Fakultät für Informatik und Mathematik
Universität Passau

**Bachelorarbeit**
**Informatik**

# Komposition von Features durch Quantifizierung

Boxleitner Stefan
boxleitn@fim.uni-passau.de

10. November 2008

Betreuer
Dr.-Ing. Sven Apel
apel@uni-passau.de

# Inhaltsverzeichnis

# 1 Zusammenfassung

*Features* sind ein Abstraktionsmodell, mit dem sich *Software Produktlinien* effizient umsetzten lassen.

In dieser Arbeit geht es um die Erforschung der Umsetzbarkeit eines theoretischen Konzepts der Featurekomposition, der *Quantifizierung* und dem damit eng verbundenen Konzept der *Modifikationen*. Die Idee und die formalen Grundlagen dafür stammen aus [ALMK08].

Zunächst werden dabei theoretische Grundlagen vermittelt die sowohl eine Verständnis für die umzusetzenden Konzepte als auch einen Einblick in deren algebraische Eigenschaften vermitteln.

Des Weiteren wird die Umsetzung als Erweiterung des *FST Composers* aufgeführt. Dabei werden Problemstellungen, Erweiterbarkeit und der Aspekt der Konsistenz mit den theoretischen Konzepten betrachtet.

Für eine praxisnahe Bewertung der Implementierung und im Allgemeinen des Konzepts der Quantifizierung wurden mehrere Fallstudien durchgeführt. Eine Zusammenfassung der Ergebnisse befindet sich am Ende dieser Arbeit.

# 2 Feature-Orientierte Programmierung

Die Vergangenheit hat gezeigt, dass das Paradigma der *objektorientierten Programmierung* durchaus seine Schwächen besitzt. Eine genaue Beschreibung der Probleme findet man z.B. in [Ape08]. Die *Feature Orientierte Programmierung* kann durch neue Konzepte einen Teil dieser Probleme lösen. Die Die folgenden Ausführungen bieten einen kleinen Einblick in dieses Programmierparadigma.

## 2.1 Produktlinien

Der Grundgedanke hinter diesem Begriff ist die Erzeugung maßgeschneiderter, an Anwendungsfälle angepasster Produkte aus einem Pool von vorhandenen Funktionalitäten, die in irgendeiner Weise miteinander verwandt sind. Funktionalitäten sind durch logische Einheiten gekapselt.

Dieser Begriff stammt ursprünglich nicht aus dem Software Engineering [Ape08]. Produktlinien gibt es in fast allen Wirtschaftsbereichen, z.B. auch in der Automobilindustrie. Viele PKW-Serien sind die durch den Käufer von der Motorleistung bis hin zur Farbe der Sitzbezüge anpassbar, siehe Abb 2.1.

Einer der Hauptvorteile hierbei ist die hohe Wiederverwendbarkeit der Funktionalitäten innerhalb der Produktfamilie, d.h. man kann eine Implementation einer Funktionalität i.d.R. für alle Produkte verwenden.

Ein weiteres Beispiel: eine Produktfamilie *Taschenrechner* besitzt die Funktionalitäten *Basis, Grundrechenarten, Analysis*. Für einfache Berechnungen, z.B. in der Grundschule reichen die Grundrechenarten noch aus. Als Plattform wählt man günstige Hardware die für diese Zwecke völlig ausreichend ist. Für analytische Berechnungen reichen jedoch Speicher und Rechenleistung der günstigen Hardware nicht mehr aus, also wird eine bessere teurere Hardware verwendet. Der Vorteil bei Produktlinien ist hierbei die Wiederverwendbarkeit der Funktionalitäten: Die Basis und die Grundrechenarten müssen für die leistungsfähige Plattform nicht neu integriert werden sondern können vom einfachen Modell einfach übernommen werden.

Eine *Software Produktlinie* wird in von *Software Engineering Institute, Carnegie Mellon University* in [Ape08] definiert als

> A software product line [...] is a set of softwareintensive 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.

Die Verwandtschaft der Funktionalitäten wird hier etwas konkretisiert als gemeinsamer

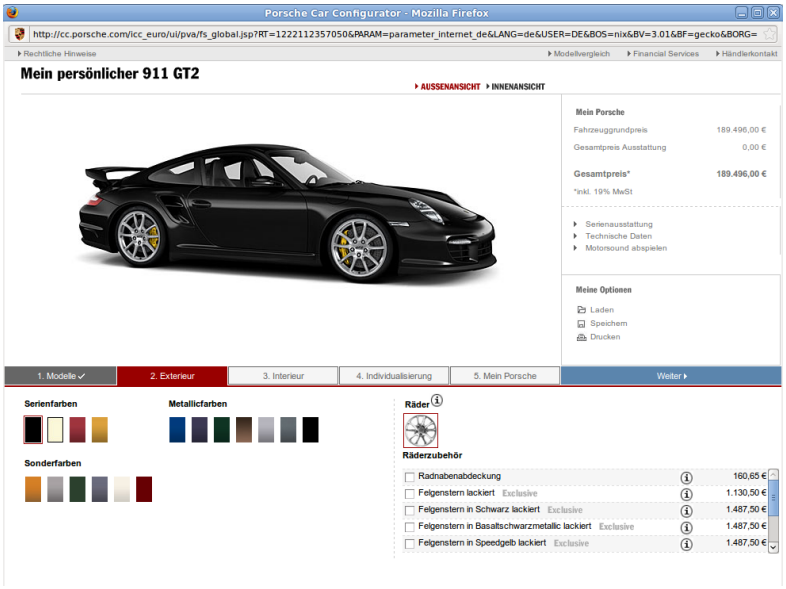


Abbildung 2.1: Produktlinie in der Automobilindustrie
[Quelle: porsche.de]

Zweck, der der Erfüllung der Anforderungen eines bestimmten Marktsegments oder einer bestimmten Aufgabe dient. Neu in dieser Definition ist der Begriff *Feature*.

## 2.2 Feature

Der Begriff Feature wird in dem Paradigma der *Feature-Orientierten Programmierung (FOP)* als Inkrement an Funktionalität bezeichnet. Features sind Abstraktionen der Funktionalitäten, d.h. ein Feature kapselt eine bestimmte Funktionalität.

### 2.2.1 Basis Feature

Eine Möglichkeit ist es, Features durch Strukturen aus *Software Artefakten*[1] umzusetzen. Man spricht hier von *Basis Features (Englisch: basic features)*.

*Feature structure trees (FST)* bilden eine solche Struktur [ALMK08]. Abb. 2.2 zeigt ein Beispiel für einen FST. Man spricht hier bei Inneren Knoten von *Nichtterminalen*, bei Blättern von *Terminalen*.

Alle Knoten des FST besitzen einen Namen und einen Typ, der den Kompositionswerkzeugen durch Typüberprüfungen die Arbeit erleichtert. Terminale haben zusätzlich noch einen Inhalt. Dieser kann wie hier bei dieser Java Klasse aus Quelltext bestehen, es könnte sich aber auch um ein binär hinterlegtes Artefakt handeln, z.B. eine jpeg-Datei.

[1] Software Artefakte sind kontextabhängige Teilstücke einer Software, wie z.B. einzelne Java-Klassen oder Teilstücke einer Betriebsanleitung als html-Datei

```
1 package util;
2 class Calc {
3   int e0=0, e1=0, e2=0;
4   void enter(int v) { e2=e1; e1=e0; e0=v; }
5   void clear() { e0=e1=e2=0; }
6   String top() { return String.valueOf(e0); }
7 }
```

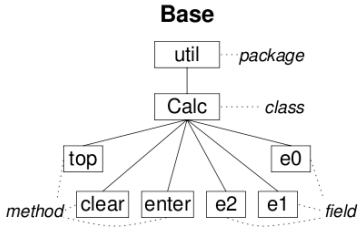


Abbildung 2.2: Feature als FST
[Quelle: [ALMK08]]

Man sieht bereits hier, dass dieser Ansatz im Allgemeinen sprachunabhängig ist, d.h. man kann Artefakte beliebiger Herkunft in dieses Modell implementieren [AL08].

Der Baum erinnert an einen Syntax Baum der Java Grammatik [ALMK08]. In der Tat werden diese Grammatiken als Hilfsmittel zur Erzeugung der FSTs verwendet, was in 3.7 noch deutlicher wird.

Im Beispiel entsteht zwar der Eindruck, dass die Reihenfolge der Kinder eines Nichtterminals beliebig ist, das stimmt jedoch im Allgemeinen nicht: Da die Granularität bei der Erzeugung eines FST im Allgemeinen beliebig fein gewählt werden kann, wäre ein auch FST für Java denkbar, der als Blätter einzelne Programmzeilen abbildet, bei denen die Reihenfolge wichtig ist.

FSTs stehen in direktem Zusammenhang zu den entsprechenden Software Artefakten. Die Abbildung von Software Artefakten nach FST ist somit eine Bijektion. Dadurch ist auch nach mehrfachen Abbildungen in beide Richtungen eine dauerhafte Konsistenz garantiert.

Zur Erzeugung eines Endprodukts einer Produktlinie wird je nach Anforderung eine Teilmenge der Features ausgewählt und miteinander kombiniert. Features können an sich bereits ein valides Produkt bilden, oder erst in einer Komposition mit anderen Features. Auf die Anforderungen für Validität eines Produkts wird hier nicht näher eingegangen, kann aber z.B. in [Bat05] nachgelesen werden.

## 2.3 Feature Komposition

Feature Komposition wird im Folgenden als mathematischer binärer Operator $\bullet$ bezeichnet. Er bildet zwei Features wieder in den Raum der Features ab.

$$\bullet : F \times F \rightarrow F \tag{2.1}$$

$$p = f_n \bullet f_{n-1} \bullet \ldots \bullet f_2 \bullet f_1 \tag{2.2}$$

Die Reihenfolge der Feature Komposition ist wichtig, d.h. $\bullet$ ist nicht kommutativ. Die Komposition ist ausserdem idempotent, d.h. das mehrfache Vorkommen eines Features in einer Komposition soll das selbe Ergebnis liefern, wie das einmalige Vorkommen [ALMK08].

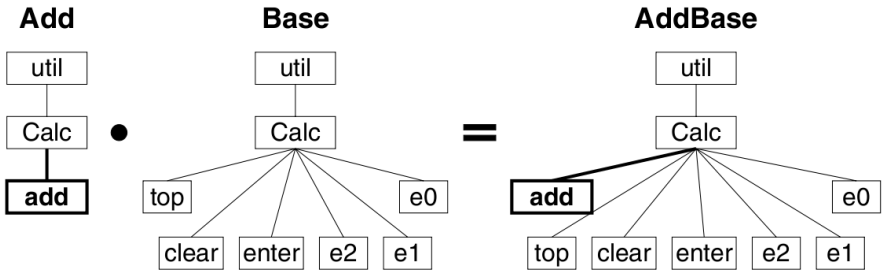


Abbildung 2.3: FST Superimposition
[Quelle: [ALMK08]]

### 2.3.1 Superimposition

Die *Superimposition* ist eine Ausprägung der oben erwähnte Operation •, d.h. die Komposition zweier FSTs auf Ebene der Software Artefakte.

Zwei FST Knoten werden dabei genau dann zu einem neuen Knoten verbunden, wenn folgende Voraussetzungen erfüllt sind:

- Ihre Eltern wurden bereits zu einem neuen Knoten verbunden oder beide sind Wurzelknoten
- wenn sie identische Namen und Typen besitzen

Der neue Knoten bekommt Name und Typ der komponierten Knoten.

