

Eine weitere Herausforderung ist die **Granularität** bei der Refaktorisierung. Ist die ursprüngliche Anwendung nicht für Features vorbereitet sind Erweiterungen oft sehr feingranular und mit vielen Erweiterungspunkten verbunden, was den Quellcode sehr schnell unlesbar macht [Ape08].

Im folgenden Abschnitt werden einige Aspekte, die für einen solchen Ansatz sprechen, näher vorgestellt.

3.2 Gründe

Es gibt mehrere Gründe, die für die Nutzung eines Feature-orientierten Ansatzes bei der Entwicklung von Sprachdefinitionen sprechen. Dazu gehören u.a. die folgenden:

Entwicklung von Programmiersprachenvarianten: Es können verschiedene Varianten einer Programmiersprache erstellt werden, indem man unterschiedliche Features selektiert und miteinander komponiert. Die so erstellten Varianten können dann in diversen Bereichen eingesetzt werden, um maßgeschneiderte Anwendungen zu entwickeln.

Experimentieren mit Programmiersprachenfeatures: Aus der Menge aller Features, werden einige Features ausgewählt und komponiert. Ausgehend von den so generierten Sprachvarianten können dann Experimente durchgeführt werden, mit der Zielsetzung herauszufinden, welche Variante sich am besten für die Entwicklung einer bestimmten Anwendung eignet. Es können immer wieder neue Kombinationen von Features ausprobiert werden, bis eine optimale Zusammenstellung der Features für eine bestimmte Anwendung erreicht ist.

Downsizing bzw. Verkleinern von Programmiersprachen: Ein Programmierer muss nicht immer alle möglichen Features einer Programmiersprache zur Entwicklung seiner Anwendungen nutzen. Es kann oft sinnvoll sein, die Programmiersprache seinen Wünschen nach anzupassen, indem die nicht benötigten Features aus der Sprachendefinition entfernt werden. Der Programmierer wählt nur die Features aus, die er auch wirklich benötigt, und komponiert diese zu einer neuen Variante, die an seine speziellen Wünsche angepasst ist.

Schrittweise Entwicklung von Programmiersprachen: Ein Beispiel für die schrittweise Entwicklung von Programmiersprachen ist Java. In der Version 1.5 – im Gegensatz zur Version 1.4 – gibt es z.B. neue Sprachfeatures, wie **Generics**. Einer der Hauptgründe für die Einführung von Generics war der Wunsch, die existierenden Containerklassen, wie z. B. **Vector**, **Map** oder **Set**, weiterhin wiederverwendbar zu halten, aber diese um Typsicherheit zu erweitern.

So waren vor der Java-Version 1.5 häufig sogenannte Downcasts (darunter versteht man die explizite Verwendung eines spezielleren Typs für ein Objekt) nötig, wenn man die Objekte aus einem Container auslesen wollte. Durch eine Schrittweise Entwicklung können somit neue Features hinzugefügt werden, die die bestehende Sprachdefinition erweitern und verbessern.

Ein anderes Beispiel ist der Internetbrowser **Firefox** von Mozilla[3]. Mittlerweile gibt es schon die 10. Version und in jeder Version kommen neue Features hinzu oder bereits bestehende Features werden verbessert. Es kann auch sein, dass Features, die sich nicht bewährt haben, wieder entfernt werden. Dies erfolgt vor allem in Form von **Add-ons**.

Komposition von DSLs: Eine domänenspezifische Sprache ist eine formale Sprache, die speziell für eine bestimmte Domäne entworfen und implementiert wird. Beim Entwurf einer DSL wird darauf geachtet, dass die Sprache alle Probleme der Domäne – und nichts was außerhalb der Domäne liegt – darstellen kann. Zu den Vorteilen einer DSL gegenüber der Nutzung einer allgemeinen Programmier- oder Spezifikationssprache zählen:

- Weniger Redundanz
- Bessere Lesbarkeit
- Deklarative Beschreibung eines Sachverhaltes
- Leichte Erlernbarkeit, aufgrund des beschränkten Umfangs

Besteht eine solche DSL aus einzelnen Features, kann sie mit Features einer anderen DSL komponiert werden und man erhält somit eine völlig neue DSL, die für beide Domänen genutzt werden kann.

[3] http://www.mozilla.org/

Kapitel 4

Implementierung

Eine Sprachdefinition beinhaltet eine Vielzahl an Dokumenten, die aus Syntax-, Typ- und Semantikregeln bestehen und aus denen Sprachtools automatisch generiert werden können. Hierfür wird das Eclipse Plug-In **Spoofax**[1] genutzt, das in diesem Kapitel näher betrachtet wird.

4.1 Spoofax Workbench

Spoofax kann zur Entwicklung neuer Sprachen und Programmtransformationen genutzt werden. Dabei wird der **Syntax Definition Formalism (SDF)** für die Beschreibung der Grammatik einer Programmiersprache genutzt, und basierend auf dieser Grammatik werden dem Benutzer grundlegende Editor-Services, wie z.B. Syntax Highlighting und Code Folding automatisch zur Verfügung gestellt.

Stratego hingegen wird zur Spezifizierung von anspruchsvolleren Editor-Services, wie Fehlererkennung bzw. -markierung und Wortvervollständigung verwendet. Dabei bietet die Sprache Stratego einen vereinheitlichten Formalismus für eine präzise Beschreibung von Analyse, Transformation und Codegenerierung an, die es erlaubt, Analyseregeln, wie z.B. dynamische Rewrite Regeln für kontextfreie Analysen und Transformationen, für unterschiedliche Zwecke zu nutzen.

Mit Spoofax kann die Grammatik einer neuen Sprache entwickelt und gleichzeitig können in dieser Programmiersprache Programme geschrieben werden. Es kann sogar ein selbständiges Plug-In aus der erstellten Programmiersprache generiert werden (vgl. Abs. 5.2.3). Spoofax bietet eine Vielzahl an Editor-Services (siehe Abs. 4.1.1 – Editor-Services), basierend auf einer fest integrierten Echtzeitanwendung von syntaktischen und semantischen Analysen.

Diese Analysen basieren auf der strukturierten, abstrakten Repräsentation des Programmtexts (wird als **abstrakter Syntaxbaum** dargestellt). Dies wird durch einen im Hintergrund laufenden Parser erreicht, der in Echtzeit den eingegebenen Text parst, und eine Fehlerbehebung ermöglicht die korrekte Funktionsweise der Editor-Services, auch wenn mehrere syntaktische Fehler im Quelltext vorhanden sind.

Die Ergebnisse der Analysen können mit Hilfe von **Origin Tracking** [DKT93] Methoden mit dem ursprünglichen Quelltext in Verbindung gebracht werden. Der folgende Abschnitt soll die wichtigsten Editor-Services vorstellen und einen kurzen Überblick zu Spoofax geben.

[1] http://spoofax.org

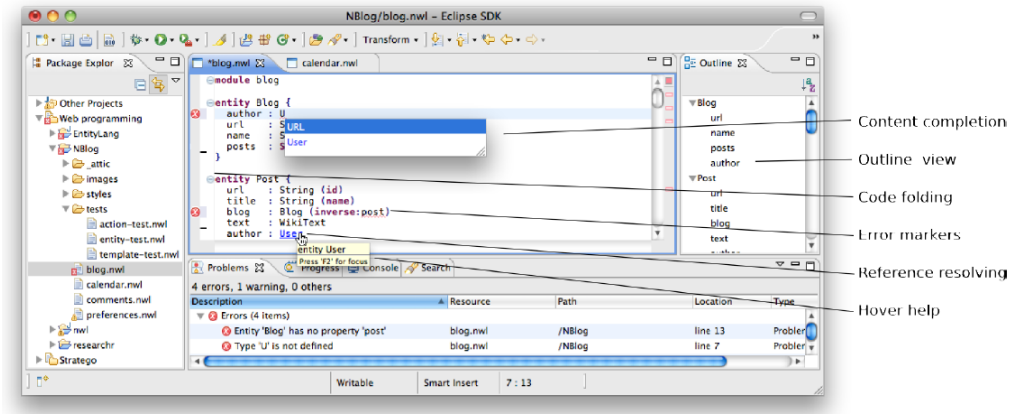

Abbildung 4.1: Editor-Services einer Websprache
[Quelle: [KV10]]

4.1.1 Spoofax im Überblick

Die in diesem Abschnitt vorgestellten Inhalte sind aus [Tou10] und [KV10] übernommen, wobei einzelne Details, die nicht für ein grundlegendes Verständnis notwendig sind, hier nicht aufgeführt sind.

Spoofax kann aus drei unterschiedlichen Perspektiven betrachtet werden:

1. Endbenutzer arbeiten mit den Editor-Services, die speziell für ihre domänenspezifische Sprache entwickelt werden.

2. Spoofaxentwickler pflegen die zugrundeliegende Architektur und deren Sprachkomponenten.

3. Entwickler von Programmiersprachen nutzen Spoofax für die Definition einer neuen Sprache.

Editor-Services: Moderne Integrated Development Environments (IDEs) wie Eclipse, bieten eine Vielzahl von sprachenspezifischen Editor-Services an. Der Editor überprüft die Syntax des Programmtextes während der Benutzer diesen eingibt und hebt bestimmte Textelemente hervor, basierend auf der syntaktischen Struktur des Programmtextes. Der syntaktische Zustand des Parsers wird u.a. für folgende Services genutzt (siehe Abb. 4.1):

- Wortvervollständigung
- Automatische Einfügung und Hervorhebung von Klammern
- Automatische Texteinrückung
- Code Folding
- Outline View
- Quick Outline Feature

Basierend auf einer semantischen Echtzeitanalyse des abstrakten Syntaxbaumes, können Fehler und Warnungen im Editorfenster angezeigt werden. Programmnavigation und -verständnis werden durch die Auflösung von Referenzen, das Hervorheben von Programmteilen und das Einblenden von kleinen Hilfefenstern (wenn mit der Maus über bestimmte Textabschnitte gefahren wird) unterstützt. Die Wortvervollständigung zeigt dem Entwickler die zulässigen Arten an, wie das aktuelle Sprachkonstrukt vervollständigt werden kann.

Komponentenarchitektur: Traditionell werden Programmiersprachen zunächst als selbständige Compiler entwickelt und IDEs werden erst später hinzugefügt. Dabei muss aber oft eine große Anzahl an Komponenten des Compilers reimplementiert werden, um die benötigten Editor-Services zu realisieren.

Die Komponenten eines Compilers, dazu gehören Parser, semantische Analysen, Transformationen und Codegenerierung, spielen eine wichtige Rolle in Editor-Services aufgrund der abstrakten Syntax und der semantischen Analyse des Programmtextes.

In Abb. 4.2 sind die Beziehungen zwischen den IDE Komponenten zu sehen. Dabei sind die rot markierten Komponenten Bestandteile des Compilers. Es gibt zwei Arten von Abhängigkeiten zwischen den einzelnen Komponenten:

- Generative Abhängigkeit: Eine Komponente kann automatisch von einer anderen Komponente abgeleitet werden (durchgezogener Pfeil in Abb. 4.2).
- Verwendungsabhängigkeit: Eine Komponente ruft eine andere Komponente auf (gestrichelter Pfeil in Abb. 4.2).

Die Grammatik und der Parser der Sprache bilden die Wurzel des Abhängigkeitsgraphen, da die syntaktische Struktur der Programme die Basis für die Implementation aller weiteren Services bildet. Zu diesen Services gehören u.a. Präsentations- und Bearbeitungs-Services, die automatisch von der Grammatik abgeleitet werden können. Sie können später vom Benutzer verändert und angepasst werden. Die semantischen Services können nicht automatisch von der Grammatik abgeleitet werden, da sie von der Interpretation der syntaktischen Struktur des Programms abhängig sind.

Definition neuer Sprachen: In Spoofax erfolgt die Definition einer neuen Sprache durch ein Eclipse-Projekt, das die sprachenspezifischen Elemente einer IDE definiert. In Tabelle 4.1 ist die standardmäßige Struktur eines solchen Projekts dargestellt. Ein Projekt besteht aus den drei folgenden Hauptkomponenten, die durch mehrere Module definiert werden:

- Syntaxdefinition: Die Syntax der Sprache wird mit SDF definiert. Die Dateien haben die Endung ***.sdf**.
- Editor-Service Deskriptoren: Die Editor-Services werden durch deklarative, regelbasierte Editor-Deskriptoren definiert. Diese können z.B.

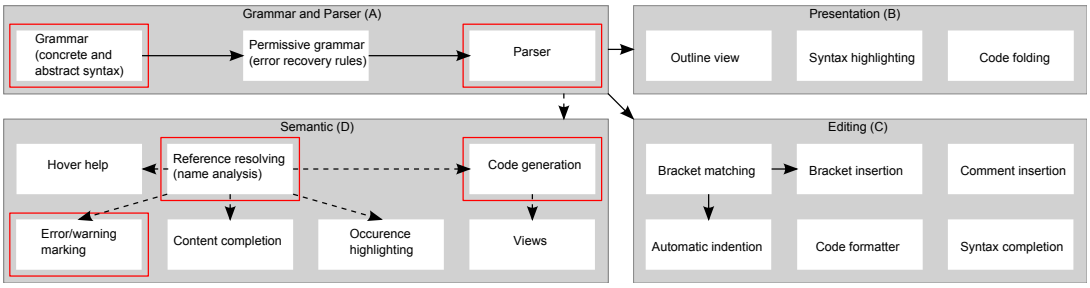

Abbildung 4.2: Beziehungen zwischen den IDE Komponenten [Quelle: [KV10]]

für die Definition von Präsentations- und Änderungs-Services genutzt werden. Die Dateien haben die Endung ***.esv**.

- Semantikdefinition: Die semantische Definition wird mit der Sprache Stratego spezifiziert. Stratego bietet Regeln für die Analyse, die Transformation und die Codegenerierung an. Die Dateien haben die Endung ***.str**.

Custom	**Generated**
Syntax definition	
Lang.sdf	Common.sdf
Editor service descriptors	
Lang.main.esv	
Lang-Builders.esv	Lang-Builders.generated.esv
Lang-Colorer.esv	Lang-Colorer.generated.esv
Lang-Completions.esv	Lang-Completions.generated.esv
Lang-Folding.esv	Lang-Folding.generated.esv
Lang-Outliner.esv	Lang-Outliner.generated.esv
Lang-References.esv	Lang-References.generated.esv
Lang-Syntax.esv	Lang-Syntax.generated.esv
Semantic definition	
lang.str	
check.str	
generate.str	

Tabelle 4.1: Komponenten einer Sprachdefinition

Entwickler können diese Module auf ihre Bedürfnisse hin anpassen. Dateien mit der Bezeichnung **.generated** weisen darauf hin, dass sie generiert und nicht selbst definiert wurden. Diese Dateien werden jedes Mal neu generiert, wenn das Projekt neu aufgebaut wird und sollten vom Entwickler nicht verändert werden.

Wenn der Entwickler trotzdem Änderungen vornehmen möchte, kann er dies in den dazugehörigen, selbst definierbaren Dateien tun (vgl. Spalte

```
1  module <ModuleName> [<Symbol>*]
2    <ImportSection>*
3    <ExportSection>*
4    <HiddenSection>*
5
6  ImportSection
7    imports
8      <Modulename>+
9
10 ExportSection
11   exports
12     <Grammar>+
13
14 HiddenSection
15   hiddens
16     <Grammar>+
```

Listing 4.1: Struktur eines Moduls

Custom in Tabelle 4.1). Damit können die standardmäßigen Einstellungen durch benutzerspezifische Einstellungen ersetzt werden.

Der nächste Abschnitt gibt eine kurzen Einblick in den SDF. Die hier vorgestellten Informationen sind aus [BKV07] übernommen, wobei nur auf die grundlegenden Eigenschaften des SDF näher eingegangen wird.

4.1.2 Syntax Definition

SDF ist die richtige Technologie, wenn man die Syntax

- einer existierenden Sprache, wie z.B. C, C++ oder Java beschreiben möchte;
- einer eingebetteten Sprache beschreiben möchte, indem man mehrere Grammatiken kombiniert;
- einer selbst definierten domänenspezifischen Sprache beschreiben möchte.

Das Hauptziel einer SDF Definition ist die Beschreibung der Syntax. Ein weiteres Ziel ist die Generierung eines Parsers, ausgehend von dieser Definition. SDF basiert hauptsächlich auf kontextfreien Grammatiken, wie z.B. die Erweiterte Backus-Naur-Form (EBNF)[2], ist aber, aufgrund von zahlreichen Erweiterungen, besser für die Beschreibung der Syntax von komplexen Programmiersprachen geeignet, weil:

- SDF erlaubt die Definition von modularen Grammatiken. Dadurch können Grammatiken kombiniert und wiederverwendet werden. Dies vereinfacht die Handhabung von eingebetteten Sprachen oder verschiedenen Dialekten einer Sprache. Ein Modul hat i.d.R. den in Listing 4.1 gezeigten Aufbau und kann die folgenden grammatikalischen Ausdrücke beinhalten:
  - Importe

[2] Quelle: http://de.wikipedia.org/wiki/EBNF

```
1 <Symbol>* -> <Symbol> {<Attribute1>,<Attribute2>,...}
```

Listing 4.2: Grundlegender Aufbau einer Produktion

  - Terminal bzw. Nichtterminal Symbole: Literale, Sorts und Buchstabenklassen
  - Startsymbole
  - Produktionen der Grammatik (siehe List. 4.2): lexikalische Produktionen beschreiben die Low-Level-Struktur von Texten (also lexikale Token) und kontextfreie Produktionen beschreiben die High-Level-Struktur.
  - Disambiguierungskonstrukte

- Mit SDF können lexikalische und kontextfreie Syntax gemeinsam definiert werden.

- SDF unterstützt eine deklarative Disambiguierung: Eine geschachtelte Anwendung von Produktionsregeln auf einen Eingabestring, kann zu einem oder mehreren Parsebäumen führen [Vin07]. Diese Ambiguität ist unerwünscht, da die Parsebäume später noch für eine semantische Analyse des Programms benötigt werden und eindeutig sein müssen. Um dieses Problem zu beheben, werden Methoden der Disambiguierung benötigt, dazu gehören:
  - Assoziativität: Produktionen werden um Attribute angereichert
  - Priorität: definiert die relative Rangfolge zwischen Produktionen
  - Restriktion: verhindert die Anwendung einer Produktion für ein bestimmtes Nichtterminal

  Jedes der oben genannten Konstrukte führt zu einem spezifischen Ableitungsfilter. Zunächst werden anhand der Produktionen alle möglichen Ableitungen für einen Eingabestring (Programm) generiert. In einem zweiten Schritt werden diese Ableitungen mit Hilfe der vorgestellten Disambiguierungsmethoden gefiltert, bis nur noch eine mögliche Ableitung übrig bleibt. Diese Ableitung kann danach für eine Weiterverarbeitung des Programms genutzt werden. Einige Beispiele für eine solche Weiterverarbeitung, die eine Manipulation des Programmquelltextes erfordert, sind:
  - Kompilierung
  - Programmgeneration
  - domänenspezifische Optimierung
  - Reverse-Engineering

Die nächsten Abschnitte zeigen, wie eine Programmtransformation in einer strukturierten und robusten Art und Weise durchgeführt werden kann. Dazu

wird das **Stratego/XT Framework** genutzt. Die vorgestellten Inhalte und Informationen sind aus [Str10] übernommen.

4.1.3 Stratego/XT

Die im letzten Abschnitt erwähnten Manipulationen des Programmquelltextes können als Transformation von einem Programm in ein abgeleitetes Programm angesehen werden. Für eine solche Programmtransformation wird das Stratego/XT Framework genutzt. Es besteht aus den folgenden zwei Teilen:

Stratego: Die Stratego Sprache unterstützt:

- Rewrite Regeln, die zum Umschreiben von Termen dienen.
- Rewriting Strategien, die die Anwendung der Rewrite Regeln steuern.
- Eine konkrete Syntax zur Beschreibung von Rewrite Regeln in der zugrunde liegenden Objektsprache.
- Dynamische Rewrite Regeln für die Beschreibung von kontextsensitiven Transformationen.

Alle diese Komponenten ermöglichen die Entwicklung von Transformationskomponenten auf einer High-Level-Abstraktion.

XT: Bietet eine Kollektion von erweiterbaren und wiederverwendbaren Transformationswerkzeugen, auch Tools genannt, an. Zu diesen Tools gehören u.a.

- leistungsfähige Parser und Pretty-Printer Generatoren und
- Entwicklungswerkzeuge für Grammatiken.

Ziel des Frameworks ist es, die Entwicklung

- einer Infrastruktur für Programmtransformationen,
- domänenspezifischer Sprachen und
- von Compilern und Programmgeneratoren zu unterstützen.

Eine Programmtransformation kann als die Überführung eines Programms in ein anderes Programm angesehen werden (siehe Abb. 4.3). Dabei wird die Programmiersprache, in der das zu transformierende Programm geschrieben ist, als Quellsprache bezeichnet und die Programmiersprache, in die das Programm transformiert wird, als Zielsprache.

Programmtransformationen werden in einer Vielzahl von Bereichen der Softwareentwicklung verwendet und lassen sich in zwei Hauptkategorien einteilen:

- Umwandlung (Translation): Die Sprache (Quellsprache) des zu transformierenden Programms unterscheidet sich von der Sprache (Zielsprache) des resultierenden Programms.

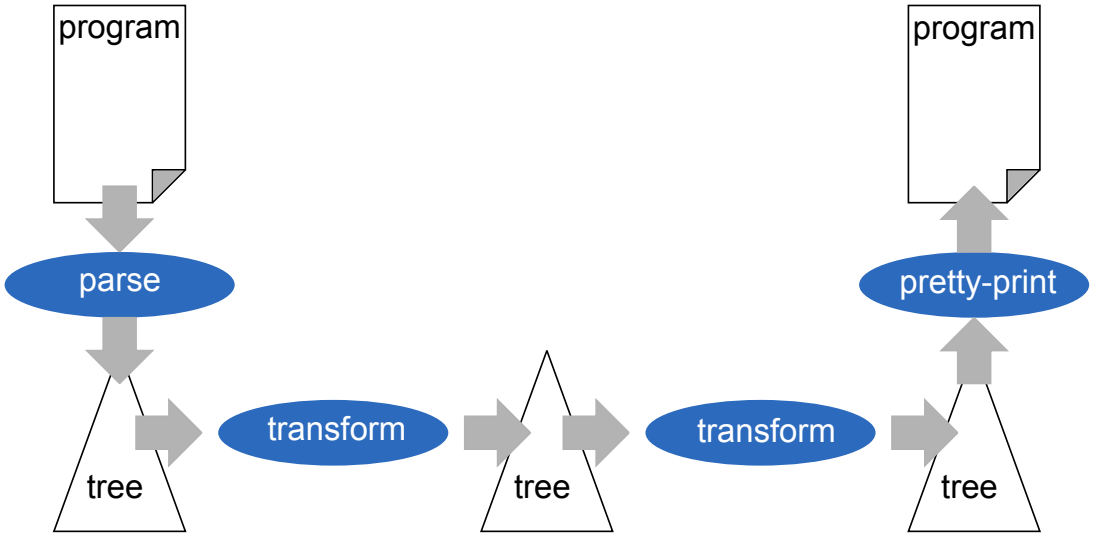

Abbildung 4.3: Ablauf einer Programmtransformation
[Quelle: http://hydra.nixos.org]

- Umformulierung (Rephrasing): Die Quell- und Zielsprache der zu transformierenden Programme ist gleich, da die Programme lediglich umformuliert werden.

Stratego: Stratego [BKVV08] bildet den Kern des Stratego/XT Frameworks und ist eine kleine, aber sehr effiziente domänenspezifische Sprache für Programmtransformationen. Die Sprache basiert auf programmierbaren Rewriting Strategien, die besonders gut geeignet für die Transformation und Traversierung von Baumstrukturen sind.

Um ein Programm, das meist in Form von Text vorliegt, in eine solche baumartige Struktur umwandeln zu können, wird ein Parser benötigt. Der Parser liest den Quelltext ein und generiert daraus einen abstrakten Syntaxbaum, der in Stratego/XT durch einen Term im Annotated Term (ATerm) Format repräsentiert wird.

Das folgende Beispiel soll die Verwendung von Termen, Rewrite Regeln und Rewrite Strategien verdeutlichen[3].

Beispiel: In Stratego werden Programme durch Terme beschrieben, die als Bäume repräsentiert werden. Die abstrakte Syntax der zugehörigen Programmiersprache wird durch Signaturen beschrieben. In List. 4.3 ist die Signatur der arithmetischen Grundoperatoren +, -, * und / dargestellt. Mit dieser Signatur können Terme gebildet werden, die z.B. den Ausdruck

$$1 + 8 * 9$$

repräsentieren und als

$$Add(1, Mul(8, 9))$$

gelesen werden.

[3] Quelle: http://strategoxt.org/Stratego/StrategoLanguage

```
1 signature
2   sorts Expr
3   constructors
4     Add : Expr * Expr -> Expr
5     Sub : Expr * Expr -> Expr
6     Mul : Expr * Expr -> Expr
7     Div : Expr * Expr -> Expr
```

Listing 4.3: Signatur für arithmetische Grundoperatoren

```
1   Eval: Add(l, r) -> Int(n) where n := <add> (l, r)
```

Listing 4.4: Rewrite Regel für Additionsbaum

Die in List. 4.4 definierte Regel mit dem Namen **Eval** kann nun auf einen Additionsbaum angewendet werden, dessen Unterbäume Konstanten sind. Dabei wird der Baum durch die Summe des linken und rechten Unterbaums ersetzt.