Wenn zwei Knoten verbunden wurden, werden ihre Kinder wieder auf die o.g. Voraussetzungen überprüft, bis man bei den Blättern terminiert, die ja keine Kinder besitzen. Wenn die Voraussetzungen für die beiden Knoten nicht erfüllt sind, wird einer der Beiden in den Ergebnis-FST eingehängt. Abb. 2.3 zeigt ein Beispiel für FST Komposition [ALMK08].

#### Komposition von Terminalen

Terminale Knoten müssen gesondert behandelt werden, weil sie neben Name und Typ noch über Inhalt verfügen und die Komposition dieser Inhalte nicht trivial ist.

Man muss an der Stelle bedenken, dass man pro Domäne in der Regel mit einem Knotentyp nicht auskommen wird bzw. aufgrund erwünschter Flexibilität garnicht damit auskommen will. In 3.7 wird man sehen, dass z.B. für java in der bisherigen Implementierung separate terminale Knotentypen für Methoden, Importdeklarationen, etc existieren. Da sich diese Knotentypen i.d.R. semantisch mehr oder weniger stark unterscheiden, braucht man viele Kompositionsregeln. Beispielsweise kann man für Java-Importdeklarationen nicht dieselbe Regel anwenden, wie für Java-Methoden.

Im schlechtesten Fall muss für jeden terminalen Knotentyp eine solche Regel erstellt werden muss [AL08].

### 2.3.2 Quantifizierung

Bei der Superimposition werden die Ansatzpunkte der Änderungen, die während der Komposition durchgeführt werden durch die Position der Knoten im FST determiniert. Diese Technik ist auf einer Ebene die sich näher an der konkreten Implementierung befindet in der Umsetzung etwas unflexibel. Man denke an ein Feature, das ein und dasselbe Artefakt an mehreren Stellen des FST anhängen soll. In der Umsetzung durch Superimposition müsste dieses Artefakt mehrfach im FST des Features vorkommen, das die Änderungen durchführt. Aus softwaretechnischer Sicht ist das schlecht.

Deshalb ist es nur schlüssig, dass neben der Superimposititon noch weitere Ansätze existieren um Featurekomposition durchzuführen.

*Quantifizierung* ist der Prozess des Auffindens von Positionen an denen Änderungen an Features durchgeführt werden sollen.

Auf Ebene der FSTs bedeutet das, man kann einen oder *mehrere* Knoten festlegen, die verändert werden sollen. Ausgehend von der Menge der Knoten, verhält sich die weitere Komposition wie die der Superimposition, d.h. es werden die Kompositionsregeln (siehe oben) angewandt. Es ergibt sich eine enge Verwandtschaft der beiden Techniken [ALMK08].

#### Vorteile dieser Technik

Das Konzept der Quantifizierung ist nicht neu. Es gibt eine Reihe sehr weit entwickelter sprachspezifischer Umsetzungen, wie AspectJ[2] für Java. Die Vorteile dieser Technik kommen vor Allem in großen Software Projekten zum Zug, wo *cross-cutting concerns (CCC)* [LHA07] (deutsch: querschneidende Belange) auftreten.

Dabei handelt es sich um Funktionalitäten in der Software die über mehrere logische Kapselungseinheiten verteilt ist. Ein Grundprinzip des Software Engineering ist das Erreichen einer möglichst hohen Kohäsion, d.h. Kapselung von logisch zusammengehörigen Teilen mit möglichst wenig Code-Duplizierung. Dieses Prinzip wäre an dieser Stelle verletzt [Sne07] (siehe auch o.g. Beispiel).

Diese verteilten Funktionalitäten können nun gekapselt und vom Rest der Software entkoppelt werden um dann durch Quantifizierung an den passenden Stellen wieder eingefügt zu werden [LHA07].

### 2.3.3 Modifikationen

Modifikationen sind ein Konzept aus [ALMK08], das Quantifizierung als Teil einer alternativen Ausprägung der Feature-Komposition in ein Modell kapselt. Eine *Modifikation* $m$ ist eine Einheit bestehend aus

- einer *Auswahlfunktion* $t$ (von Englisch *(tree-)traversal*)
- der Spezifikation einer *Veränderungsfunktion* $c$ (von Enlisch *change*)

[2] http://www.eclipse.org/aspectj/

Formal betrachtet sind Modifikationen Tupel der Form

$$m = (t, c) \tag{2.3}$$

### Auswahlfunktion

In dem Artikel [ALMK08] wird die *Auswahlfunktion* durch Ausdrücke bestehend aus einer oder mehreren Knotenadressen eines FST festgelegt, die durch eine Reihe von nicht genauer spezifizierten syntaktischen Konstrukten ein möglichst mächtiges Mittel zur Selektierung einer beliebigen Teilmenge der Knoten eines FST bieten soll.

Bei der Quantifizierung wird der FST durchlaufen (daher *traversal*), bei einer Passung von Knoten und Ausdruck wird der Knoten in die Ergebnismenge übernommen. Die Auswahlfunktion ist demnach eine Abbildung

$$t : FSTNode \rightarrow FSTNode \tag{2.4}$$

Sei $n$ aus FSTNode

$$t(n) = \begin{cases} n & \text{, wenn n zu dem gegebenen Ausdruck passt} \\ \emptyset & \text{, wenn n nicht zu dem gegebenen Ausdruch passt} \end{cases} \tag{2.5}$$

Näheres zur Umsetzung findet man in 3.7.

### Veränderungsfunktion

Es gibt für unser Modell zwei Typen von Veränderungen:

- Einfügen von neuen Kindern mit den aus der Quantifizierung hervorgehenden Knoten (Voraussetzung: nichtterminal) als Eltern
- Alterieren bestehender terminalen Knotens

In Abb. 2.4 sieht man den Ablauf bei der Komposition einer Modifikation mit einem Basis-Feature: links die Quantifizierung, rechts die Anwendung der Veränderungsfunktion mit den Knoten aus der Ergebnismenge der Quantifizierung.

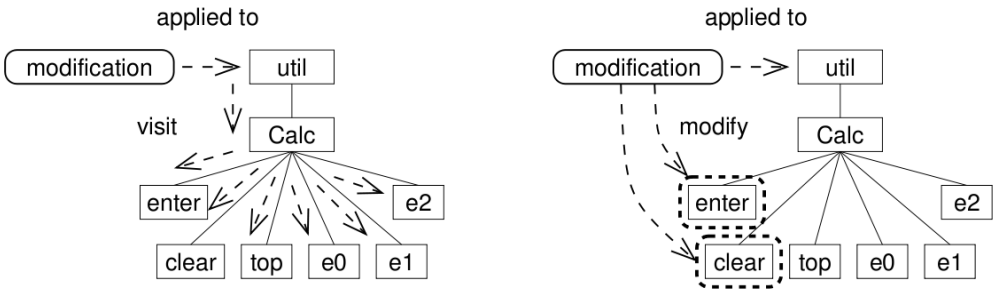


Abbildung 2.4: Quantifizierung und Veränderungsfunktion
[Quelle: [ALMK08]]

# 3 Feature Algebra

Die Feature Algebra aus [ALMK08] stellt noch weitere Formalismen für die Feature Komposition bereit. Die Abstraktion von der Umsetzung einzelner Operationen oder von Details einzelner Programmiersprachen ermöglicht eine sprachunabhängige Untersuchung der Konzepte. Dies ist nötig, da die Umsetzung ebenfalls möglichst sprachunabhängig sein sollte.

Für diese Arbeit sind die dabei gewonnenen Erkenntnisse über die eingeführten Operatoren, wie z.B. Assoziativität oder Idempotenz von besonderer Bedeutung. Sie bieten eine Basis für eine korrekte und konsistente Implementierung dieser Konzepte.

## 3.1 Introduktionen

*Atomare Introduktionen* sind prinzipiell nur ein anderer Begriff für FST-Knoten. Durch die Abbildung auf einen algebraischen Formalismus erhält man eine neue Schreibweise für die Knoten.

Die Notation der Introduktionen beinhaltet neben dem Namen noch den kompletten Pfad von der FST-Wurzel aus gesehen und sieht wie folgt aus:

```
<Name des Features>::<Pfad zum Knoten>.<Knotenname>
```

## 3.2 Introduktionssumme

Gleichzeitig wird der Begriff der *Introduktionssumme* $\oplus$ eingeführt, eine Abstraktion der Komposition terminaler Knoten.

$$\oplus : I \times I \to I \tag{3.1}$$

$$i_2 \oplus i_1 = i \tag{3.2}$$

Features können auch als Introduktionssumme aller enthaltenen atomarer Introduktionen formuliert werden. Durch die Pfadnotation kann daraus ohne Informationsverlust ein FST erzeugt werden. Das Feature Base als FST aus Abb. 2.3 wird abgebildet auf die Introduktionssumme

$$\begin{aligned} Base \quad = \quad & Base.util.Calc.enter \oplus Base.util.Calc.clear \\ & \oplus Base.util.Calc.top \oplus Base.util.Calc.e0 \\ & \oplus Base.util.Calc.e1 \oplus Base.util.Calc.e2 \\ & \oplus Base.util.Calc \oplus Base.util \end{aligned} \tag{3.3}$$

Die Reihenfolge der Summanden ist genau wie in den FSTs wichtig für die Semantik. An dieser Stelle wird der Begriff der Wohldefiniertheit für die Introduktionssumme eingeführt. Das bedeutet, dass für jeden Präfix-Teilpfad der in einer Introduktion vorkommt, auch ein Summand existiert der dem Teilpfad entspricht. Beispielsweise impliziert das Vorkommen von $Base :: util.calc$ das Vorkommen von $Base :: util$.

Unter dieser Voraussetzung ist die Abbildung von FST in den Raum der Introduktionen $I$ bijektiv.

Die Introduktionssumme ersetzt auf diese Weise auch die Komposition von ganzen Features: zwei Features werden komponiert, indem man ihre atomaren Introduktionen addiert. Umgesetzt wird das ganze dann ähnlich der Superimposition:

Zwei Introduktionen werden genau dann miteinander verbunden, wenn sie in Pfad und Knotennamen übereinstimmen. Alle anderen Introduktionen werden einfach in das Kompositum übernommen.

### 3.2.1 Algebraische Eigenschaften der Introduktionssumme

Die Introduktionssumme $\oplus$ über der Menge der Introduktionen bildet ein nicht-kommutatives idempotentes Monoid $(I, \oplus, \xi)$, wobei $\xi$ die leere Introduktion ist. Es gelten somit folgende Eigenschaften

- Assoziativität, d.h. so lange die absolute Reihenfolge in der Summe nicht vertauscht wird, können einzelne Summen mit zwei Operanden zu beliebiger Zeit addiert werden.
- Identität, d.h. es existiert ein neutrales Element $\xi$ was hier z.B. einem leeren FST entspricht
- Nicht-Kommutativität, d.h. die Reihenfolge der Summanden darf nicht beliebig vertauscht werden, da dies zu unterschiedlichen Ergebnissen führen kann. Man betrachte folgenden Fall: zwei kompatible Java-Methoden, repräsentiert als $i_1$ und $i_2$ sollen komponiert werden. Die Kompositionsregel ist einfaches Überschreiben des ersten Operanden durch den zweiten. $i_1 \oplus i_2$ ergibt somit als Ergebnis $i_2$, $i_2 \oplus i_1$ ergibt $i_1$
- Idempotenz, d.h. die mehrfache Addition derselben Introduktion erzeugt keine Veränderung zur einfachen Addition.

[ALMK08].