Die Rewrite Regel **Eval** kann nicht nur auf einzelne Terme angewendet werden, sondern auch auf größere Bäume, indem eine Strategie, wie z.B. bottomup, genutzt wird (siehe List. 4.5). Diese Strategie versucht auf jeden Teilbaum die Regel **Eval** anzuwenden. Ist die Anwendung auf den jeweiligen Teilbaum erfolgreich, wird dieser durch das Ergebnis der Rewrite Regel ersetzt, ansonsten wird der ursprüngliche Teilbaum beibehalten.

Der nächste Abschnitt befasst sich mit der Erweiterung von FeatureHouse um den SDF und die Sprache Stratego.

4.2 Erweiterung von FeatureHouse

Die Integration einer neuen Sprache in FeatureHouse basiert fast ausschließlich auf der Spezifikation der Grammatik der Sprache, zuzüglich einiger Attribute (vgl. Abschnitt 2.2.4), welche als Annotationen verwendet werden, und Kompositionsregeln. Manche Sprachen können durch eine Transformation bereits integrierter Grammatiken in FeatureHouse eingebunden werden. Für andere Sprachen muss die Grammatik von Grund auf neu geschrieben werden, was einen sehr viel höheren Aufwand bedeutet.

Bei der Entwicklung der beiden neuen Grammatiken wurde teilweise auf die bereits integrierte Grammatik für Java zurückgegriffen. Hinzu kamen noch einige spezifische Konstrukte von SDF und Stratego, für die neue Grammatikbestandteile erstellt werden mussten.

```
1   bottomup(try(Eval))
```

Listing 4.5: Bottomup Strategie für Rewrite Regel *Eval*

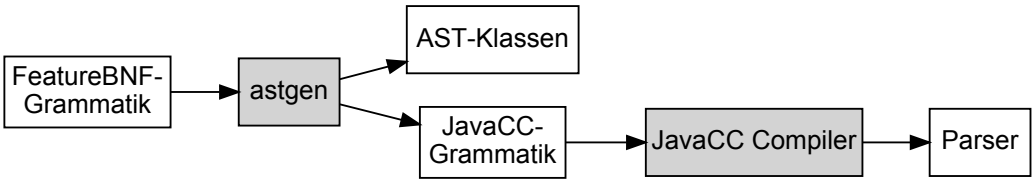

Abbildung 4.4: Generierung des Parsers ausgehend von der Grammatik

In den folgenden Abschnitten werden die benötigten Schritte für das Einbinden von SDF und Stratego etwas näher betrachtet:

Spezifikation der Grammatik: Für die Integration einer neuen Sprache, muss deren Grammatik im FeatureBNF-Format vorliegen und in den FSTGenerator (vgl. Abschnitt 2.2.4) integriert werden. Ausgehend von dieser Grammatik können dann mit Hilfe des FSTGenerators andere Tools, wie z.B. der Parser oder der Pretty Printer, für die Sprache automatisch generiert werden. In Abb. 4.4 ist dieser Vorgang im Überblick dargestellt:

- Das Tool **astgen** generiert anhand der Grammatik einen **Abstract Syntax Tree (AST)** und eine **JavaCC** Grammatik.
- Der **JavaCC Compiler** erstellt aus der JavaCC Grammatik den Parser.

Aufgrund dieser beiden aufeinander folgenden Schritte, lässt sich die Grammatik in zwei Abschnitte aufteilen. Der erste Teil beinhaltet Anweisungen für den JavaCC Compiler, insbesondere die Deklarationen für den lexikalischen Analysator. Diese Deklarationen sind im JavaCC-Format beschrieben und werden vom astgen-Tool nicht prozessiert.

Der zweite Teil besteht aus annotierten Produktionen, die vom astgen-Tool verarbeitet werden. Beide Teile werden durch das Schlüsselwort **GRAMMARSTART** getrennt. Listing 4.6 zeigt einen kleinen Ausschnitt aus der Grammatik für SDF.

Artefact-Builder erstellen: Um den aus der JavaCC Grammatik generierten Parser verwenden zu können, muss ein sogenannter **Builder** erstellt werden. Dieser Builder dient als Vermittler – wird auch als **Wrapper** bezeichnet – zwischen FeatureHouse und dem erstellten Parser und legt fest, wann der Parser aufgerufen wird. Für SDF und Stratego wird jeweils ein eigener Builder benötigt.

Print-Visitor erstellen: Neben dem Parser wird automatisch auch ein Pretty Printer generiert und für diesen Pretty Printer wird ebenfalls ein Wrapper benötigt, der analog zum Builder, die Aufgabe hat, den Pretty Printer in FeatureHouse einzubinden. Dieser wird als **Print-Visitor** bezeichnet.

Artefact-Builder und Print-Visitor registrieren: Die beiden erstellten Wrapper müssen anschließend in FeatureHouse registriert werden, damit FeatureHou-

```
1  ...
2  SPECIAL_TOKEN :
3    { " " | "\t" | "\n" | "\r" | "\f" }
4  MORE :
5    { "%%" : IN_SINGLE_LINE_COMMENT |
6      <"/**" ~["/"]> { input_stream.backup(1); } : IN_FORMAL_COMMENT |
7      "/*" : IN_MULTI_LINE_COMMENT
8    }
9    <IN_SINGLE_LINE_COMMENT>
10     SPECIAL_TOKEN :
11       { <SINGLE_LINE_COMMENT: "\n" | "\r" | "\r\n" > : DEFAULT }
12   <IN_FORMAL_COMMENT>
13     SPECIAL_TOKEN :
14       { <FORMAL_COMMENT: "*/" > : DEFAULT }
15   <IN_MULTI_LINE_COMMENT>
16     SPECIAL_TOKEN :
17       { <MULTI_LINE_COMMENT: "*/" > : DEFAULT }
18   <IN_SINGLE_LINE_COMMENT,IN_FORMAL_COMMENT,IN_MULTI_LINE_COMMENT>
19     MORE :
20       { < ~[] > }
21 ...
22
23 GRAMMARSTART
24
25 NatInteger :
26   <INTEGER_LITERAL> ;
27 RealDigit:
28   <INTEGER_LITERAL> "." <INTEGER_LITERAL> ;
29 Int :
30   [<MINUS>] NatInteger ;
31 Real :
32   [<MINUS>] RealDigit ;
33 String :
34   <STRING_LITERAL> ;
35 @FSTTerminal(name="{<IDENTIFIER>}")
36 Id :
37   <IDENTIFIER> ;
38 @FSTNonTerminal(name="{ModuleDeclaration}")
39 Module :
40   ModuleDeclaration @! @!
41   [ImportDeclaration] @-!
42   [ExportDeclaration]
43   [HiddenDeclaration]
44   <EOF> ;
45 ...
```

Listing 4.6: FeatureBNF-Grammatik für SDF

```
1  public class FSTGenProcessor {
2    ...
3    public FSTGenProcessor() {
4      ...
5      registerArtifactBuilder(new SDFBuilder());
6      registerArtifactBuilder(new STRBuilder());
7      ...
8      registerPrintVisitor(new SDFPrintVisitor());
9      registerPrintVisitor(new STRPrintVisitor());
10     ...
11   }
12   ...
13 }
```

Listing 4.7: Registrieren der Builder und der PrintVisitor

se mit ihnen arbeiten kann. Dies erfolgt in der Java-Klasse **FSTGenProcessor.java** (siehe Listing 4.7).

Kapitel 5

Fallstudien

In diesem Kapitel werden zwei Fallstudien und die damit erzielten Ergebnisse vorgestellt. Bei der ersten Fallstudie handelt es sich um eine Sprache für arithmetische Ausdrücke, die neu definiert wurde. Die Sprache MoBL bildet die zweite Fallstudie. Sie ist eine frei verfügbare Programmiersprache für eine einfache und schnelle Entwicklung von Applikationen für mobile Geräte, wie z.B. Handys. Der nachfolgende Abschnitt beschreibt zunächst die Erstellung der Ausdruckssprache.

5.1 Sprache für Ausdrücke

In Eclipse lässt sich mit Hilfe des Projektassistenten[1] (siehe Abb. 5.1) sehr einfach ein neues *Spoofax/IMP* Projekt anlegen. Dieser erstellt standardmäßig (vgl. Tabelle 4.1) eine Beispielsprache mit den zugehörigen Editordefinitionen und öffnet zusätzlich die Editorfenster für die wichtigsten Bestandteile dieser Definitionen, wie in Abb. 5.2 dargestellt.