## 3.3 Modifikationen

Modifikationen (siehe 2.3.2) werden wie folgt in die Feature Algebra integriert:

Die Quantifizierung erfolgt nicht mehr als Durchlauf durch einen FST, sonder als Durchlauf über die Summanden einer Introduktionssumme. Die Abbildungsvorschrift

für die Auswahlfunktion lautet:

Sei $i$ ein Summand einer Introduktionssumme

$$t(i) = \begin{cases} i & \text{, wenn i zu dem gegebenen Ausdruck passt} \\ \xi & \text{, wenn i nicht zu dem gegebenen Ausdruch passt} \end{cases} \tag{3.4}$$

$t$ ist ausserdem linksdistributiv über der Introduktionssumme

$$q(i_n \oplus \ldots \oplus i_2 \oplus i_1) = q(i_n) \oplus \ldots \oplus q(i_2) \oplus q(i_1) \tag{3.5}$$

Die Veränderungsfunktion $c$ ist ebenfalls linksdistributiv über der Introduktionssumme

$$c(i_n \oplus \ldots \oplus i_2 \oplus i_1) = c(i_n) \oplus \ldots \oplus c(i_2) \oplus c(i_1) \tag{3.6}$$

$$c(i_j) = \tau_c(i_c, i_j) \oplus i_j \tag{3.7}$$

Die atomare Introduktion $i_c$ repräsentiert den Inhalt der Änderungsfunktion. In welcher Form diese Introduktion vorliegt, liegt in der Verantwortung des Umsetzers. Denkbar wäre jedoch z.B. ein Code-Fragment.

Die Funktion $\tau_c$ enthält die Ausprägung der Veränderungsfunktion, d.h. das Einfügen eines Knotens oder die Alterierung eines Knotens und bildet $i_c$ in Abhängigkeit von dem zu modifizierenden Knoten $i_j$ auf eine Introduktion ab, die dann per Introduktionssumme zu dem ursprünglichen $i_j$ addiert wird. Diese Addition verbietet bereits hier in der Algebra eine Modifikation die Knoten löschen kann. Die Abbildungsvorschrift von $\tau_c$ liegt auch hier in der Hand des Umsetzers [ALMK08].

Details zur Umsetzung finden sich in 4.2.2.

## 3.4 Modifikationsanwendung

Um das ganze ein wenig zu vereinfachen werden Quantifizierung und Veränderung zusammengefasst als *Anwendung einer Modifikation* ($\odot$). Es handelt sich dabei wieder um einen binären Operator, der Tupel aus Modifikation und Introduktion in den Raum der Introduktionen abbildet.

$$\odot : M \times I \to I \tag{3.8}$$

mit der Abbildungsvorschrift

$$m \odot i = (t, c) \odot i = \begin{cases} c(i) & \text{, wenn } t(i) = i \wedge i \neq \xi \\ i & \text{, wenn } t(i) = \xi \end{cases} \tag{3.9}$$

Eine weitere Folge für Modifikationen hier ist die Unfähigkeit, die leere Introduktion, d.h. z.B. das leere Programm zu erweitern, was mit der Introduktionssumme an sich ohne weiteres möglich ist.

Modifikationsanwendung ist linksdistributiv über der Introduktionssumme, d.h. es ist egal ob vor Anwendung einer Modifikation zunächst die atomaren Introduktionen addiert werden, oder die Modifikation auf alle Summanden einzeln angewendet wird.

$$m \odot (i_n \oplus \ldots \oplus i_2 \oplus i_1) = (m \odot i_n) \oplus \ldots \oplus (m \odot i_2) \oplus (m \odot i_1) \tag{3.10}$$

## 3.5 Komposition von Modifikationen

Der Operator wird $\odot$ überladen mit der *Modifikationskomposition*.

$$\odot : M \times M \to M \tag{3.11}$$

mit der Abbildungsvorschrift

(nur der erste $\odot$ ist eine Modifikationskomposition)

$$(m_2 \odot m_1) \odot i = m_2 \odot (m_1 \odot i) \tag{3.12}$$

Das bedeutet, dass nach Modifikationskomposition eine sequenzielle Anwendung der Modifikationen stattfindet. Somit kann eine Serie von Modifikationen Schritt-für-Schritt abgearbeitet werden

$$(m_n \odot \ldots \odot m_2 \odot m_1) \odot i = m_n \odot (\ldots \odot (m_2 \odot (m_1 \odot i)) \ldots) \tag{3.13}$$

Bei dieser Sequenz kann es vorkommen, dass $m_1$ eine Introduktion $i_{m_1}$ zu $i$ addiert, für die die Auswahlfunktion von $m_2$, $t_{m_2}(i_{m_1}) = i_{m_1}$ liefert. Das bedeutet dass in der zeitlichen Reihenfolge eher angewandte Modifikationen die zeitlich späteren Modifikationen in ihrer Anwendung durchaus beeinflussen können. Wichtig dabei ist, dass dies nicht auf eine schädliche Weise passieren kann, wie z.B. $m_1$ löscht eine Intruduktion, so dass $m_2$ bei der Quantifizierung plötzlich eine leere Menge von Introduktionen erzielt und die Modifikation ohne Effekt bleibt, weil Introduktionen durch die gegebenen Operatoren nicht entfernt werden können (siehe 3.3).

### 3.5.1 Algebraische Eigenschaften der Modifikationsanwendung / Modifikationskomposition

Zwei Modifikationen $m_1, m_2$ sind genau dann äquivalent, wenn sie angewandt auf alle Introduktionen das selbe Ergebnis erzeugen, d.h es muss gelten

$$\text{Für alle } i \in I : m_1 \odot i = m_2 \odot i \tag{3.14}$$

Modifikationen werden aufgrund dieser Eigenschaft in den weiteren Betrachtungen durch Repräsentanten ihrer Äquivalenzklasse ersetzt. Sei $M$ die Menge der Äquivalenzklassen der Modifikationen, dann bildet das Tupel $(M, \odot, \zeta)$ ein nicht kommutatives, nicht idempotentes Monoid [ALMK08] und hat somit folgende Eigenschaften:

- Assoziativität, d.h. es ist für das Ergebnis egal, ob zunächst Modifikationen komponiert und dann auf das Ergebnis angewandt oder die Modifikationen nicht komponiert und der Reihe nach angewandt werden.
- Identität, d.h. es existiert eine leere Modifikation $\zeta$, die effektiv an keiner Introduktion etwas verändert. Dies kann z.B. von einer Auswahlfunktion $t_{leer} := \xi$ kommen, die immer die leere Introduktion zurückgibt. Genausowenig wird von dieser leeren Modifikation eine andere Modifikation durch Komposition der beiden beeinflusst.

- Nicht-Kommutativität, d.h. die Reihenfolge der Modifikationsanwendung ist relevant. Ersichtlich beispielsweise für folgenden Fall: Eine Modifikation $m_1$ fügt eine Introduktion $i_{m_1}$ ein. Eine Modifikation $m_2$ alteriert alle Introduktionen. Das Permutieren der Reihenfolge erzeugt somit im Ergebnis entweder einen alterierten Summanden $i_{m_1}$ oder den original von $m_1$ eingefügten nicht durch $m_1$ alterierten Knoten.
- Nicht-Idempotenz, d.h. das mehrfache Anwenden einer Modifikation ist im Allgemeinen nicht äquivalent zum einfachen Anwenden einer Modifikation. Wenn eine Modifikation z.B. eine konkrete Java Methode so abändert, dass ein Codefragment konkateniert wird, so wird bei mehrfacher Anwendung das Codestück auch mehrfach konkateniert.

## 3.6 Das Quark Modell

Wir haben jetzt die Mengen der Introduktionen $I$ und der Modifikationen $M$, dazu die Operatoren $\oplus : I \times I \to I$, $\odot : M \times M \to M$ und $\odot : M \times I \to I$ zur Verfügung um Features zu beschreiben und zu komponieren.

An dieser Stelle werden *vollwertige Features* eingeführt, die Konzepte der Introduktion und Modifikation in sich vereinen. Desweiteren werden Modifikationen aufgeteilt in *lokale* und *globale* Modifikationen.

Zusammengefasst wird das in dem *Quark Modell*. Ein *Quark* ist ein 3er Tupel bestehend aus Kompositionen von lokalen Modifikationen $l$ und von globalen Modifikationen $g$ sowie aus einer Summe aus Introduktionen $i$:

$$f = \langle g, i, l \rangle = \langle g_j \odot \ldots \odot g_1, i_k \oplus \ldots \oplus i_1, l_m \odot \ldots \odot l_1 \rangle \tag{3.15}$$

**lokale Modifikationen** können nur Introduktionen verändern, die bei der Anwendung des kapselnden Features bereits existieren

**globale Modifikationen** werden erst angewandt wenn alle Introduktionen und lokalen Modifikationen einer Serie von zu komponierenden Features abgearbeitet sind, d.h. diese Modifikationen beeinflussen auch Introduktionen die zeitlich gesehen erst nach Anwendung des kapselnden Features addiert werden.

Ein *Basis Feature*, sieht in Quark Notation so aus:

$$f_{Basis} = \langle \zeta, i, \zeta \rangle \tag{3.16}$$

Die Komposition ist

$$f_{Basis_1} \bullet f_{Basis_2} = \langle \zeta, i_1, \zeta \rangle \bullet \langle \zeta, i_2, \zeta \rangle = \langle \zeta, i_1 \oplus i_2, \zeta \rangle \tag{3.17}$$

## 3.7 Komposition von Quarks

Die Schwierigkeit bei der Kompostion von *vollwertigen Features* ist die Reihenfolge der Modifikationsanwendung.

$$\begin{aligned} f_n \bullet \ldots \bullet f_2 \bullet f_1 &= \langle g_n, i_n, l_n \rangle \bullet \ldots \bullet \langle g_2, i_2, l_2 \rangle \bullet \langle g_1, i_1, l_1 \rangle \\ &= \langle g_n \odot \ldots \odot g_1, (g_n \odot \ldots \odot g_1) \odot \\ &\quad (i_n \oplus (l_n \odot (\ldots (i_2 \oplus (l_2 \odot i_1))))), l_n \odot \ldots \odot l_1 \rangle \end{aligned} \tag{3.18}$$

Man sieht, dass pro Feature abwechselnd zunächst die atomaren Introduktionen addiert , dann die lokalen Modifikationen angewandt werden. Die globalen Modifikationen werden erst angewandt, wenn dieser Berabeitungsschritt abgeschlossen ist.

# 4 Umsetzung

Dieser Teil der Arbeit befasst sich mit der Umsetzung der o.g. Konzepte.

Zunächst wird die bestehende Implementierung für ein besseres Verständnis kurz umrissen, dannach wird auf die Umsetzung der Konzepte Modifikation und Quantifizierung eingegangen. In Betrachtung gezogen werden dabei Erweiterungen und Änderungen an dem bestehenden Softwareprojekt, dadurch gegebene Problemstellungen und der Aspekt der Erweiterbarkeit der Implementierung um sprachunabhängigkeit zu gewährleisten.

## 4.1 FSTComposer

Der FSTComposer ist ein Softwareprojekt, dass die sprachunabhängige Komposition von Basis Features durch Superimposition von FSTs in Java realisiert.

### 4.1.1 FSTs

Die FSTs der Features werden nach folgendem System erzeugt: Für jedes Feature existiert eine hierarchische Ordnerstruktur, eine sogenannte *containment hierarchy* [ALMK08] (siehe Abb. 4.1). Das Wurzelverzeichnis trägt den Namen des Features. Alle darin enthaltenen Artefakte werden je nach Dateityp/Dateiendung auf eine separate Weise auf FST-Knoten bzw. FST-Unterbäume abgebildet.

Diese Abbildung übernehmen sogenannte builder, beispielsweise gibt es einen *FST-Builder* für Java Dateien. Die Analyse des Quellcodes erfolgt dabei anhand der Java-Grammatik.

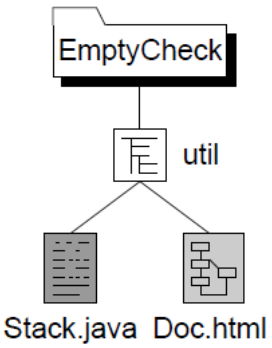


Abbildung 4.1: Containment hierarchy
[Quelle: [AL08]]

```
<Knotentyp (Terminal/Nichtterminal)> -> <Knotenname> : <
   Knotentyp>
<bei Terminalen: Inhalt als Zeichenfolge> compose:<
   Kompositionsregel>
```

Listing 4.1: FST: Notation in der Konsolenausgabe des FST Composer

```
class existingClass1 {
  private String existingField;

  void existingMethod() {
    System.out.print("classic feature");
  }
}
```

Listing 4.2: Java: Beispiel-Klasse

Wie erwähnt ist die Granularität bei der Erzeugung eines FST zwar fast beliebig wählbar, um Artefakte für die hier benötigten Zwecke sinnvoll abzubilden werden Einschränkungen gemacht. Der Builder für Java z.B. besitzt eine Granularität, die bei Methoden terminiert, d.h. Methoden werden als geschlossene Einheit behandelt und nicht weiter analysiert. Aus dem Artefakt in List. 4.2 wird der FST aus List. 4.3 generiert. Die FST-Notation der Konsolenausgabe des FSTComposers ist in List. 4.1 aufgeführt.

Wichtig ist, dass im FST Composer kein globaler FST erzeugt wird, sondern pro Domäne, d.h. pro builder ein separater.

Artefakte, die keinen Quellcode enthalten, z.B. Bilder im jpg Format können nicht mit Hilfe einer Sprachgrammatik analysiert werden. Dann wird im Feld Inhalt lediglich ein Verweis auf das Artefakt im Dateisystem hinterlegt.

### 4.1.2 Komposition

Die gewünschten Features werden vor der Komposition in einer Steuerungsdatei namentlich aufgezählt. Nach Erzeugung der FSTs der Features werden die ausgewählten Features komponiert, und von FST wieder auf Software Artefakte abgebildet.

Bei der Komposition von Features wird das oben erwähnte Konzept der Superimposition eingesetzt. In dem Beispiel in Abb. 4.2 wird die Bedeutung des Begriffs Superimposition nochmal veranschaulicht: Die zu komponierenden FSTs werden sozusagen ‘übereinander gelegt’.

#### Terminale

Die Komposition von zwei Terminalen wird durch sehr verschiedene Algorithmen implementiert. Exemplarisch wird hier die Umsetzung der Komposition zweier Java Mehoden

```
[NT -> existingClass1.java : Java-File]
    [NT -> existingClass1 : CompilationUnit]
      [T -> - : PackageDeclaration "package Graph;" compose:
         Replacement]
      [NT -> existingClass1 : ClassDeclaration]
        [T -> - : Modifiers "" compose:
           ModifierListSpecialization]
        [T -> - : ClassOrInterface1 "class" compose:
           Replacement]
        [T -> existingClass1 : Id "existingClass1" compose:
           Replacement]
        [T -> String-existingField,existingField : FieldDecl
            "private String existingField;" compose:
           Replacement]
        [T -> existingMethod({FormalParametersInternal}) :
           MethodDecl "void existingMethod() { System.out.
           print("classic feature"); }" compose:
           JavaMethodOverriding]]]]
```

Listing 4.3: FST: Java Beispiel-Klasse

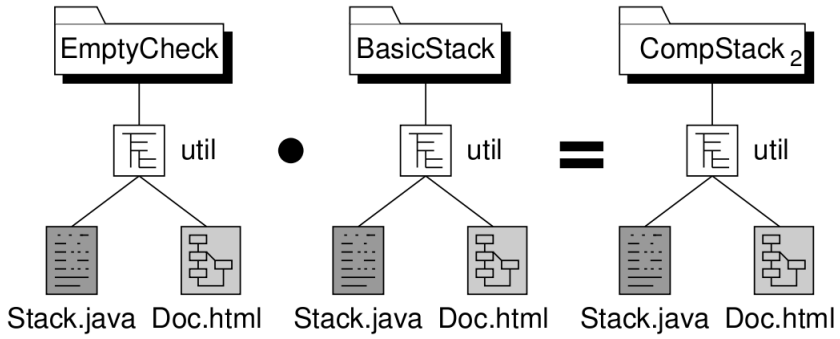


Abbildung 4.2: Superimposition im FSTComposer
[Quelle: [AL08]]

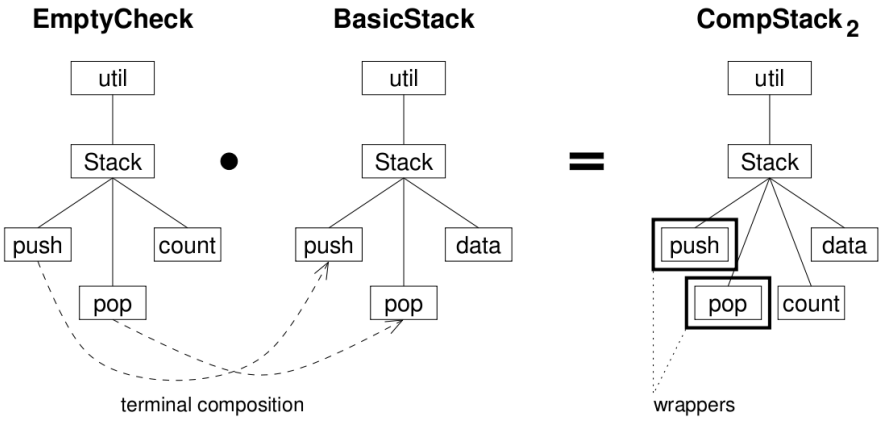


Abbildung 4.3: Komposition von Java Methoden, FST
[Quelle: [AL08]]

angeführt.

Wir betrachten das Codebeispiel in Abb. 4.4. In die Klasse Stack soll ein Zähler eingefügt werden, d.h. es wird ein neues Feld count eingeführt und die Methoden push bzw. pop um einen Aufruf erweitert, der counter in- bzw. dekrementiert.

Um diesen Aufruf zu integrieren wird das Schlüsselwort original eingeführt, das Einfügen von Code vor bzw. nach Ausführung einer Methode ermöglicht. Der Algorithmus des *Java method overriding*[1] erzeugt dann eine zweite Mehode, in die der Code der ursprünglichen Methode ausgelagert wird. Der einzufügende Code ersetzt den ursprünglichen Code, das Schlüsselwort wird mit einem Aufruf der neu erzeugten Methode ersetzt.

Der Code der ursprünglichen Methode wird praktisch von dem neuen Code umschlossen oder umwickelt, deshalb spricht man hier auch von *wrapping* [AL08].

## 4.2 Ausdruckssprache für Duchlauffunktionen

Die Aufgabenstellung war es, eine möglichst mächtige und komfortable Sprache zu entwerfen, die die es dem Benutzer ermöglicht Auswahlfunktionen wie in 2.3.3 durch einfache Klartextausdrücke zu spezifizieren.

Zunächst wurde die Spezifikation der Funktion angepasst. Diese bildet in ihrer ursprünglichen Form einzelne Knoten aus FST auf sich selbst oder die leere Menge ab. Für den Einsatzbereich hier ist es jedoch zweckmäßiger, einzelne FSTs auf Mengen von Knoten abzubilden. So kann man die Funktionen auf einen beliebigen FST anwenden und bekommt als Rückgabe die Menge der Knoten, für die die ursprüngliche Funktion

[1] Es handelt sich dabei um eine Kompositionsregel für Terminale, in dem Fall Java Methoden

```
1 package util;
2 class Stack {
3   int count = 0;
4   void push(Object obj) { original(obj); count++; }
5   Object pop() {
6     if(count > 0) { count--; return original(); } else return null;
7   }
8 }
```

●

```
1 package util;
2 class Stack {
3   LinkedList data = new LinkedList();
4   void push(Object obj) { data.addFirst(obj); }
5   Object pop() { return data.removeFirst(); }
6 }
```

=

```
1 package util;
2 class Stack {
3   int count  = 0;
4   LinkedList data = new LinkedList();
5   void push_wrappee(Object obj) { data.addFirst(obj); }
6   void push(Object obj) { push_wrappee(obj); count++; }
7   Object pop_wrappee() { return data.removeFirst(); }
8   Object pop() {
9     if(count > 0) { count--; return pop_wrappee(); } else return null;
10  }
11 }
```

Abbildung 4.4: Komposition von Java Methoden, Code
[Quelle: [AL08]]

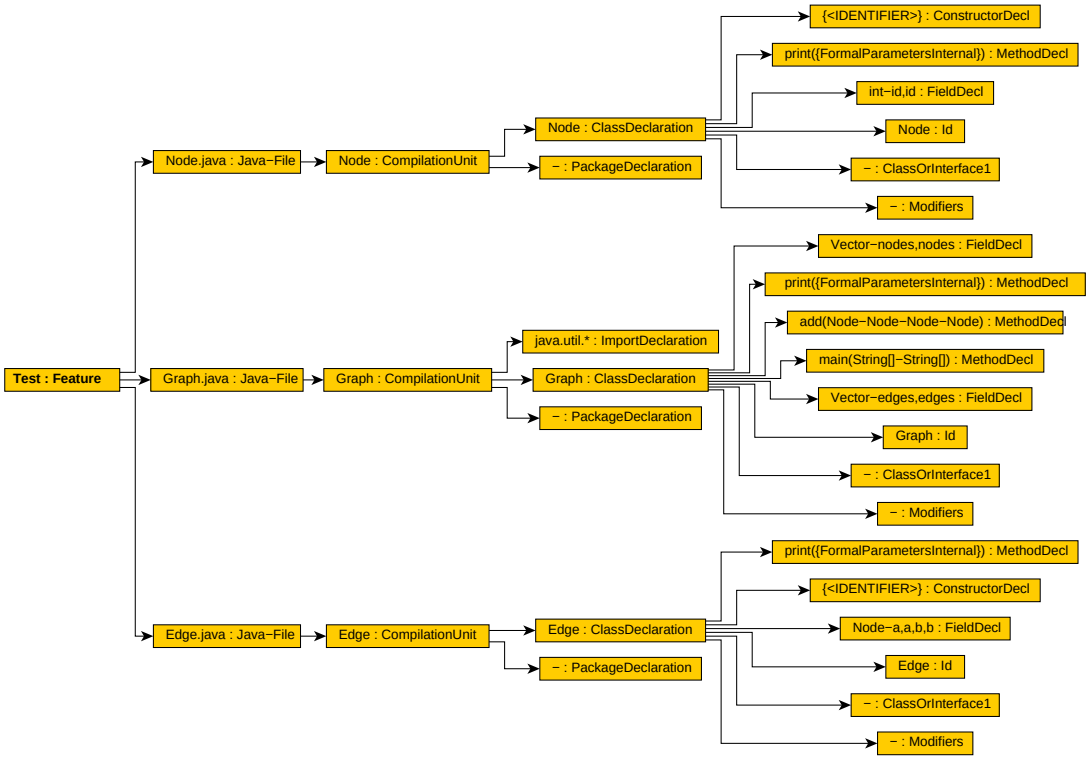


Abbildung 4.5: Generierter Beispiel-FST

die Argumente selbst zurückgegeben hätte.

$$t_{neu} : FSTNode \rightarrow \mathcal{P}(FSTNode) \tag{4.1}$$

$$t_{neu}(n_{Wurzel}) = A; \qquad n_{Wurzel} \in FSTNode, A \in \mathcal{P}(FSTNode) \tag{4.2}$$

$$\tag{4.3}$$

so dass für alle Nachfahren $n \in FSTNode$ von $n_{Wurzel} \in FSTNode$ gilt:

$$t_{alt}(n) = n \leftrightarrow n \in A \tag{4.4}$$

$$t_{alt}(n) = \emptyset \leftrightarrow n \notin A \tag{4.5}$$

### 4.2.1 Syntax

Die Grundidee der Syntax für die Ausdrücke ist in [ALMK08] bereits angedeutet. Es handelt sich dabei um die Angabe von Pfaden im FST, die im Folgenden als *Adressen* bezeichnet werden. Die Auswahl eines Knotens durch eine Auswahlfunktion nennt man deshalb *Adressierung*.