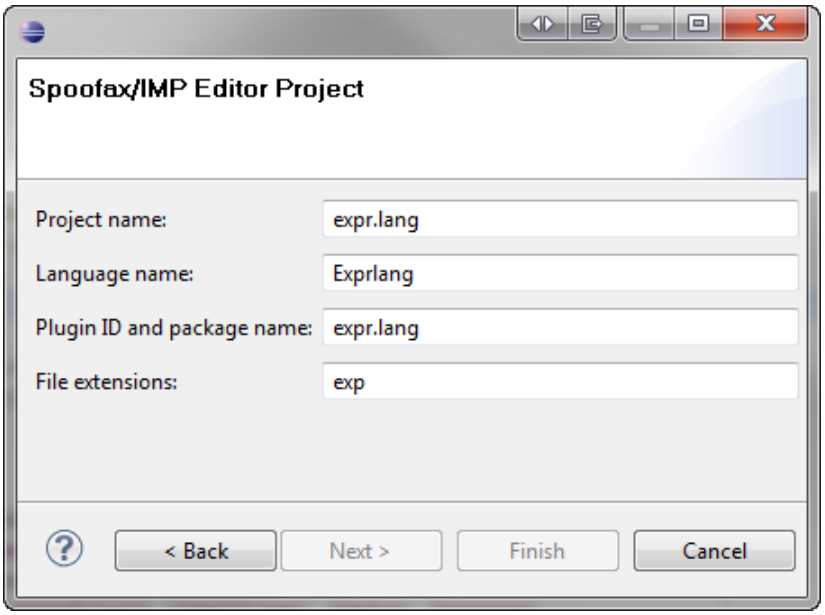

Abbildung 5.1: Anlegen eines neuen Spoofax/IMP Projekts

[1] *File -> New -> Project -> Spoofax/IMP editor project*

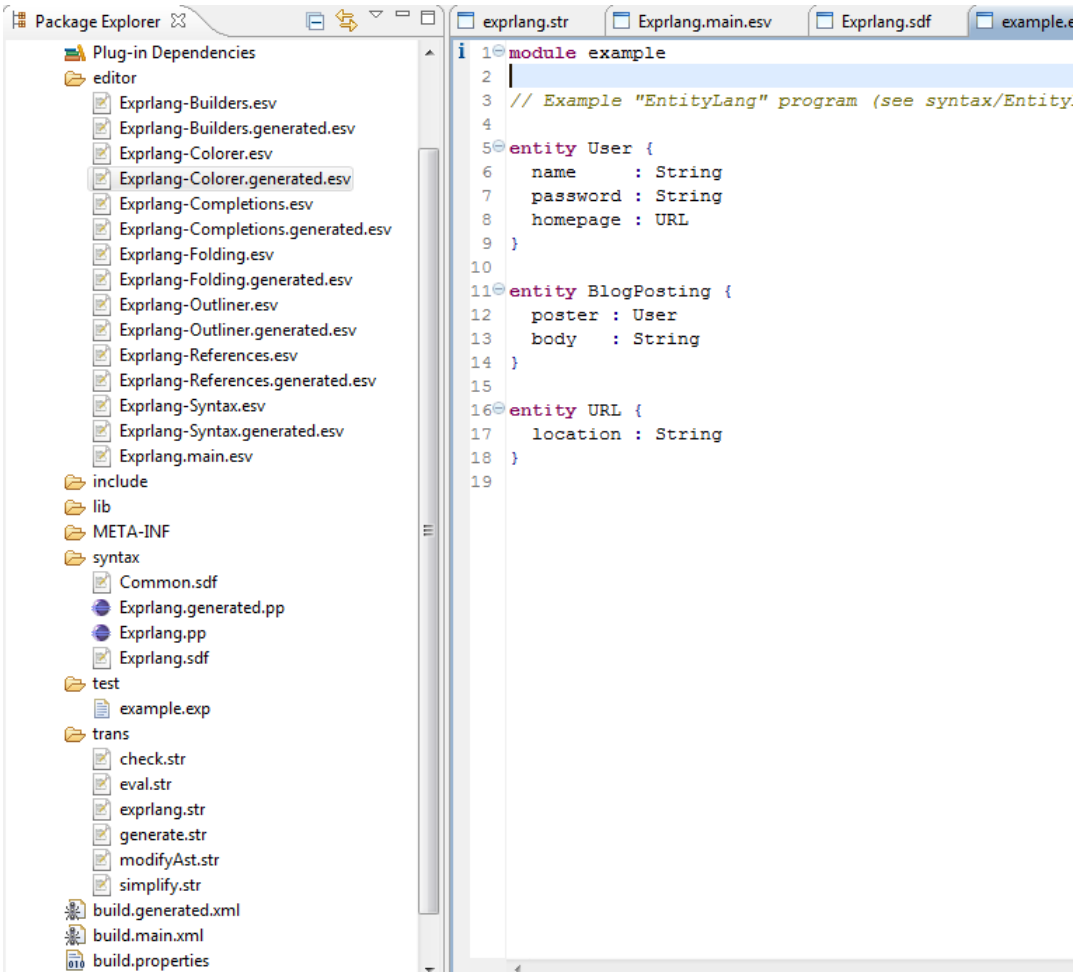

Abbildung 5.2: Übersicht des Spoofax/IMP Projekts

```
1  module Exprlang
2    imports Common
3  exports
4    sorts Type Stat Exp Var
5    context-free start-symbols
6      Start
7    context-free syntax
8      Stat*                       -> Start {cons("Start")}
9      Var ":=" Exp ";"            -> Stat {cons("Assign")}
10     ID                          -> Type {cons("Type")}
11     "(" Exp ")"                 -> Exp {bracket}
12     "-" Exp                     -> Exp {cons("UnaryMinus")}
13     INT                         -> Exp {cons("Int")}
14     "eval" "(" Exp ")" ";"      -> Stat {cons("Eval")}
15     "print" "(" Exp ")" ";"     -> Stat {cons("Print")}
16     "simplify" "(" Exp ")" ";" -> Stat {cons("Simplify")}
17     "var" ID ":" Type ";"       -> Stat {cons("VarDec")}
18     ID                          -> Var
19     Var                         -> Exp {cons("Var")}
20     STRING                      -> Exp {cons("String")}
21     Exp "+" Exp                 -> Exp {cons("Add"),assoc}
22     Exp "-" Exp                 -> Exp {cons("Sub"),left}
23     Exp "*" Exp                 -> Exp {cons("Mul"),assoc}
24     Exp "/" Exp                 -> Exp {cons("Div"),assoc}
25     Exp "%" Exp                 -> Exp {cons("Mod"),non-assoc}
26   context-free priorities
27     "-" Exp -> Exp
28   > {left:
29         Exp "*" Exp -> Exp
30         Exp "/" Exp -> Exp
31         Exp "%" Exp -> Exp }
32   > {left:
33         Exp "+" Exp -> Exp
34         Exp "-" Exp -> Exp }
```

Listing 5.1: Grammatik für die Ausdruckssprache

Das so erstellte Projekt dient als Basis für die eigentliche Definition der Ausdruckssprache. Als erstes muss die Grammatik (siehe Listing 5.1) der Sprache definiert werden. Hierfür wird SDF genutzt. Die Grammatik hat folgende Aufgabe:

1. Definition der konkreten Syntax (Schlüsselwörter etc.).
2. Definition der abstrakten Syntax (die Datenstruktur, die für die Analyse und Transformation eines Programms, geschrieben in der Ausdruckssprache, genutzt wird).
3. Generierung von Editor-Services.
4. Generierung des Parser, der die zentrale Komponente einer textuellen Sprache darstellt.

Um die Grammatikdefinition übersichtlicher und besser wiederverwendbar zu machen, wird sie in mehrere Module zerlegt. Das Modul **Exprlang.sdf** dient als Hauptmodul und legt fest, wie Ausdrücke, arithmetische Operationen auf Ausdrücken und Prioritäten der einzelnen Ausdrücke, definiert sind. Zusätzlich wird

```
1  module Exprlang-Colorer
2    imports Exprlang-Colorer.generated
3  colorer
4    keyword    : 205 112 084 bold   // salmon
5    identifier : default
6    _.Var      : 178  034 034 bold  // firebrick
7    _.Type     : 139  0 139 bold    // magenta
8    Exp.String : 0 0 255            // blue
9    number     : 0 100 0            // dark green
10   layout     : 105 105 105 italic // gray
```

Listing 5.2: Editor-Service: Syntax Highlighting

```
1  module Exprlang-Folding
2    imports Exprlang-Folding.generated
3  folding
4    Start.Start
```

Listing 5.3: Editor-Service: Code Folding

das Modul **Common.sdf** importiert, das zuständig für die Definition von Low-Level-Konstrukten, wie **ID**, **INT**, **STRING**, **Kommentare** und **LAYOUT**, ist.

Wie in Abschnitt 4.1.1 (Definition neuer Sprachen) bereits beschrieben, werden die Editor-Services durch Editor-Deskriptoren, die für die Definition von Präsentations- und Änderungs-Services genutzt werden, spezifiziert. Für die Ausdruckssprache sind die folgenden Editor-Services definiert:

Syntax Highlighting: Legt die graphische Darstellung von Programmkonstrukten, wie z.B.:

- Schlüsselwörtern,
- Operatoren,
- Identifier,
- Strings,
- Zahlen,
- Typen und
- Variablen fest (siehe Listing 5.2).

Code Folding: Legt fest, welche Textabschnitte in sogenannte Folds bzw. Falten gruppiert werden, die dann vom Benutzer im Editor einfach ein- und ausgeblendet werden können (siehe Listing 5.3). Hier wird der Einfachheit halber das gesamte Programm zu einem Fold gruppiert.

Builder: Legt die Einträge im Transformationsmenü (siehe Abb. 5.3 und Listing 5.4) fest. Damit können Transformationsregeln auf eine gesamte Datei oder nur auf einen ausgewählten Teil einer Datei angewendet werden. Markiert der Benutzer einen Teil des Programms und klickt mit der Maus im

```
1  module Exprlang-Builders
2    imports Exprlang-Builders.generated
3  builders
4    provider : include/exprlang.ctree
5    provider : include/exprlang-java.jar
6    observer : editor-analyze
7    builder  : "Generate Java code (for selection)"   =
8               generate-java (openeditor) (realtime)
9    builder  : "Show abstract syntax (for selection)" =
10              generate-aterm (openeditor) (realtime) (meta) (source)
11   builder  : "Show modified abstract syntax (for selection)" =
12              generate-modified-aterm (openeditor) (realtime) (meta) (source)
```

Listing 5.4: Editor-Service: Builder

```
1  module Exprlang-Completions
2    imports Exprlang-Completions.generated
3  completions
4    completion template : Stat = "print()"
5    completion template : Stat = "eval()"
6    completion template : Stat = "simplify()"
```

Listing 5.5: Editor-Service: Wortvervollständigung

Transformationsmenü z.B. auf den Punkt **Show abstract syntax (for selection)**, so wird für diesen Programmteil der entsprechende abstrakte Syntaxterm in einem neuen Editorfenster dargestellt.

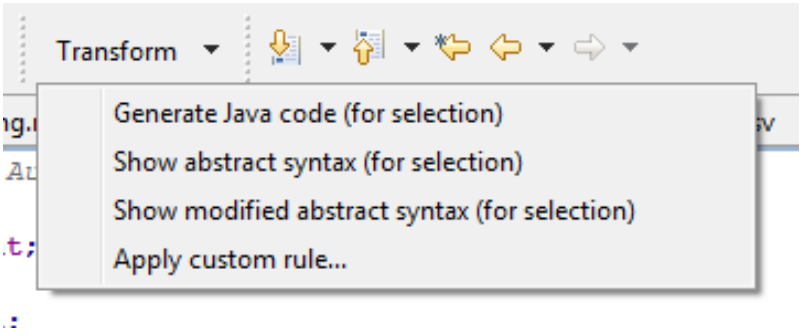

Abbildung 5.3: Menü zur Auswahl von Transformationen

Wortvervollständigung: Legt fest, für welche Programmteile die Wortvervollständig verfügbar ist (siehe Listing 5.5). Gibt der Benutzer im Editor, z.B. den Präfix **pri** des Schlüsselwortes **print** ein und drückt die Tastenkombination **STRG+LEERTASTE**, erscheint ein Popup mit der möglichen Wortvervollständigung, die der Benutzer dann auswählen kann (siehe Abb. 5.4).

Outline View: Legt fest, welche Programmkonstrukte in der Outline View dargestellt werden (siehe Listing 5.6). Zu diesen Konstrukten gehören alle Zuweisungen (**Assign**), die im Programm auftreten, und die Aufrufe von **Print**,

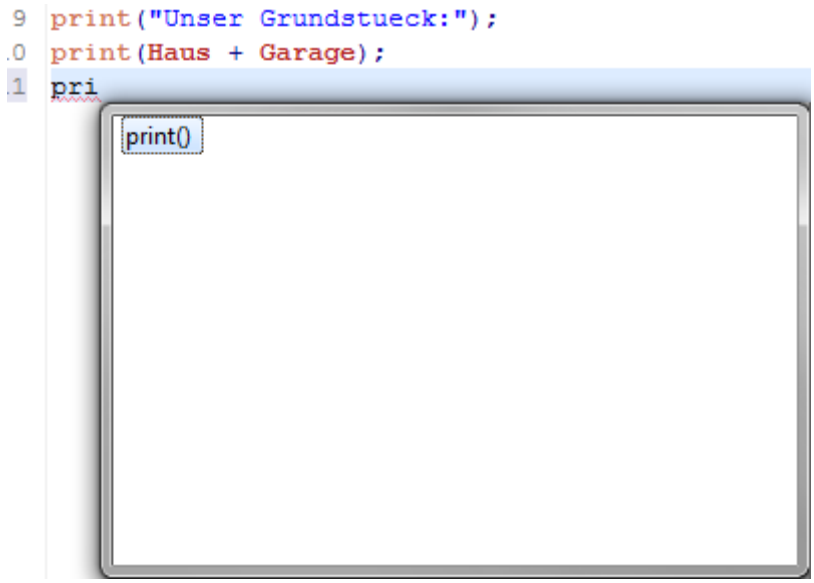

Abbildung 5.4: Editor-Service: Wortvervollständigung

```
1 module Exprlang-Outliner
2   imports Exprlang-Outliner.generated
3 outliner
4   Stat.Eval
5   Stat.Print
6   Stat.Simplify
7   Stat.Assign
```

Listing 5.6: Editor-Service: Outline View

Eval und **Simplify** (siehe Abb. 5.5).

Reference Resolving: Definiert die Auflösung von Referenzen und das Einblenden von kleinen Hilfefenstern, wenn mit der Maus über bestimmte Textabschnitte gefahren wird (siehe Listing 5.7 und Abb. 5.6).

Wie in Kapitel 3 bereits beschrieben, wird im Rahmen dieser Masterarbeit das extraktive Vorgehensmodell zur Erstellung einer Produktlinie genutzt. Die im vorherigen Abschnitt definierte Sprachdefinition ist Ausgangspunkt für eine Refaktorisierung in Features. Dabei wird der Quellcode der Sprachdefinition, die aus Syntax-, Typ- und Semantikregeln besteht, in Features verschoben (siehe Abschnitt 5.1.2), ohne das sich dabei das Verhalten der Anwendung ändert. Mit

```
1 module Exprlang-References
2   imports Exprlang-References.generated
3 references
4   reference _ : editor-resolve
5   hover _     : editor-hover
```

Listing 5.7: Editor-Service: Reference Resolving

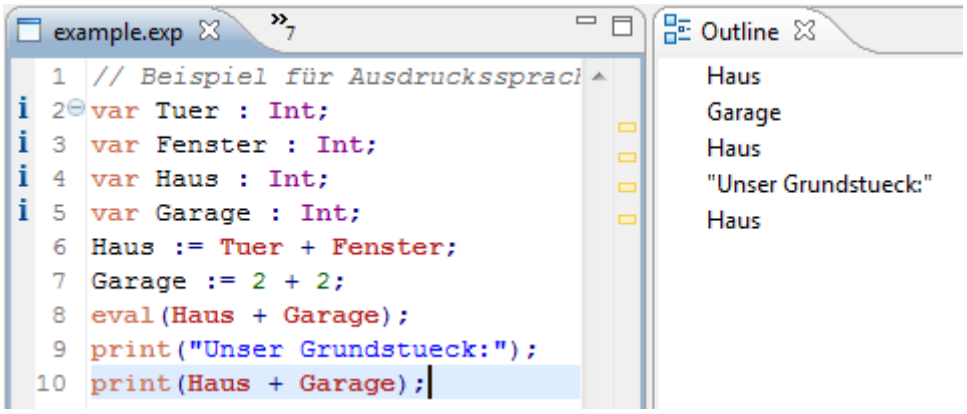

Abbildung 5.5: Editor-Service: Outline View

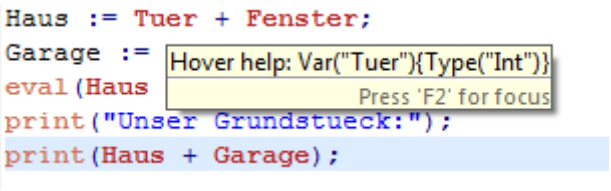

Abbildung 5.6: Editor-Service: Reference Resolving

Hilfe der gefundenen und extrahierten Features, können dann unterschiedliche Varianten von Sprachdefinitionen erstellt werden (siehe Abschnitt 5.1.3).

5.1.1 Feature–Lokalisation

Durch die Analyse des Quellcodes der Sprachdefinition, konnten die folgenden Features bestimmt werden:

- Add
- Sub
- Mul
- Div
- Mod
- Eval
- Print
- Simplify

Mit Hilfe der Features *Add*, *Sub*, *Mul*, *Div* und *Mod* können arithmetische Ausdrücke, wie z.B.

$$((6 \,\%\, 2 + 10) * (3 - 2)) \,/\, 8$$

beschrieben werden. Die Features *Eval*, *Print* und *Simplify* repräsentieren Methoden, mit denen die arithmetischen Ausdrücke evaluiert, ausgegeben oder vereinfacht werden können.

In einem ersten Schritt werden die Stellen im Quellcode, die zu einem bestimmten Feature gehören, mit Annotationen versehen, um sie leichter im Quellcode wiederfinden zu können. In Abb. 5.7 kann man gut erkennen, dass der Feature-Code der einzelnen Features in mehreren Modulen der Sprachdefinition verteilt vorliegt.

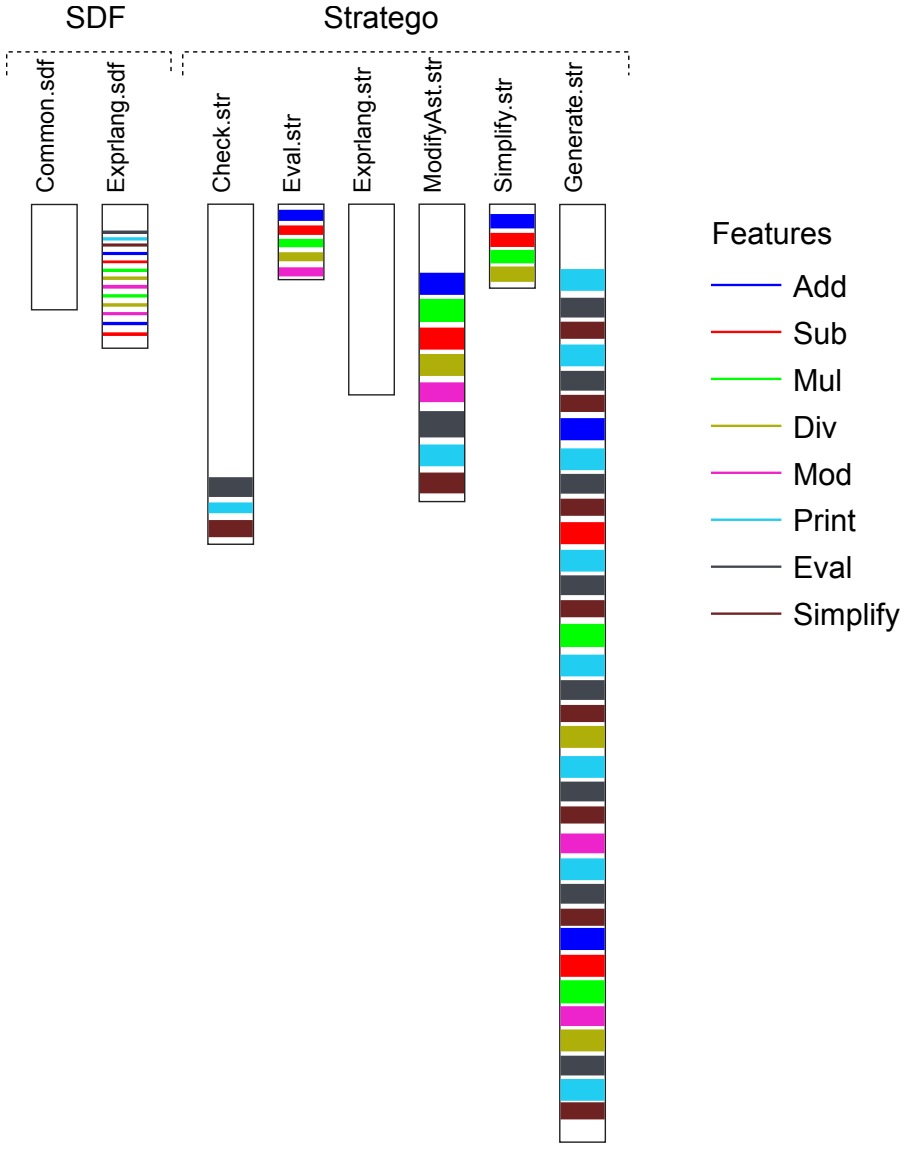

Abbildung 5.7: Lokalisation des Feature-Codes

Für die gefundenen Features wird anschließend in einem zweiten Schritt ein Feature-Modell und -Diagramm erstellt (vgl. Abschnitt 2.2.2), um die Features und deren Beziehungen graphisch darstellen zu können (siehe Abb. 5.8).

Im Feature-Diagramm kann man gut erkennen, dass die Features *Add*, *Sub* und *Print* obligatorisch sind – dunkel ausgefüllter Kreis – und bei einer Feature-Selektion (vgl. Abschnitt 2.2.2) immer ausgewählt werden müssen. Diese Festlegung ist einerseits willkürlich und anderseits beruht sie auf der Tatsache, dass die Addition und Subtraktion zu den Grundrechenarten gehören. Die restlichen Features sind optional und können bei Bedarf ausgewählt werden.

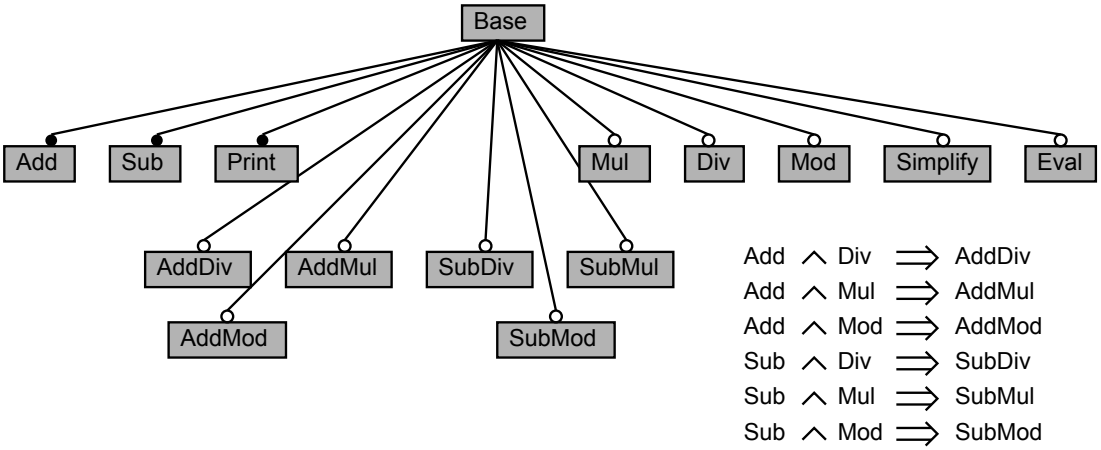

Abbildung 5.8: Feature-Diagramm der Ausdruckssprache

Werden bei einer Feature-Selektion von den optionalen Features zusätzliche Features, wie z.B. das Feature *Div*, ausgewählt, kann dies zu einer Feature-Interaktion mit bereits ausgewählten Features, in diesem Fall mit den Features *Add* und *Sub*, führen. Daher muss auch noch der entsprechende Feature-Code von *AddDiv* bzw. *SubDiv* berücksichtigt werden. Dies wird durch die Bedingungen

$$Add \wedge Div \Rightarrow AddDiv$$

und

$$Sub \wedge Div \Rightarrow SubDiv$$

im Feature-Diagramm gekennzeichnet.

5.1.2 Feature–Dekomposition

Nachdem die Features im Quellcode der Sprachendefinition lokalisiert wurden, was durch Annotationen gekennzeichnet ist, wird der jeweilige Feature-Code in Feature-Module verschoben (vgl. Abschnitt 2.2.2). Dieses Vorgehen soll am Beispiel des Features *Add* genauer betrachtet werden.

Wie in Abb. 5.7 zu sehen ist, ist der Feature-Code des Features *Add* auf die folgenden Module der Sprachdefinition verteilt:

- Exprlang.sdf
- eval.str
- modifyAst.str

```
1 module Exprlang
2 exports
3   context-free syntax
4     Exp "+" Exp  -> Exp {cons("Add"),assoc}
5   context-free priorities
6     "-" Exp -> Exp
7   > {left: Exp "+" Exp -> Exp }
```

Listing 5.8: Feature-Code des Features Add

- simplify.str
- generate.str

Für jedes dieser Module wird ein neues Modul mit demselben Namen angelegt und der entsprechende Feature-Code in das neu erstellte Modul kopiert. In Listing 5.8 ist der zum Feature *Add* gehörige Code, der im Modul *Exprlang.sdf* vorliegt, dargestellt.

Die erstellten Module werden dann in einem Ordner, der das Feature *Add* repräsentiert, abgelegt (siehe Abb. 5.9). Der Ordner enthält für jedes der oben aufgeführten Module ein entsprechendes Modul mit dem jeweiligen Feature-Code.

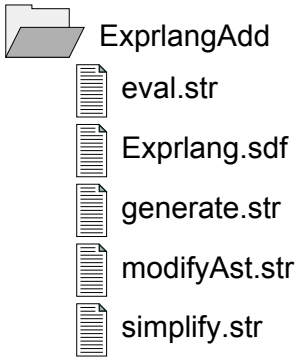

Abbildung 5.9: Ordnerstruktur für das Feature Add

Analog dazu, wird für die anderen Features ebenfalls der Feature-Code in neue Module kopiert und jeweils ein Ordner pro Feature angelegt. Die gesamte Ordnerstruktur ist in Abb. 5.10 dargestellt. Diese Ordnerstruktur ist wichtig für die Komposition der Features mit Hilfe von FeatureHouse, was im nachfolgenden Abschnitt näher betrachtet wird.

5.1.3 Feature–Komposition mit FeatureHouse

Die in den vorherigen Abschnitten gefundenen und extrahierten Features können in FeatureHouse zu verschiedenen Varianten komponiert werden. Dazu sind die folgenden Schritte notwendig:

Durchführung der Feature-Selektion: Im Verzeichnis (siehe Abb. 5.10), in dem die Feature-Module gespeichert sind, muss eine zusätzliche Datei mit der

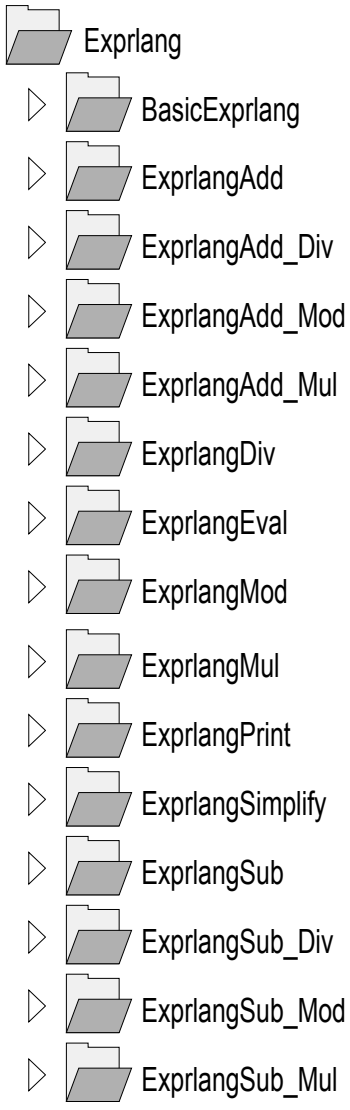

Abbildung 5.10: Ordnerstruktur aller Features

```
ExprlangComp.features:
  BaseExprlang
  ExprlangAdd
  ExprlangSub
  ExprlangPrint
```

Listing 5.9: Feature-Selektion zur Erstellung einer Variante

```
fstcomp Exprlang.launch:
  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <launchConfiguration type="org.eclipse.jdt.launching.localJavaApplication">
    <listAttribute key="org.eclipse.debug.core.MAPPED_RESOURCE_PATHS">
      <listEntry value="/fstcomp/composer/FSTGenComposer.java"/>
    </listAttribute>
    <listAttribute key="org.eclipse.debug.core.MAPPED_RESOURCE_TYPES">
      <listEntry value="1"/>
    </listAttribute>
    <stringAttribute key="org.eclipse.jdt.launching.MAIN_TYPE"
      value="composer.FSTGenComposer"/>
    <stringAttribute key="org.eclipse.jdt.launching.PROGRAM_ARGUMENTS"
      value="--expression
      examples/Exprlang/ExprlangComp.features"/>
    <stringAttribute key="org.eclipse.jdt.launching.PROJECT_ATTR"
      value="fstcomp"/>
  </launchConfiguration>
```

Listing 5.10: Konfigurationsdatei für die Ausdruckssprache

Endung **.features** angelegt werden. Sie enthält die ausgewählten Features, aus denen eine Variante der Sprachdefinition erstellt werden soll.