Der Einfachheit halber folgt die detaillierte Sprachbeschreibung exemplarisch. Wir erinnern uns, dass alle Knoten im FST sowohl Name, als auch Typ besitzen, die Notation für einen einzelnen Knoten ist dabei

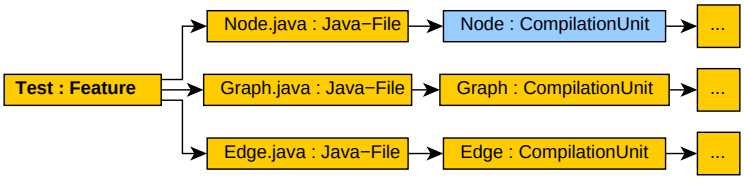


Abbildung 4.6: Einfacher Ausdruck

`<Name>:<Typ>`

Diese Struktur wird im Folgenden als *Addressteil* bezeichnet. Alle vorkommenden Ausdrücke werden für den FST in Abb. 4.5 ausgewertet. Dieser FST wurde erzeugt aus Artefakten eines Features der *Graph Product Line (GPL)*, welche in 5 nochmal separat als Studie betrachtet wird.

### Einfache Ausdrücke

Der einfache Ausdruck

`Test:Feature.Node.java:Java-File.Node:CompilationUnit`

ist die genaue Adresse eines einzelnen Knotens. Die Einzelschritte von der Wurzel bis zum Endknoten werden durch das Zeichen ‘.’ aneinander gereiht.[2]

Angewandt auf den Beispiel-FST Abb. 4.6 ist der Knoten blau markiert. Mit diese Art von Ausdruck können im allgemeinen bereits alle Einzel-Knoten eines FST erreicht, d.h. *adressiert* werden.

### Platzhalter

Ein sehr mächtiges Konzept für diese Ausdrücke sind Platzhalter. Sie erfüllen mehrere Zwecke.

Zum Einen können dadurch dem Benutzer unbekannte Bereiche sowohl im FST selbst, als auch in Namen oder Typen einzelner Knoten trotzdem erreicht werden.

Zum Anderen können dadurch bewusst Pfadverzweigungen verallgemeinert werden und somit gleich mehrere Pfade durchlaufen bzw. mehrere Knoten auf einmal adressiert werden.

**Adressteile** Der Platzhalter ‘..’ hat die Bedeutung von *jede beliebige existierende Adresse*. Intern werden für den Platzhalter alle möglichen Pfade errechnet, jeder mögliche Pfad wird separat ausgewertet. Rückgabe ist die Vereinigung aller ausgewerteten Adressen.

Der Ausdruck

`..Node:CompilationUnit`

[2] Auf eine Sonderbehandlung des Zeichens ‘.’ in Node.java wird hier der Einfachheit halber zunächst verzichtet. In 4.2.1 werden Sonderzeichen behandelt.

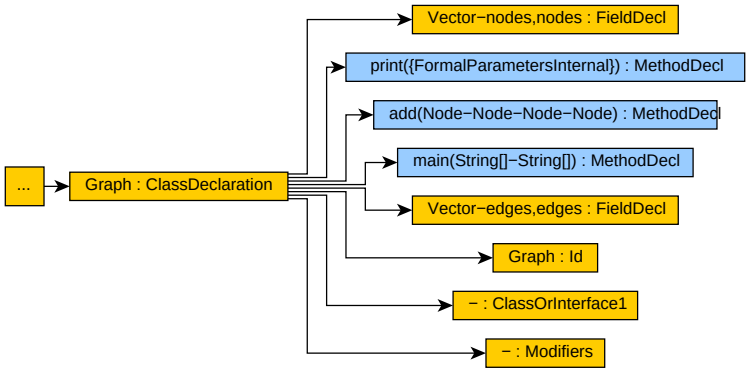


Abbildung 4.7: Platzhalter 1

wird ausgewertet als 'Finde alle Knoten, die Node:CompilationUnit als Name/Typ Paar besitzen'. Auf unseren Beispiel-FST bezogen ergibt das wieder genau denselben blau markierten Knoten wie in Abb. 4.6.

Dieser Platzhalter beinhaltet auch den leeren Pfad, d.h.

```
..Test.Feature
```

ist ein legitimer Ausdruck und adressiert die Wurzel Test.Feature.

**Name bzw. Typ** Platzhalter können auch innerhalb von Adressteilen eingesetzt werden. Das Zeichen hierfür ist '*'. Die Bedeutung ist *jede beliebige Zeichenfolge*, d.h. auch die leere Zeichenfolge.

Der Ausdruck

```
..Graph:ClassDeclaration.*:MethodDecl
```

liefert die blau markierten Knoten in Abb. 4.7 zurück. Das sind alle Knoten vom Typ MethodDecl, d.h. konkret alle Methoden.

Diese Platzhalter können beliebig mit normalen Zeichenfolgen kombiniert werden. Der Ausdruck

```
..Graph:ClassDeclaration.Vector*:MethodDecl
```

gibt alle Knoten vom Typ FieldDecl zurück, deren Name mit Vector beginnt (Abb. 4.8)

Konkret heißt das alle Felder der angegebenen Klasse die vom Typ Vector sind.

### Bedingte Adressierung

Dieses Konzept wurde erst nachträglich im Verlauf der Fallstudie zur *Berkeley DB* eingeführt. Die Gründe hierfür findet man in 5.1.2.

Die Idee dahinter ist die Adressierung eines Knotens, wenn für mindestens einen seiner Nachkommen die angegebene Bedingung erfüllt wird. Der Ausdruck

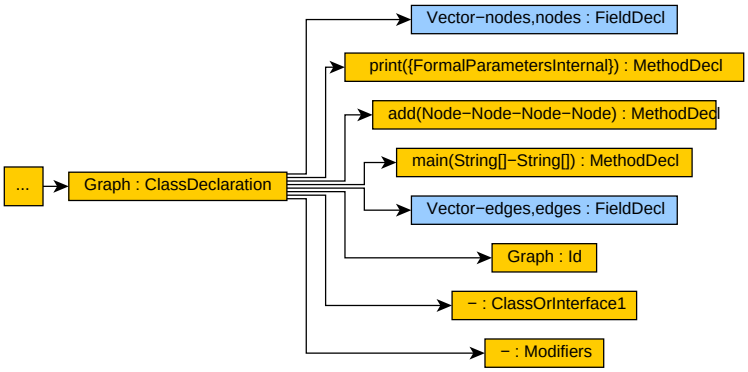


Abbildung 4.8: Platzhalter 2

```
..*:Java-File[..add*.MethodDecl]
```

liefert alle Knoten vom Typ Java-File (Abb. 4.9 blau und grün), die als zusätzliche Bedingung mindestens einen Nachfolger vom Typ MethodDecl mit Namen add* (Abb. 4.9 rot) besitzen müssen.

Konkret bedeutet das alle Java Dateien, die eine Methode add* beinhalten.

### Sonderzeichen

Viele Zeichen bzw. Zeichenfolgen bestitzen eine besondere Semantik und sind daher reserviert. Wenn man diese Zeichen/-folgen in der Beschreibung von Namen bzw. Typen einsetzen will, so muss man diese in die Umgebung

```
%s<beliebige Zeichen>%e
```

setzen. Die reservierten Zeichen/-folgen sind

```
"||" "&&" "--" "sib+" "." ":" ".." "*" "]" "[" "(" ")"
```

Leerzeichen haben in Ausdrücken keine Bedeutung, es sei denn sie werden innerhalb der o.g. Umgebung benutzt.

### Operatoren

Dem Anwender stehen auch eine Reihe von Operatoren zur Verfügung, die syntaktisch korrekte Ausdrücke miteinander verknüpfen oder verändern. Die ersten drei Operatoren sind von der Semantik äquivalent zu ihren Pendanten aus der Mengenalgebra.

Die Operation an sich bildet ebenfalls einen korrekten auswertbaren Ausdruck.

**Vereinigung** Zeichenfolge: ||

```
..Graph:ClassDeclaration.*:MethodDecl
|| ..Graph:ClassDeclaration.*:FieldDecl
```

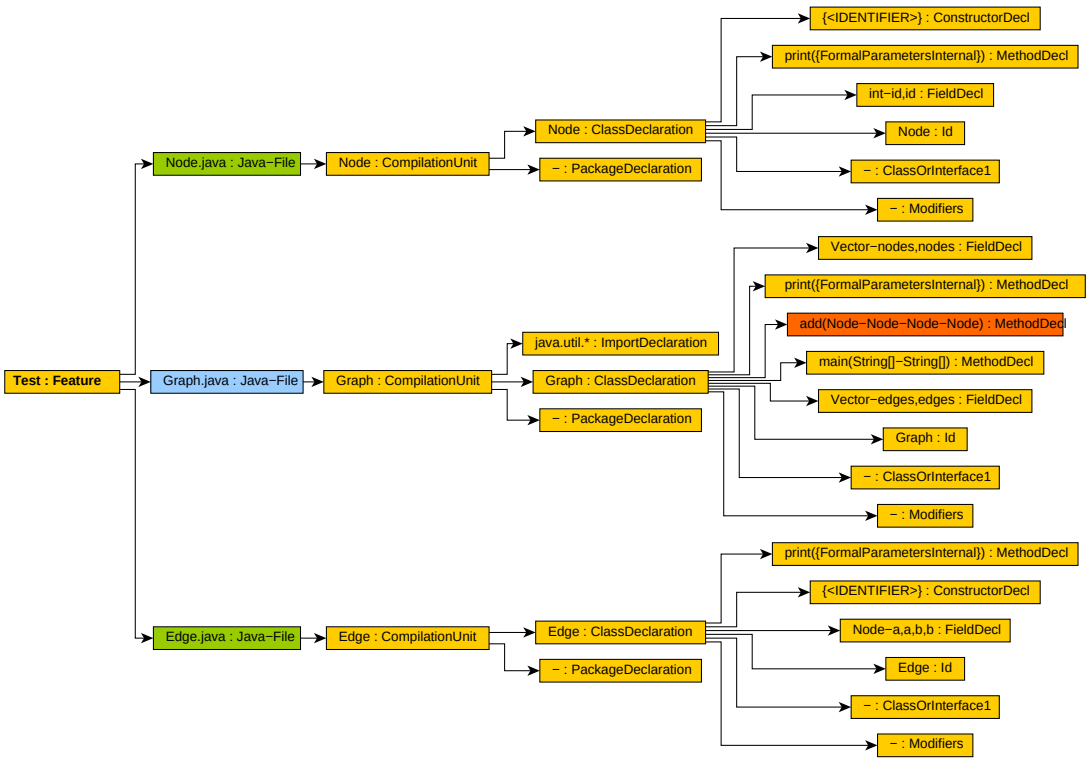


Abbildung 4.9: Bedingte Adressierung

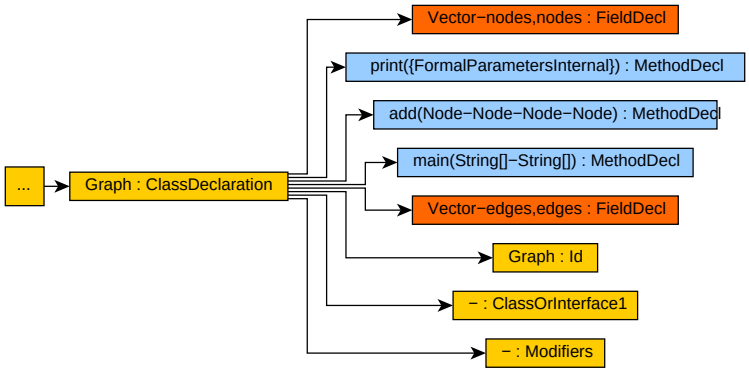


Abbildung 4.10: Vereinigung

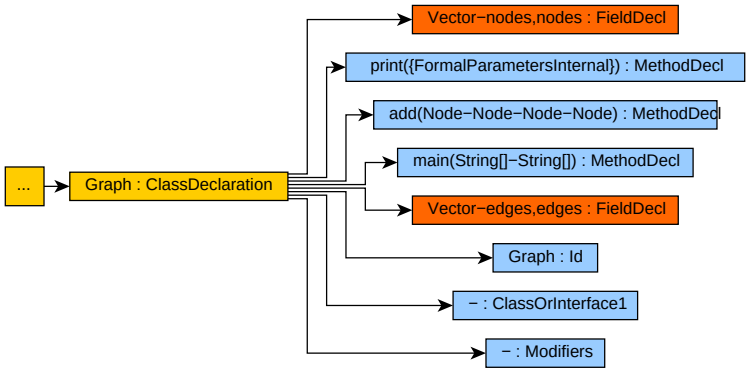


Abbildung 4.11: Subtraktion

Abb. 4.10: Das linke Argument des Ausdrucks (blau) wird mit dem rechten Teil (rot) vereinigt (blau und rot zusammen)

**Subtraktion** Zeichenfolge: –

```
..Graph:ClassDeclaration.*:*
-- ..Graph:ClassDeclaration.*:FieldDecl
```

Abb. 4.11: Der Subtrahend (rot) wird vom Minuend (alle Kinder von Graph:ClassDeclaration) abgezogen und ergibt die Differenz (blau).

**Schnitt** Zeichenfolge: &&

```
..*:MethodDecl
&& (..Graph:ClassDeclaration.*:*
        || ..Node:ClassDeclaration.*:*)
```

siehe Abb. 4.12: Das linke Argument (alle Knoten vom Typ MethodDecl) wird geschnitten mit dem rechten Argument (alle Kinder von Graph:ClassDeclaration und Node:ClassDeclaration) ergibt die blau markierten Knoten. Rot und Grün fallen dadurch aus der Ergebnismenge raus.

**Addition von Geschwistern** Dieser Operator wurde ebenfalls erst im Laufe der Fallstudie zur *Berkeley DB* eingeführt.

Er ist unär und wird dem Argument vorangestellt. Dadurch werden alle Geschwister der adressierten Knoten des Arguments in die Ergebnismenge mit eingeschlossen.

Der Ausdruck

```
+sib ..*:PackageDeclaration
```

adressiert somit neben aller Knoten vom Typ PackageDeclaration (blau) auch noch alle Knoten, die auf derselben FST-Ebene liegen wie diese Knoten (rot).

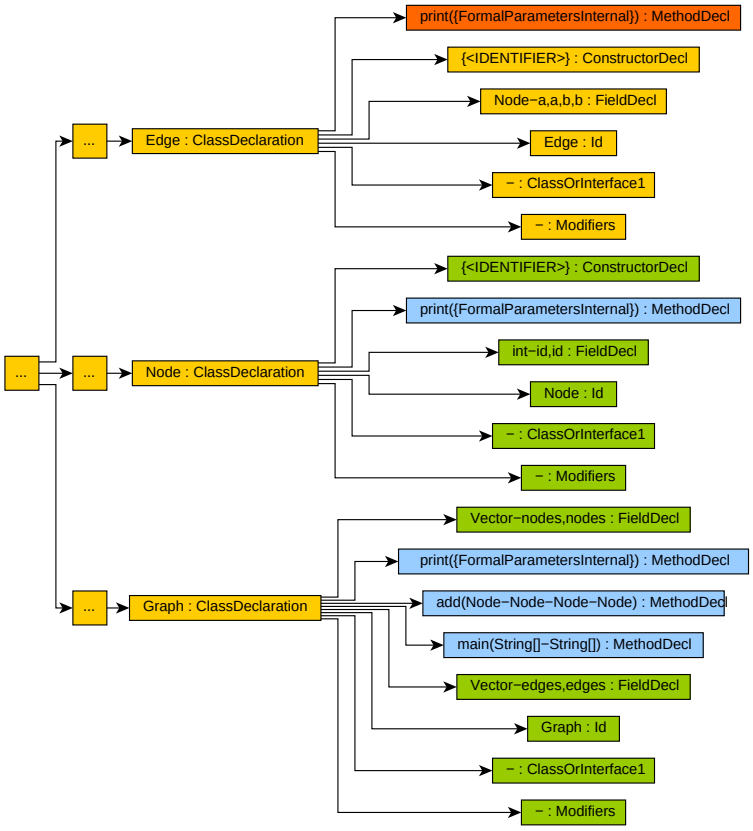


Abbildung 4.12: Schnitt

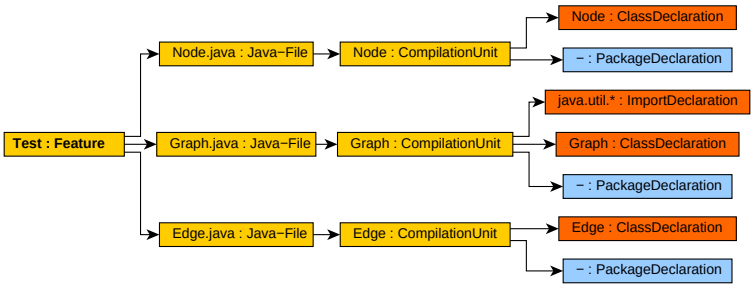


Abbildung 4.13: Addition von Geschwistern

### 4.2.2 Implementierungsdetails

Die Interpretation der Ausdruckssprache wird von einem Parser durchgeführt, der mithilfe des *Java Compiler Compilers*[3] generiert wurde.

Daneben kommen noch eine Hand voll Hilfsklassen für das Adressmanagement, wie z.B. das Auflösen einer vollständigen Adresse anhand der Wurzel eines FSTs, zum Einsatz.

Die Mengen wurden als DuplicateFreeLinkedLists umgesetzt. Diese Klasse erweitert LinkedList mit dem Zusatz der Duplikatfreiheit.

## 4.3 Veränderungsfunktion

Die Veränderungsfunktion wird umgesetzt als Tupel aus Spezifizierung des Typs (2.3.3) und einem als FST hinterlegten Software Artefakt, wie z.B. einem Code-Stück das eine Java-Methode enthält. Dieser Knoten wird im Folgenden als *Inhalt* der Veränderungsfunktion bezeichnet.

### 4.3.1 Typ

#### Einfügen von neuen Knoten

Das Einfügen ist in der Umsetzung trivial: Die bestehende Implementierung der FSTs erlaubt es bereits neue Kinder an nichtterminale Knoten zu hängen.

Allerdings können beim Einfügen von neuen Kindern zu diesem Zeitpunkt noch keine genaueren Positionsangaben bezüglich der Reihenfolge gemacht werden: neue Knoten werden immer hinten, d.h. rechts von den bestehenden Kindern eingefügt.

Hier besteht also eine Abweichung von dem theoretischen Konzept, bei dem die Reihenfolge wichtig wäre für die Erhaltung einer bijektiven Abbildung von FST nach Software-Artefakt.

Im Verlauf dieser Arbeit rückte diese Problematik allerdings in den Hintergrund. Die bearbeiteten FST-Strukturen können aufgrund der Granularität der FST-Erzeugung und aufgrund der Beschaffenheit der vorkommenden Domänen als äquivalent bezüglich Permutierung der Reihenfolge der Kinder eines Knotens betrachtet werden.

Man denke z.B. an Felder bzw. Methoden in Java. Als Einheit betrachtet ist es irrelevant an welchen Stellen im Quellcode sich diese Deklarationen befinden.

Eine mögliche Lösung des Problems bestünde in einer erweiterten Quantifizierung, die auch Positionsangaben wie ‘rechts von einem bestimmten Knoten’ erlaubt.

#### Alterieren von bestehenden Knoten

Wie bereits in 2.3.1 erläutert braucht man bei der Komposition von Terminalen im Allgemeinen für fast jeden terminalen Knoten-Typ eine andere Regel.

[3] https://javacc.dev.java.net/

Die Umsetzung ermöglicht eine Wiederverwendung dieser Regeln bei der Alterierung bestehender Knoten. Das funktioniert, weil nach der Quantifizierung die Alterierung analog zur Komposition von Terminalen während der Superimposition abläuft.

### 4.3.2 Inhalt

Der Inhalt der Veränderungsfunktion besteht wie oben erwähnt aus einem FST. Dieser FST kann auf zwei verschiedene Arten spezifiziert werden.

#### manuelle Spezifizierung

Diese Art der Spezifizierung erfordert relativ viel Wissen über die Struktur der FSTs in der Implementierung des FST Composers. Man benötigt zur Erstellung eines Knotens Name, Typ, Inhalt und einige weitere implementationstechnische Attribute. Dieser Typ wurde deshalb fast ausschließlich in der Testphase genutzt.

#### Spezifizierung durch ein geparstes Software-Artefakt

Hier kommen die in dem Software Projekt bereits vorhandenen FST Generatoren zum Einsatz. Ursprünglich werden diese Generatoren eingesetzt, um eine beliebige hierarchische Anordnung von Software Artefakten eines bestimmten Typs, z.B. Java in einer Ordnerstruktur, auf einen FST abzubilden.

Für den Inhalt benötigt man jedoch nicht immer die Granularität der als Datei gekapselten Software-Artefakte. Um diese Problematik zu lösen, wurden zwei Konzepte umgesetzt:

- Übersetzten eines als Datei gekapselten Software Artefakts mit anschließender optionaler Auswahl eines Unterbaums durch eine Auswahlfunktion, z.B. wird eine ganze Java-Datei auf einen FST abgebildet, und dann optional per Auswahlfunktion aus dem FST eine bestimmte Methode eingeschränkt.
- Anpassung des FST Generators zur direkten Übersetzung von Artefakten höherer Granularität, z.B. einer Java-Methode. Optional kann auch hier eine Auswahlfunktion angegeben werden, um die Auswahl auf einen Unterbaum des generierten FST einzuschränken.