In Listing 5.9 ist die Datei *ExprlangComp.features* dargestellt, die der Feature-Selektion zur ersten Variante aus Tabelle 5.1 entspricht. Für die restlichen Varianten aus Tabelle 5.1 muss die Datei *ExprlangComp.features* entsprechend angepasst werden.

Anlage der Konfigurationsdatei: In der sogenannten Konfigurationsdatei *fstcomp Exprlang.launch* müssen neben einigen Eclipse-spezifischen Angaben (vgl. die *<listAttribute>* in Listing 5.10) die folgenden Einträge angegeben werden (vgl. die *<stringAttribute>* in Listing 5.10):

- die Java-Klasse *FSTGenComposer*, die den Kompositionsprozess anstößt
- die Datei *ExprlangComp.features*, die die Auswahl der Features, die durch FeatureHouse komponiert werden sollen, enthält
- das Projekt *fstcomp*, das die Hauptdatei *FSTGenComposer* enthält

In Tabelle 5.1 sind verschiedene Varianten von Sprachdefinitionen für die Ausdruckssprache dargestellt, die durch eine unterschiedliche Featureauswahl generiert werden können. Dabei sind die Module, die durch eine Feature-Interaktion zusätzlich ausgewählt werden müssen, durch das Symbol „#“ gekennzeichnet (vgl. Abschnitt 5.1.1) und in der Spalte Lines of Code (LOC) sind nur die

Anzahl der Zeilen der SDF- bzw. Stratego-Module aufgeführt. Die dritte Spalte enthält die Anzahl der Semantik- und Typregeln einer Variante der Ausdruckssprache.

Aus einer solchen Sprachvariante kann mit Hilfe von Eclipse ein Plug-In erstellt werden, das in Form einer Update-Seite auf einen Webserver hochgeladen oder auch nur lokal gespeichert werden kann. Das so erstellte Plug-In kann anschließend in einer neuen Eclipse-Instanz installiert und genutzt werden. Dieses Vorgehen wird im Abschnitt 5.2.3 für die Sprache MoBL genauer betrachtet.

Variante	**#Features**	**#Regeln**	**#LOC**
Variante_1	Base, Add, Sub, Print	53	476
Variante_2	Base, Add, Sub, Print, Div #AddDiv #SubDiv	59	521
Variante_3	Base, Add, Sub, Print, Mod #AddMod #SubMod	57	514
Variante_4	Base, Add, Sub, Print, Mul #AddMul #SubMul	60	523
Variante_5	Base, Add, Sub, Print, Div #AddDiv #SubDiv, Eval, Simplify	67	575
Variante_6	Base, Add, Sub, Print, Mod #AddMod #SubMod, Simplify	61	541
Variante_7	Base, Add, Sub, Print, Mul #AddMul #SubMul, Eval	64	550
Variante_8	Base, Add, Sub, Print, Eval, Simplify	61	530
Variante_9	Base, Add, Sub, Print, Div #AddDiv #SubDiv, Mod #AddMod #SubMod, Mul #AddMul #SubMul, Eval, Simplify	78	660

Tabelle 5.1: Varianten von Sprachdefinitionen: Exprlang

5.2 MoBL

Wie bereits am Anfang dieses Kapitels beschrieben, ist MoBL eine frei verfügbare Programmiersprache für eine einfache und schnelle Entwicklung von Applikationen für mobile Geräte [MoB11]. Durch die Nutzung modernster Hypertext Markup Language (HTML)-Technologien bietet MoBL die folgenden Vorteile [MoB11]:

- Präzise Sprache, mit der einfach und schnell Webapplikationen erstellt werden können
- Sehr gute Unterstützung durch die Entwicklungsumgebung Eclipse

- Schneller Speicher- und Testzyklus (jedes Mal wenn gespeichert wird, werden die Module im Hintergrund kompiliert und können sofort in einem Webbrowser getestet werden)

Im Gegensatz zur Ausdrucksprache, die neu entwickelt wurde, ist MoBL ein bereits bestehendes Projekt, das frei verfügbar ist und als Repository[2] heruntergeladen werden kann. Die in diesem Projekt enthaltenen Syntax-, Typ- und Semantikregeln, werden – wie bei der Ausdrucksprache auch – auf mögliche Features hin untersucht. Die gefundenen Features werden in Feature-Module ausgelagert und anschließend mit Hilfe von FeatureHouse komponiert. Dieses Vorgehen wird in den nächsten Abschnitten näher beschrieben.

5.2.1 Feature–Lokalisation

Durch die Analyse der Syntax-, Typ- und Semantikregeln konnten die folgenden Features bestimmt werden:

HTML: Dieses Feature ermöglicht die direkte Nutzung von HTML-Konstrukten innerhalb einer Applikation.

Service: Viele Applikationen brauchen bzw. nutzen Daten, die sie von externen Quellen, wie z.B. Web Services, beziehen. Das Feature Service stellt alle benötigten Konstrukte zur Verfügung, um sich mit einem Web Service verbinden zu können und Daten von diesem zu empfangen.

Async: Da die Verarbeitung von Daten innerhalb einer Applikation auf dem mobilen Gerät und das anschließende Anzeigen der berechneten Ergebnisse manchmal etwas länger dauern kann, bietet das Feature Async die Möglichkeit, die Verarbeitung asynchron auszuführen. Bis das Ergebnis zur Verfügung steht, kann beispielsweise eine Ladeanzeige auf dem Ergebnisscreen eingeblendet werden.

OrderBy: Das Feature OrderBy stellt eine Filterfunktion dar. Damit können z.B. Ergebnislisten auf- bzw. absteigend sortiert werden.

Analog zum Abschnitt 5.1.1 werden die Stellen im Quellcode, die zu einem der oben beschriebenen Features gehören, mit Annotationen versehen. In Abb. 5.11 ist die Verteilung des Feature-Codes auf die verschiedenen Module der Sprachdefinition von MoBL zu sehen.

Für die gefundenen Features wird ebenfalls ein Feature-Modell und -Diagramm erstellt, das in Abb. 5.12 dargestellt ist. Aus dem Feature-Diagramm lässt sich ablesen, dass das Feature HTML obligatorisch und die anderen Features optional sind. HTML kann nicht optional gemacht werden, da wichtige Teile der Standardbibliothek, wie z.B. das Konstrukt in Listing 5.11, mit Hilfe von HTML-Code definiert sind und stets benötigt werden.

[2] Das MoBL-Projekt kann von der Seite https://github.com/mobl/mobl heruntergeladen und als Projekt in Eclipse importiert werden.

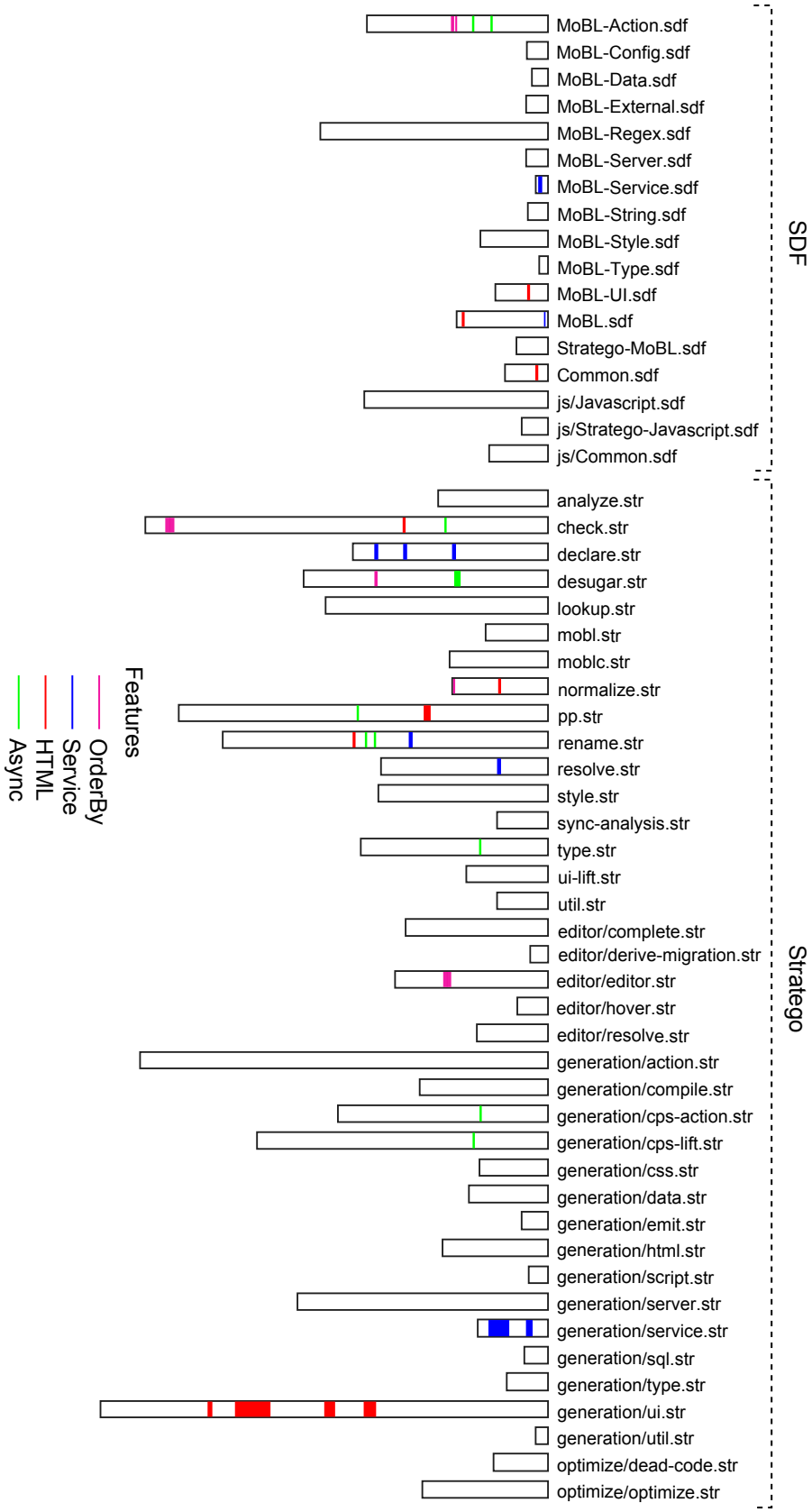

Abbildung 5.11: Lokalisation des Feature-Codes

```
1  control header(text : String, fixedPosition : Bool = false, onclick :
       Callback = null) {
2    <div class=headerStyle onclick=onclick style=fixedPosition ?
3    "position:fixed;" : null>
4      <div class=headerContainerStyle><div databind=text
5      class=headerTextStyle/></div>
6      elements()
7    </div>
8    ...
9    when(fixedPosition) {
10     <div id="hello" style="height: 2.9em;"/>
11   }
12 }
```

Listing 5.11: Definition des Konstrukts *header*

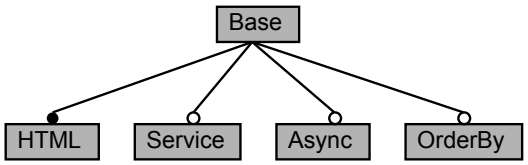

Abbildung 5.12: Feature-Diagramm der Sprache MoBL

5.2.2 Feature–Dekomposition

Entsprechend dem Vorgehen bei der Dekomposition von Features der Ausdruckssprache (siehe Abschnitt 5.1.2), wird auch der Quellcode der Features von MoBL in Feature-Module verschoben. Dies soll noch einmal am Bespiel des Features Async verdeutlicht werden.

Aus der Abb. 5.11 lässt sich ablesen, dass sich der Feature-Code von Async auf die folgenden Syntax-, Typ- und Semantikmodule der Sprachdefinition von MoBL erstreckt:

- MoBL-Action.sdf
- cps-action.str
- cps-lift.str
- check.str
- desugar.str
- pp.str
- rename.str
- type.str

Es werden entsprechende Module mit demselben Namen erstellt und der Quellcode in die neuen Module kopiert (vgl. Abschnitt 5.1.2). Daraus ergibt sich die in Abb. 5.13 gezeigte Ordnerstruktur für das Feature Async und die in Abb. 5.14

gezeigte Ordnerstruktur für alle Features von MoBL. Letztere bildet die Grundlage für die Komposition der Features in FeatureHouse, die im nächsten Abschnitt genauer untersucht wird.

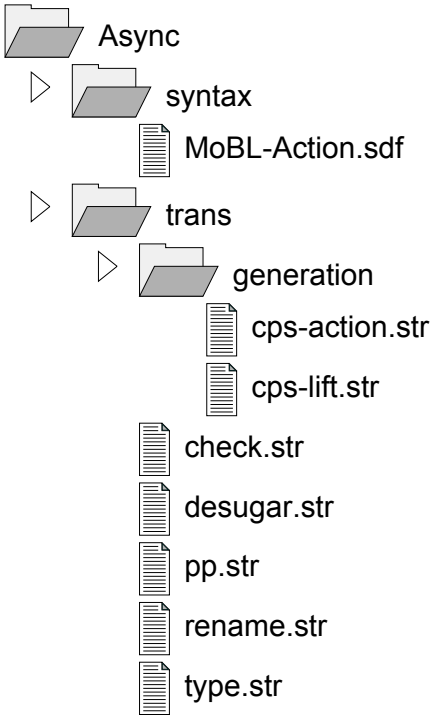

Abbildung 5.13: Ordnerstruktur für das Feature Async

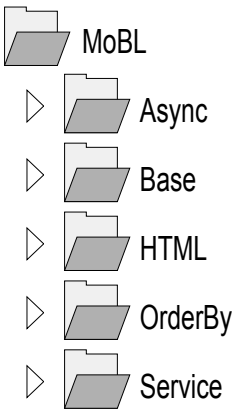

Abbildung 5.14: Ordnerstruktur aller MoBL Features

5.2.3 Feature–Komposition mit FeatureHouse

Für die Komposition der Features mit Hilfe von FeatureHouse, wird auch für MoBL zunächst eine Feature-Selektion durchgeführt und anschließend eine Konfigurationsdatei angelegt:

```
1 MoBLComp.features:
2   HTML
3   Base
4   Async
5   Service
6   OrderBy
```

Listing 5.12: Feature-Selektion zur Erstellung einer MoBL Variante

```
1 fstcomp MoBL.launch:
2   ...
3     <stringAttribute key="org.eclipse.jdt.launching.PROGRAM_ARGUMENTS"
4       value="--expression
5       examples/Exprlang/ExprlangComp.features"/>
6   ...
```

Listing 5.13: Konfigurationsdatei für MoBL

Durchführung der Feature-Selektion: Dafür wird die Datei *MoBLComp.features* benötigt, die in Listing 5.12 dargestellt ist. Sie enthält die Features für die zu erstellende Variante von MoBL (als Beispiel werden hier alle optionalen Features selektiert, was der Variante 8 in Tabelle 5.2 entspricht).

Anlage der Konfigurationsdatei: Analog zur Konfigurationsdatei für die Ausdruckssprache (vgl. Listing 5.10), wird für MoBL ebenfalls eine entsprechende Datei *fstcomp MoBL.launch* angelegt. Die beiden Dateien unterscheiden sich lediglich in dem in Listing 5.13 gezeigten Attribut.

Für die mit FeatureHouse generierte Variante von MoBL kann mit Hilfe von Eclipse ein Plug-In erstellt werden, das in einer neuen Eclipse-Instanz installiert werden kann. Mit diesem Plug-In können jedoch nur solche MoBL-Applikationen entwickelt werden, die von den enthaltenen Features unterstützt werden.

In Abb. 5.15 ist der Quelltext einer Applikation zu sehen, die mit einer MoBL-Variante entwickelt wurde, welche das Feature Async nicht unterstützt. Daher ist das Konstrukt *async(...)* unbekannt und führt zu einem Fehler, der im Editor – rot markiert – angezeigt wird.

Für die Erstellung eines solchen Plug-Ins sind folgende Schritte notwendig[3]:

1. Es werden ein sogenanntes *plugin feature project* und *update site project* benötigt, die ebenfalls als Repository[4] heruntergeladen werden können.

2. Anhand der MoBL-Variante und den beiden Projekten, wird ein Plug-In generiert, das lokal gespeichert oder auf einen Webserver hochgeladen werden kann.

[3] Für eine ausführlichere Beschreibung siehe How To auf der beigefügten DVD.

[4] Die beiden Projekte können von der Seite https://github.com/zefhemel/mobl-plugin.git heruntergeladen und in Eclipse als Projekte importiert werden.

```
58 // UI
59 screen root() {
60   header("Twitter trends")
61   var trends = async(Twitter.trends())
62   whenLoaded(trends) {
63     group {
64       list(topic in trends) {
65         item(onclick={ search(topic.name); }) {
66           label(topic.name)
67         }
68       }
69     }
70   }
71 }
72
73 screen search(query : String) {
74   header(query) {
75     backButton()
76   }
77   var results = async(Twitter.search(query))
78   whenLoaded(results) {
79     list(tweet in results) {
80       block(tweetStyle) {
81         image(tweet.profile_image_url,
82               style=tweetIconStyle)
83         <b>label(tweet.from_user) ": "</b>
84         label(tweet.text)
85       }
86     }
87   }
88 }
```

Abbildung 5.15: MoBL-Applikationen ohne Feature Async

3. Den letzten Schritt bildet die Installation einer neuen Eclipse-Instanz und des im 2. Schritt erstellten Plug-Ins, über den *Software-Update-Mechanismus* von Eclipse.

In Tabelle 5.2 sind alle möglichen Varianten von Sprachdefinitionen für MoBL dargestellt, die mit Hilfe der gefundenen Features erstellt werden können. Für jede Variante sind die Schritte 1-3 durchgeführt worden und in der letzten Spalte der Tabelle ist die Größe des erstellten MoBL-Plug-Ins aufgeführt (die Größe bezieht sich dabei auf die Java Archive (JAR)-Datei, die für die MoBL-Variante im Rahmen des Plug-Ins erstellt wird).

Die Spalte LOC enthält wiederum nur die Anzahl der Zeilen der SDF- bzw. Stratego-Module und die dritte Spalte die Anzahl der Semantik- und Typregeln einer Variante von MoBL.

Variante	**#Features**	**#Regeln**	**#LOC**	**Footprint (in Bytes)**
Variante_1	Base, HTML	976	5865	25.055.902
Variante_2	Base, HTML, Async	985	5936	25.105.105
Variante_3	Base, HTML, Service	985	5946	25.135.604
Variante_4	Base, HTML, OrderBy	986	5924	25.111.345
Variante_5	Base, HTML, Async, Service	994	6017	25.177.297
Variante_6	Base, HTML, Async, OrderBy	995	5995	25.157.797
Variante_7	Base, HTML, Service, OrderBy	995	6005	25.190.444
Variante_8	Base, HTML, Async, Service, OrderBy	1004	6076	25.235.756

Tabelle 5.2: Varianten von Sprachdefinitionen: MoBL

Kapitel 6

Diskussion und Zusammenfassung

Dieses Kapitel soll dazu dienen, die wichtigsten Resultate dieser Masterarbeit noch einmal kurz zusammenzufassen. Dabei wird vor allem auf die aus den beiden Fallstudien gewonnenen Ergebnisse eingegangen.

Ausgangspunkt dieser Masterarbeit war die folgende, zentrale Fragestellung:

> Kann die Entwicklung von Programmiersprachen und dazugehörigen Werkzeugen, wie z.B. Parser, Pretty Printer oder Typsystem, in einen Feature-orientierten Ansatz zerlegt werden?

Für die Definition einer Programmiersprache werden zunächst mehrere Bestandteile benötigt. Dazu gehören

- eine formale Definition der Syntax,
- die Definition des Typsystems und
- eine formale Definition der Semantik.

Bei der Beschreibung der Syntax wurde der SDF genutzt. Dieser Formalismus erlaubt die Definition von modularen Grammatiken. Dadurch können Grammatiken kombiniert und wiederverwendet werden. Dies vereinfacht die Handhabung von eingebetteten Sprachen oder verschiedenen Dialekten einer Sprache. Desweiteren können mit SDF lexikalische und kontextfreie Syntax gemeinsam definiert werden.

Für die Definition von Typ- und Semantikregeln wurde die Sprache Stratego genutzt, die Teil des Stratego/XT Frameworks ist. Stratego ist eine sehr effiziente domänenspezifische Sprache, mit der Programmtransformationen in einer strukturierten und robusten Art und Weise durchgeführt werden können. Dies geschieht auf Grundlage von Termen, Rewrite Regeln und Rewrite Strategien, die besonders gut geeignet für die Transformation und Traversierung von Baumstrukturen sind.

Warum die Entwicklung von Sprachdefinitionen und Sprachwerkzeugen in einem Feature-orientierte Ansatz erfolgen soll ist durch mehrere Aspekte motiviert. Dazu gehören unter anderem folgende:

- Die Sprachfeatures besitzen eine hohe Wiederverwendbarkeit.
- Die Sprachfeatures bieten eine hohe Variabilität und Variantenvielfalt.

- Die unterschiedliche Zusammensetzung von Sprachfeatures ermöglicht das Experimentieren mit Sprachvarianten, bis eine optimale Auswahl für eine spezielle Anwendung gefunden wird.
- Durch eine schrittweise Entwicklung von Sprachvarianten, können immer wieder neue Features hinzu gefügt werden.
- Mit Hilfe von FeatureHouse können Sprachfeatures selektiert und miteinander komponiert werden und anschließend können dafür Sprachwerkzeuge automatisch erzeugt werden.

Dass die gefundenen Features, z.B. für die Sprache MoBL (siehe Abschnitt 5.2), eine hohe Wiederverwendbarkeit, Variabilität und Variantenvielfalt besitzen, zeigt sich darin, dass die Features in mehreren MoBL-Varianten verwendet werden können (vgl. Tabelle 5.2). Durch eine unterschiedliche Selektion der Features konnten insgesamt acht verschiedene Varianten von MoBL – in Form von Plug-Ins für die Entwicklungsumgebung Eclipse – erstellt werden.

Für die Selektion und Komposition der Features wurde FeatureHouse verwendet. Das Framework FeatureHouse bietet die Möglichkeit der Komposition von Features, basierend auf einem sprachenunabhängigen Modell für Softwareartefakte. Dabei kann ein Artefakt jegliche Art von Information sein, die Teil einer Sprache ist. Dazu gehören Codestücke, wie z.B. Pakete, Klassen und Methoden, oder unterstützende Dokumente, wie z.B. Modelle und Dokumentationen.

Hier werden solche Artefakte durch die SDF- und Stratego-Dateien repräsentiert. Damit FeatureHouse mit diesen Softwareartefakten umgehen kann, mussten u.a. zwei entsprechende Grammatiken für SDF und Stratego in FeatureHouse integriert werden. Für die Erstellung der oben erwähnten Plug-Ins bzw. Varianten wurde die Spoofax Workbench genutzt.

Mit Hilfe der generierten Varianten können entsprechende Applikationen erstellt werden, die dann auch nur auf die Sprachkonstrukte zurückgreifen können, die von den enthaltenen Features zur Verfügung gestellt werden (siehe Abb. 5.15). Dadurch können unterschiedliche Anwendungsbereiche abgedeckt werden und maßgeschneiderte Anwendungen erstellt werden.

Auch die Größe der Sprachvariante kann reduziert werden, wenn nicht alle Sprachfeatures benötigt werden, wie aus der Spalte *Footprint in Bytes* der Tabelle 5.2 ersichtlich ist. Die MoBL-Varianten unterscheidet sich zwar nur marginal, was deren Größe betrifft, aber für andere Sprachen können diese Unterschiede durchaus signifikanter sein, je nach Größe der gefundenen Features.

Die beiden Fallstudien sollen als Grundlage für eine weiterführende Forschung in diesem Bereich dienen und sie haben gezeigt, dass der Ansatz des Feature-Oriented Language Engineerings mit Hilfe von FeatureHouse und der Spoofax Workbench, die den SDF und die Sprache Stratego beinhaltet, gut umgesetzt werden kann.

Kapitel 7

Enthaltene Software

Dieser Ausarbeitung ist eine DVD beigefügt, die folgenden Inhalt hat:

- Quelltext der Ausdruckssprache
- Quelltext der Sprache MoBL: Für jede Variante ein entsprechender Ordner bestehend aus:
  - dem MoBL-Projekt,
  - dem generierten Plug-In und
  - einer Eclipseinstanz, in der das Plug-In installiert ist.
- FeatureHouse Erweiterungen:
  - Grammatik für SDF, die in *FeatureHouse* integriert wurde
  - Grammatik für Stratego, die in *FeatureHouse* integriert wurde
- How To: hier wird beschrieben, wie eine Variante von MoBL erstellt werden kann (vgl. Tabelle 5.2)
- Dargestellte Abbildungen
- Digitale Version dieser Ausarbeitung im PDF-Format

Kapitel 8

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich diese Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

_____

Daniel Rolf

Glossar

Ambiguität
Von Mehrdeutigkeit oder Ambiguität (lat. ambo: beide, ambiguus: doppeldeutig, mehrdeutig, uneindeutig) spricht man, wenn ein Zeichen mehrere Bedeutungen hat.[1].

Assemblersprache
Eine Assemblersprache (oft abgekürzt als ASM bzw. asm) ist eine spezielle Programmiersprache, welche die Maschinensprache einer spezifischen Prozessorarchitektur in einer für den Menschen lesbaren Form repräsentiert. Jede Computerarchitektur hat folglich ihre eigene Assemblersprache.[2]

bottomup
Engl. für „von unten nach oben“.

Bytecode
Der Bytecode ist eine Sammlung von Befehlen für eine virtuelle Maschine und ist i.d.R. unabhängig von realer Hardware, d.h. er kann auch auf einem anderen Rechner, sogar über ein Netzwerk, interpretiert werden.[3]

Code Folding
Code Folding oder auch Code Faltung, bezeichnet eine Funktion in Editoren, vor allem in modernen IDEs, um logisch zusammengehörende Textabschnitte in sogenannte Folds bzw. Falten zu gruppieren. Die Abschnitte können dann einfach ein- und ausblenden werden.