## 4.4 Modifikationen

komplette Modifikationen werden mit Hilfe der *Extensible Markup Language (XML)*[4] deklariert. Vorteile gegenüber beispielsweise einer Klartext-Spezifikation finden sich zum einen in der sehr verbreitete Unterstützung durch bereits bestehende Parser-Gerüste und

[4] http://www.w3.org/XML/

```
<modification>
  <type>
  <traversal>
  <content>
</modification>
```

Listing 4.4: XML: Grundstruktur einer Modifikation

zum anderen in der Einfachheit/Erweiterbarkeit der XML-Struktur an sich. Die Nachfolgenden Erklärungen setzten grundlegende Kenntnisse der *XML Schemavalidierung*[5] voraus.

### 4.4.1 Syntax

List. 4.8 enthält das vollständige Schema für die Validierung der Syntax. Zur Erläuterung wird das Schema aufgeteilt.

Für eine valide Modifikationen müssen nun wie aufgezählt eine Reihe von Parametern angegeben werden, die im Folgenden implementationsbedingt durch englische Begriffe ersetzt werden. Wenn von *Typen* die Rede ist bedeutet das *Typen des XML Schemas*.

List. 4.4 zeigt die Grundstruktur einer Modifikation:

- traversal: der Ausdruck zum Spezifizieren der Auswahlfunktion ist vom einfachen Typ string
- type: der Typ (nicht XML!) der Änderungsfunktion ist ebenfalls vom einfachen Typ string. Dieses Attribut besitzt zwar nur eine endliche Anzahl von validen Werten, wurde aber vom Schema her der einfachen Erweiterbarkeit wegen nicht auf diese validen Ausprägungen beschränkt. Diese sind zum Zeitpunkt der Abgabe dieser Arbeit:

  **introduction:** Einfügen in den FST

  **superimposition:** Alterieren eines Knotens

  **javaMethodBodyOverriding:** Eine sehr spezielle Ausprägung der Änderungsfunktion, eingeführt während der Fallstudien zur Berkeley DB (siehe 5.1.2)

- content: der Inhalt der Modifikation ist ein komplexer Typ, der weiter aufgeschlüsselt werden muss.

Um die in 4.3.1 aufgelisteten Konzepte zu integrieren, enthält content genau einen der folgenden Typen:

- custom: der manuell spezifizierte Inhalt (siehe List. 4.5). Alle Typen in custom sind vom einfachen Typ string.

[5] http://www.w3.org/TR/xmlschema-0/

```
<content>
  <custom>
    <nodeType>
    <name>
    <body>
    <prefix>
    <compositionMechanism>
  </custom>
</content>
```

Listing 4.5: XML: manuelle Spezifikation von content

```
<content>
  <parsed>
    <plainText>
      <text>
      <tType>
    </plainText>
    <cTraversal>
  </parsed>
</content>
```

Listing 4.6: XML: geparstes Code-Stück als content

```
<content>
  <parsed>
    <externLink>
    <cTraversal>
  </parsed>
</content>
```

Listing 4.7: XML: geparste Datei mit Auswahlfunktion als content

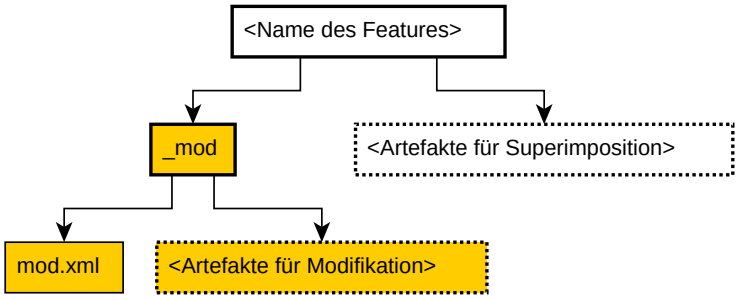


Abbildung 4.14: Dateistruktur eines Features mit Modifikationen

- parsed: Ein zu parsendes Artefakt. Enthält zunächst genau einen der beiden folgenden Typen:
  - plainText: ein zu parsendes Code-Stück. Es enthält text, d.h. den Quelltext des Code-Stücks als Klartext (einfacher Typ string) und tType den Typen des Code-Stücks, ein Hilfsparameter für den FST Generator, z.B. java.method (einfacher Typ string).
  - externLink: eine Referenz auf eine zu parsende Datei (einfacher Typ string).

  und optional einem Ausdruck zur Spezifizierung einer Auswahlfunktion cTraversal, die einen Teil des Inhalts selektiert.

### 4.4.2 Integration in den FST Composer

Modifikationen können in beliebiger Zahl in ein Feature gekapselt werden. Die Spezifikation der Modifikationen erfolgt nach dem o.g. Schema (List. 4.8) in einer Datei mod.xml. Diese Datei befindet sich in einem Ordner _mod der widerum direkt unter der Wurzel der Artefaktstruktur liegt. In Abb 4.14 ist diese Struktur in einem Diagramm schematisiert. Referenzen innerhalb der XML-Datei werden relativ zum Ordner _mod gesetzt.

In [ALMK08] wurde bei der Konstruktion von Quarks zwischen globalen und lokalen Modifikationen unterschieden (siehe auch 3.5.1). Bei der Integration in den FST Composer wurden lediglich globale Modifikationen umgesetzt. Dies ist vor allem auf implementationstechnische Hindernisse zurückzuführen: Der Einstiegspunkt für die Modifikationsanwendung liegt nämlich im zeitlichen Verlauf der Feature-Komposition hinter der Superimposition aller Features. Eine Anpassung dieses Verhaltens wäre mit einer größeren Änderung des FST Composers einher gegangen und hätte den Rahmen dieser Arbeit überschritten.

Die Reihenfolge der Modifikationen in der XML-Spezifikation entspricht der Reihenfolge der Anwendung derselben im Verlauf der Komposition.

```
<xml version="1.0" encoding="UTF-8">
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="modificationComposition">
    <complexType>
      <sequence>
        <element name="modification" minOccurs="1" maxOccurs="unbounded">
          <complexType>
            <sequence>
              <element name="type" type="string"></element>
              <element name="traversal" type="string"></element>
              <element name="content">
                <complexType>
                  <choice>
                    <element name="parsed">
                      <complexType>
                        <sequence>
                          <choice>
                            <element name="plainText">
                              <complexType>
                                <sequence>
                                  <element name="text" type="string">
                                  </element>
                                  <element name="tType" type="string">
                                  </element>
                                </sequence>
                              </complexType>
                            </element>
                            <element name="externLink" type="string">
                            </element>
                          </choice>
                          <element name="cTraversal" type="string" maxOccurs="1"
                              minOccurs="0">
                          </element>
                        </sequence>
                      </complexType>
                    </element>
                    <element name="custom">
                      <complexType>
                        <sequence>
                          <element name="nodeType" type="string">
                          </element>
                          <element name="name" type="string">
                          </element>
                          <element name="body" type="string">
                          </element>
                          <element name="prefix" type="string">
                          </element>
                          <element name="compositionMechanism" type="string">
                          </element>
                        </sequence>
                      </complexType>
                    </element>
                  </choice>
                </complexType>
              </element>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
</schema>
```

Listing 4.8: XML: Schema zur Validierung der Modifikationen

```
[NT -> ClassicFeature : Feature]
  [NT -> existingClass1.java : Java-File]
    [NT -> existingClass1 : CompilationUnit]
      [...]
  [NT -> existingClass2.java : Java-File]
    [NT -> existingClass2 : CompilationUnit]
      [...]
```

Listing 4.9: FST: ClassicFeature, Java

### 4.4.3 Einfache Beispiele

Zur Validierung der Implementierung wurden konstruierte einfache Beispiele umgesetzt, von denen hier einige als Einführung in den Umgang mit Modifikationen aufgegriffen werden.

Es geht in diesem Beispiel um die Komposition von zwei Features. ClassicFeature enthält keine Modifikationen und dient als Basis für das Testen der Modifikationen, die ausschließlich in dem zweiten Feature ModificationFeature enthalten sind.

Wie in 4.1 erwähnt, werden die einzelnen Domänen, wie im FST Composer zunächst seperat bearbeitet und erst beim Abbilden von FST nach Software Artefakten wieder zusammengeführt. Die folgenden Betrachtungen sind ebenfalls nach Domäne separiert.

#### Java

Wir betrachten zunächst den FST des ClassicFeature der Domäne Java in List. 4.9. Enthalten sind zwei Klassen mit Methoden und Feldern.

List. 4.10 enthält eine Modifikation, die in jeder vorkommenden Klasse die als zu parsendes Code-Stück angegebene Methode newMethod einfügt.

Ein zu dieser Modifikation äquivalenter Inhalt ist in List. 4.11 als zu parsende Datei mit Auswahlfunktion gegeben. In der Klasse AuxClass gegeben als Datei AuxClass.java befindet sich eine Methode newMethod. Bei Anwendung der Modifikation wird der Inhalt der Modifikation zunächst übersetzt in einen FST, dann wird der Knoten der gewünschten Methode per Auswahlfunktion selektiert.

List. 4.12 zeigt die Auswirkungen der Modifikation. Beide Klassen besitzen nun eine Methode newMethod.

Eine weitere Modifikation (siehe List. 4.13) alteriert die gerade eben eingefügte Methode newMethod. Dabei kommt das Schlüsselwort original zum Einsatz, d.h. die alte Methode wird nicht einfach überschrieben, sondern per Wrapping der Ausführung des neuen Code-Stücks vorangestellt.

Das Ergebnis in List. 4.14 zeigt die abgeänderte Methode newMethod und die per Wrapping eingefügte Methode newMethod___wrappee___ClassicFeature

```
<modification>
  <type>introduction</type>
  <traversal>..*:ClassDeclaration</traversal>
  <content>
    <parsed>
      <plainText>
        <text>public void newMethod(){System.out.println("
           bim");}
        </text>
        <tType>java.method</tType>
      </plainText>
    </parsed>
  </content>
</modification>
```

Listing 4.10: XML: Modifikation, Einfügen von Knoten, Inhalt als zu parsendes Code-Stück, Java

```
[...]
<content>
  <parsed>
    <externLink>AuxClass.java</externLink>
    <cTraversal>..*newMethod*:methodDecl</cTraversal>
  </parsed>
</content>
[...]
```

Listing 4.11: XML: Modifikation, Einfügen von Knoten, Inhalt als zu parsende Datei mit Auswahlfunktion,Java

```
[NT -> ClassicFeature : Feature]
  [NT -> existingClass1.java : Java-File]
    [NT -> existingClass1 : CompilationUnit]
      [T -> newMethod({FormalParametersInternal}) : [...]
      [...]
  [NT -> existingClass2.java : Java-File]
    [NT -> existingClass2 : CompilationUnit]
      [T -> newMethod({FormalParametersInternal}) : [...]
      [...]
```

Listing 4.12: FST: Einfügen von Knoten, Java

```
<modification>
  <type>superimposition</type>
  <traversal>..newMethod*:*</traversal>
  <content>
    <parsed>
      <plainText>
        <text>public void newMethod()
          {original();System.out.println("bam");}</text>
        <tType>java.method</tType>
      </plainText>
    </parsed>
  </content>
</modification>
```

Listing 4.13: XML: Modifikation, Alterieren von Knoten, Java

```
[NT -> ClassicFeature : Feature]
  [NT -> existingClass1.java : Java-File]
    [NT -> existingClass1 : CompilationUnit]
      [T -> newMethod__wrappee__ClassicFeature :  [...]
      [T -> newMethod({FormalParametersInternal}) : [...]
      [...]
  [NT -> existingClass2.java : Java-File]
    [NT -> existingClass2 : CompilationUnit]
      [T -> newMethod__wrappee__ClassicFeature : [...]
      [T -> newMethod({FormalParametersInternal}) : [...]
      [...]
```