Compiler
Wird oft auch Übersetzer oder Kompilierer genannt und ist ein Computerprogramm, das ein in einer Quellsprache geschriebenes Programm in ein semantisch äquivalentes Programm einer Zielsprache umwandelt. Meist ist die Zielsprache Assemblersprache, Bytecode oder Maschinensprache.[4]

Disambiguierung
Auflösung von sprachlicher Mehrdeutigkeit.

[1] Quelle: http://de.wikipedia.org/wiki/Mehrdeutigkeit
[2] Quelle: http://de.wikipedia.org/wiki/Assemblersprache
[3] Quelle: http://de.wikipedia.org/wiki/Bytecode
[4] Quelle: http://de.wikipedia.org/wiki/Compiler

Drei-Adress-Code

Drei-Adress-Codes haben i.d.R. die Form

$$t_1 = t_2 \; Operator \; t_3$$

und verfügen über höchstens einen Operator auf der rechten Seite. Der berechnete Wert wird in t_1 gespeichert.

Feature

Ein Feature ist eine semantisch zusammenhängende Einheit, die einer Anforderung an ein Programm entspricht. Es kann als eine Abstraktion einer Funktionalität gesehen werden und dient zur Spezifikation von Programmen. Weiterhin repräsentiert es Gemeinsamkeiten oder Unterschiede eines Programms und bietet eine Konfigurationsmöglichkeit.[5]

Interpreter

Ein Interpreter ist ein Computerprogramm, das einen Programm-Quellcode nicht in eine auf dem System direkt ausführbare Datei umwandelt, sondern den Quellcode einliest, analysiert und ausführt. Die Analyse des Quellcodes erfolgt also zur Laufzeit des Programms.[6]

kontextfreie Grammatik

Ist eine Grammatik, die nur solche Ersetzungsregeln enthält, bei denen immer genau ein Nichtterminal auf eine beliebig lange Folge von Nichtterminalen und Terminalen abgeleitet wird. Dabei steht das zu ersetzende Nichtterminal allein (daher „kontextfrei“) auf der linken Seite der Ersetzungsregel.[7]

Nichtterminal

Ein Nichtterminal einer formalen Grammatik ist ein Symbol, das nur in Zwischenschritten einer Ableitung vorkommt. Das Nichtterminal wird durch die Anwendung von Grammatikregeln nach und nach ersetzt, bis nur noch Terminale vorhanden sind.[8]

Normalform

Ein Term ist eine Normalform, wenn er nicht weiter evaluiert werden kann.[9]

Parser

Ein Parser ist ein Tool, das einen String (repräsentiert ein Programm) als Eingabe erhält und daraus eine Ausgabe (ein Baum, der das Programm in einer strukturierten Form darstellt) erstellt.

[5] Quelle: [Ape08]
[6] Quelle: http://de.wikipedia.org/wiki/Interpreter
[7] Quelle: http://de.wikipedia.org/wiki/Kontextfreie_Grammatik
[8] Quelle: http://de.wikipedia.org/wiki/Nichtterminal
[9] Quelle: [Ape10]

Refaktorisierung

Refaktorisierung ist die manuelle oder automatisierte Strukturänderung von Programmquelltexten unter Beibehaltung des externen Programmverhaltens.[10]

Repository

Ein Repository (engl. für Lager, Depot, Quellen oder Archiv, Plural Repositories), auch Repositorium, ist ein verwaltetes Verzeichnis zur Speicherung und Beschreibung von digitalen Objekten. Bei den verwalteten Objekten kann es sich beispielsweise um Programme (Software-Repository) handeln.[11]

Rewrite Regel

Eine Rewrite Regel oder auch Umschreibungsregel, hat die Form

$$Lab : l \text{ -> } r \text{ where } s$$

wobei *Lab* die Bezeichnung oder der Name der Regel ist, l die linke bzw. r die rechte Seite des anzupassenden Terms und s die Strategie, die als Bedingung fungiert. Bei der Anwendung einer Regel *Lab* auf einen Term t wird zuerst überprüft, ob l mit dem Term übereinstimmt, unter Berücksichtigung der Bedingung s, und dann wird der Term t nach r überführt.[12]

Rewriting Strategie

Eine Rewriting Strategie ist ein Algorithmus zur Transformation eines Terms bezüglich einer Menge von Rewrite Regeln. Einige Rewrite Strategien sind Normalisierungs-Strategien, die z.B. einen Term in Normalform überführen (umschreiben).[13]

Tablet-PC

Ein Tablet-PC ist ein tragbarer Computer, der ohne Tastatur benutzt werden kann. Die Bedienung erfolgt per Eingabestift und teilweise auch per Finger direkt auf einem berührungsempfindlichen Bildschirm.[14]

Terminal

Ein Terminal einer formalen Grammatik ist ein Symbol, das einzeln nicht weiter durch eine Produktionsregel ersetzt werden kann.[15]

Transitionsfunktion

Eine Transitionsfunktion definiert, wie sich der Zustand einer abstrakten Maschine über die Zeit ändert.[16]

[10] Quelle: [Ape08]
[11] Quelle: http://de.wikipedia.org/wiki/Repository
[12] Quelle: http://strategoxt.org/Stratego/RewriteRule
[13] Quelle: http://strategoxt.org/Stratego/RewritingStrategy
[14] Quelle: http://de.wikipedia.org/wiki/Tablet_PC
[15] Quelle: http://de.wikipedia.org/wiki/Terminalsymbol
[16] Quelle: [Ape10]

Typüberprüfung

Die Typüberprüfung ist eine effektive und etablierte Technik, um Inkonsistenzen in Programmen abzufangen. Sie kann verwendet werden, um Fehler herauszufinden, z.B. wenn eine Operation auf den falschen Typ von Objekt angewendet wird oder wenn die an eine Prozedur übergebenen Parameter nicht mit der Signatur der Prozedur übereinstimmen.[17]

Wasserfallmodell

Das Wasserfallmodell ist ein lineares (nicht iteratives) Vorgehensmodell, insbesondere für die Softwareentwicklung, das in Phasen organisiert wird. Dabei gehen die Phasenergebnisse, wie bei einem Wasserfall, immer als bindende Vorgaben in die nächsttiefere Phase ein. Im Wasserfallmodell hat jede Phase vordefinierte Start- und Endpunkte mit eindeutig definierten Ergebnissen.[18]

[17] Quelle: [ALSU08]
[18] Quelle: http://de.wikipedia.org/wiki/Wasserfallmodell

Acronyms

APIs	Application Programming Interfaces
AST	Abstract Syntax Tree
ATerm	Annotated Term
DSLs	Domain Specific Languages
EBNF	Erweiterte Backus-Naur-Form
FOP	Feature-Oriented Programming
FOSE	Feature-Oriented Software Engineering
FST	Feature Structure Tree
HTML	Hypertext Markup Language
IDEs	Integrated Development Environments
JAR	Java Archive
LOC	Lines of Code
SDF	Syntax Definition Formalism
SEI	Software Engineering Institute
SPL	Software Product Line
XML	Extensible Markup Language

Literaturverzeichnis

[AKL09] Apel, Sven ; Kästner, Christian ; Lengauer, Christian: FEATUREHOUSE: Language-independent, automated software composition. In: *ICSE*, IEEE, 2009. – ISBN 978–1–4244–3452–7, 221–231

[AL08] Apel, Sven ; Lengauer, Christian: Superimposition: A Language-Independent Approach to Software Composition. In: Pautasso, Cesare (Hrsg.) ; Tanter, Éric (Hrsg.): *Software Composition, 7th International Symposium, SC 2008, Budapest, Hungary, March 29-30, 2008. Proceedings* Bd. 4954, Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–78788–4, 20–35

[ALMK08] Apel, Sven ; Lengauer, Christian ; Möller, Bernhard ; Kästner, Christian: An Algebra for Features and Feature Composition. In: Meseguer, José (Hrsg.) ; Rosu, Grigore (Hrsg.): *Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, Proceedings* Bd. 5140, Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–79979–5, 36–50

[ALSU08] Aho, Alfred V. ; Lam, Monica S. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compiler: Prinzipien, Techniken und Werkzeuge*. 2., aktualisierte Auflage. Martin-Kollar-Straße 10-12, D-81829 München : Pearson Studium, 2008. – ISBN 978–3–8273–7097–6. – Kapitel 1, Einleitung

[Ape08] Apel, Dr.-Ing. S.: *Moderne Programmier Paradigmen*. Vorlesung Universität Passau – Fakultät für Informatik und Mathematik, Wintersemester 2007/2008

[Ape10] Apel, Dr.-Ing. S.: *Typen und Programmiersprachen*. Vorlesung Universität Passau – Fakultät für Informatik und Mathematik, Wintersemester 2009/2010

[Bat05] Batory, Don: Feature Models, Grammars, and Propositional Formulas / The University of Texas at Austin, Department of Computer Sciences. Version: Apr. 11 2005. ftp://ftp.cs.utexas.edu/pub/techreports/tr05-14.pdf. 2005 (CS-TR-05-14). – Forschungsbericht. – Fri, 28 Sep 107 13:03:38 GMT

[BF88] Bougé, Luc ; Francez, Nissim: A Compositional Approach to Superimposition. In: *POPL*, 1988, 240–249

[BKV07] Brand, Mark van d. ; Klint, Paul ; Vinju, Jurgen: *The Syntax Definition Formalism SDF.* http://homepages.cwi.nl/~daybuild/daily-books/syntax/2-sdf/sdf.html. Version: Oct. 2007

[BKVV08] Bravenboer, Martin ; Kalleberg, Karl T. ; Vermaas, Rob ; Visser, Eelco: Stratego/XT 0.17. A Language and Toolset for Program Transformation. In: *Science of Computer Programming* 72 (2008), jun, Nr. 1-2, 52–70. http://dx.doi.org/10.1016/j.scico.2007.11.003. – ISSN 0167–6423. – Special issue on experimental software and toolkits

[BSR04] Batory, Don S. ; Sarvela, Jacob N. ; Rauschmayer, Axel: Scaling Step-Wise Refinement. In: *IEEE Transactions on Software Engineering* 30 (2004), jun, Nr. 6, 355–371. http://doi.ieeecomputersociety.org/10.1109/TSE.2004.23

[DKT93] Deursen, A. V. ; Klint, P. ; Tip, F.: Origin Tracking. In: *JSC* 15 (1993), may & jun, Nr. 5&6, S. 523–546

[Fea] *The FeatureBNF Grammar Specification Language (formerly gCIDE).* http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/gcidegrammar.html

[Kun08] Kunert, Andreas: *LR(k)-Analyse für Pragmatiker.* Humboldt-Universität zu Berlin – Institut für Informatik, April 2008

[KV10] Kats, Lennart C. L. ; Visser, Eelco: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Cook, William R. (Hrsg.) ; Clarke, Siobhán (Hrsg.) ; Rinard, Martin C. (Hrsg.): *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010.* Reno/Tahoe, Nevada : ACM, 2010. – ISBN 978–1–4503–0203–6, S. 444–463. – (best student paper award)

[MH03] McDirmid, Sean ; Hsieh, Wilson C.: Aspect-oriented programming with Jiazzi. In: *AOSD*, 2003, 70–79

[MO] Mezini, Mira ; Ostermann, Klaus: Conquering aspects with Caesar. In: *AOSD 03*, S. 90–99

[MoB11] *MoBL: The new language of the mobile web.* http://www.mobl-lang.org/. Version: 2011

[OH92] Ossher, Harold ; Harrison, William H.: Combination of Inheritance Hierarchies. In: *OOPSLA*, 1992, S. 25–40

[Pie02] Pierce, Benjamin: *Types and Programming Languages.* The MIT Press, 2002 http://www.cis.upenn.edu/~bcpierce/tapl/index.html

[Pre97] Prehofer, Christian: Feature-Oriented Programming: A Fresh Look at Objects. In: *ECOOP*, 1997, 419–443

[SEI12] *Software Engineering Institute of Carnegie Mellon University*. http://www.sei.cmu.edu/productlines/. Version: 2012

[Str10] *Stratego/XT*. http://strategoxt.org/. Version: May 2010

[TOHS99] Tarr, Peri ; Ossher, Harold ; Harrison, William ; Sutton, Jr, Stanley M.: N Degrees of Separation: Multi-dimensional Separation of Concerns. In: *Proceedings of ICSE '99*. Los Angeles CA, USA, 1999, S. 107–119

[Tou10] *A Tour of Spoofax/IMP*. http://strategoxt.org/Spoofax/Tour. Version: Oct. 2010

[Vin07] Vinju, Jurgen: *SDF Disambiguation Medikit for Programming Languages*. http://homepages.cwi.nl/~daybuild/daily-books/syntax/3-sdf-disambiguation/sdf-disambiguation.html. Version: Oct. 2007