Listing 4.14: FST: Alterieren von Knoten, Java

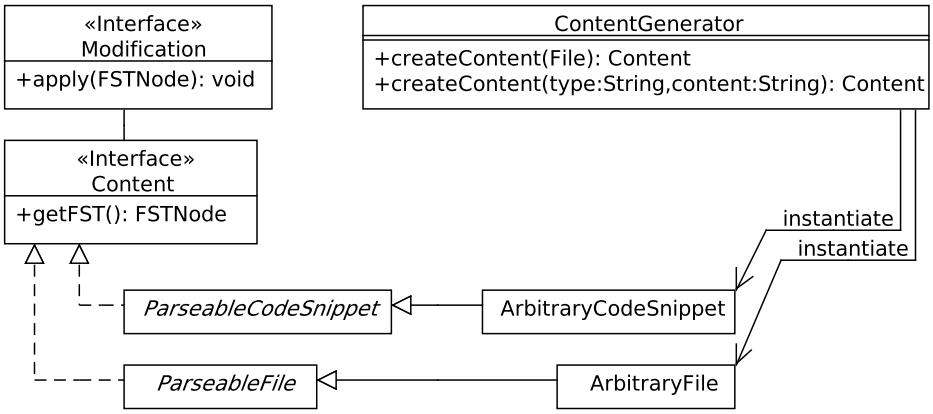


Abbildung 4.15: UML: Änderungsfunktion, Inhaltsgenerierung

#### andere Domänen

Zur Bekräftigung des sprachunabhängigen Ansatzes wurden noch einfache Testfälle für *C* und *Haskell* konstruiert. Die Unterschiede liegen in der Sprachstruktur und somit auch in der FST Struktur. Die Spezifizierung der Modifikationen geschieht ansonsten analog zu Java.

Die implementatorischen Erweiterungen für die Unterstützung weiterer Domänen werden in 4.4.3 aufgeführt.

## 4.5 Erweiterbarkeit

Der domänen/sprachunabhängige Ansatz der Feature Komposition wurde auch bei der Implementierung konsequent durchgesetzt. Trotz aller Abstraktionen von spracheigenen Konstrukten müssen dennoch implementationsbedingt Anpassungen für jede einzelne Domäne durchgeführt werden. Es geht dabei um die Inhalte der Änderungsfunktionen: Aufgrund der unterschiedlichen Struktur der FSTs für jede Domäne, müssen auch diese Inhalte seperat implementiert werden.

Diese Erweiterungen basieren auf der vorhandenen Umsetzung im FST Composer: FST Generator und Kompositionsregeln für Terminale sind dort bereits umgesetzt.

Zunächst erfolgt eine kleine Einführung in die Architektur der Modifikationen. Abb. 4.15 beinhaltet ein vereinfachtes UML-Klassendiagramm der hier relevanten Aspekte: Von Interesse ist die Erzeugung der Inhalte der Änderungsfunktionen (Content), im Folgenden auch als *Inhalte der Modifikationen* bezeichnet.

Der ContentGenerator generiert durch eine gegebene Abbildungsvorschrift dynamisch die richtige Content-Instanz. Für jeden der beiden in 4.3.1 gelisteten Typen enthält ContentGenerator dafür eine separate Methode createContent:

- bei Code-Stücken mit Spezifikation des Typs (ParseableCodeSnippet) erfolgt die

```
public static Content createContent(String type, String
    content) {
  if (type.equals("java.method")) {
      return new JavaMethod(content);
  } else if [...]
}
```

Listing 4.15: Java, createContent Methode

Entscheidung anhand des Typs (String type)

- bei Dateien anhand der Dateiendung

Diese Abbildungsvorschrift ist als if... else.. Konstrukt gegeben.

Welche Änderungen bei der Einführung von neuen Ausprägungen des Inteface content konkret durchgeführt werden müssen, wird im Folgenden exemplarisch an der Integration der Java Methode als Code-Stück gezeigt.

Zunächst wird eine Klasse JavaMethod erstellt, die von der abstrakten Klasse ParseableCodeSnippet beerbt wird. Dann muss in der Klasse ContentGenerator die Abbildungsvorschrift für createContent(String type, String content) angepasst werden. Man überlegt sich zunächst eine aussagekräftige repräsentative Bezeichnung für das Code-Stück. Hier wurde die Notation <Domäne>.<Typ des Code-Stücks> verwendet, in unserem Fall wählen wir also java.method. Dann fügt man in createContent die passende if... else... Klausel ein. Es ergibt sich die veränderte Methode in List. 4.15

Dannach wird die Methode getFST in JavaMethod mit einem Rumpf bestückt. Dafür wirft man einen Blick in den Parser, der bei der FST-Generierung der entsprechenden Domäne zum Einsatz kommt. Diese Parser sind mit dem Java Compiler Compiler umgesetzt. Darin sucht man sich den passenden Einstiegspunkt in die Grammatik für das erwünschte Code-Stück: Die Grammatik ist im Allgemeinen auf vollständige als Datei gekapselte Sprachkonstrukte ausgelegt, d.h. in dem Fall Java-Dateien. Hier benötigen wir jedoch nur eine Teilfunktionalität, d.h. wir überspringen die hierarchisch weiter oben angesetzten Sprachkonstrukte und gehen gleich zu der Stelle, an der ganze Java Methoden bearbeitet werden. Diese befindet sich im Java15Parser bei der Methode ClassOrInterfaceBodyDeclaration.

List. 4.16 enthält die vollständige Methode getFST, erweitert um einige für das Verständnis weniger relevante Implementationsdetails.

```
public FSTNode getFST() {
  Java15Parser p = new Java15Parser(getCharStream());
  p.ClassOrInterfaceBodyDeclaration(false);
  return p.getRoot();
}
```

Listing 4.16: Java, getFST Methode

# 5 Fallstudien

Zur Validierung der Umsetzung wurden mehrere Fallstudien durchgeführt. Die Wertungskriterien beziehen sich sowohl auf die theoretischen Konzepte der Quantifizierung bzw. der Modifikationen als auch auf deren Umsetzung.

Um die Mächtigkeit der Modifikationen zu ermessen, sollten bestehende Software Projekte um Features erweitert werden, die ausschließlich Modifikationen enhalten.

Die Flexibilität der Modifikationen wurde abhand der Umsetzung von Strukturen aus dem Paradigma der *Aspektorientierten* Programmierung getestet.

## 5.1 GPL

Die *Graph Product Line*[1] ist ein Forschungsprojekt der Feature Orientierten Programmierung von kleiner bis mittlerer Größe. Es wurde von Beginn an in diesem Paradigma entwickelt und implementiert. Die Software besitzt die Fähigkeit anhand von Features zunächst einen Graphen zu spezifizieren (gerichtet/ungerichtet, zyklisch/azyklisch, gewichtete/ungewichtete Kanten ...) und dann verschiedene Algorithmen (Tiefensuche, Breitensuche, Zyklenerkennung, Berechnung von minimalen Spannbäumen ...) darauf anzuwenden.

Als Basis der Studie lag eine Version vor, die bereits für den FST Composer fertig umgesetzt war. Ziel war die Erzeugung einer neuen Funktionalität, die Kantenfärbung in Graphen realisieren sollte. Aufgrund der Komplexität der algorithmischen Lösung wurde lediglich eine Heuristik verwendet.

Zunächst wurde die Funktionalität in mehrere Teile, d.h. Features zerlegt:

**Coloring** ermöglicht es jedem Knoten und jeder Kante im Graphen eine Farbe zuzuweisen.

**Degree** ermöglicht es von jedem Knoten im Graphen den Grad auszulesen.

**EdgeGraph** erzeugt den sog. *Kantengraph*, d.h. vereinfacht Knoten werden zu Kanten und Kanten zu Knoten, benötigt Degree.

**EdgeColoring** die geforderte Funktionalität, benötigt alle vorher genannten Features zur ordnungsgemäßen Funktion.

**EdgeColoringTest** ein kleines Testprogramm zur Überprüfung der Implementierung

[1] http://www.cs.utexas.edu/users/dsb/GPL/graph.htm

```
<modification>
  <type>introduction</type>
  <traversal>..Edge:*..*:ClassDeclaration||..Vertex:*..*:
     ClassDeclaration
  </traversal>
  <content>
    <parsed>
      <plainText>
        <text>public int getColor() {return color;}</text>
        <tType>java.method</tType>
      </plainText>
    </parsed>
  </content>
</modification>
```

Listing 5.1: Modifikation, GPL Java, Coloring Feature, getter

### 5.1.1 Java

Die genannten Features wurden zunächst in Java implementiert. Es folgt eine Auflistung der Features und der entstandenen Erfahrungen.

#### Coloring, Degree

Diese beiden Features werden zusammengefasst, weil sie beide lediglich Felder und getter/ setter einfügen und die display Methode, die für die Konsolenausgabe von Knoten und Kanten zuständig ist, alterieren.

Exemplarisch wird die Modifikation betrachtet, die den Getter in die beiden Klassen einfügt. List. 5.1 enthält im Auswahlfunktionsausdruck den Operator ||, wodurch gleich mehrere FST-Knoten, d.h. die beiden Klassen Vertex und Edge als Argument für die Änderungsfunktion übergeben werden.

Hier lässt sich ein Vorteil der Quantifizierung gegenüber der Superimposition klar ablesen: Der querschneidende Belang des getters wird in dieser Modifikation gekapselt und an mehreren Stellen im Quellcode eingebaut.

#### EdgeGraph, EdgeColoring

EdgeGraph und EdgeColoring kapseln keine querschneidenden Belange, es tritt jedoch ein weiterer Vorteil der Quantifizierung auf: Der in dem Feature enthaltene Algorithmus wurde in einer Integrierten Entwicklungsumgebung (IDE) in dem Kontext des ganzen GPL Projektes geschrieben. Dies erhöht den Komfort der Implementierung und bietet die Möglichkeit die Funktionalität ständig zu prüfen.

Wenn das erwünschte Ergebnis erreicht ist, werden die Dateien die die Funktionalitäten enthalten, d.h. Graph.java ohne weitere Änderungen als Inhalt der Modifikationen

```
<modification>
  <type>introduction</type>
  <traversal>..Graph*:*..*:ClassDeclaration</traversal>
  <content>
    <parsed>
      <externLink>Graph.java</externLink>
      <cTraversal>..*getEdgeGraph*:*</cTraversal>
    </parsed>
  </content>
</modification>
```

Listing 5.2: Modifikation, GPL Java, EdgeGraph Feature, getEdgeGraph Methode

verwendet. Durch die Angabe eines weiteren Quantifizierungs-Ausdrucks wird von der ganzen Java-Klasse nur der erwünschte Teil, d.h. die Methode getEdgeGraph für das Feature EdgeGraph und die Methode edgeColoring für das Feature EdgeColoring benutzt.

List. 5.2 zeigt die so entstehende Modifikation für das Feature EdgeGraph, die Modifikation für EdgeColoring wird analog aufgebaut.

#### EdgeColoringTest

Die in diesem Feature einzige enthaltene Modifikation fügt lediglich die komplette Klasse EdgeColoringTest in das Kompositum ein. In dem Fall sind keinerlei Vorteile der Quantifizierung gegenüber der Superimposition erkennbar. Im Gegenteil ist es subjektiv aufwändiger den overhead einer Modifikation zu erzeugen, als die Klasse einfach in einen Ordner zu schieben und per Superimposition komponieren zu lassen.

### 5.1.2 C♯

Für Betrachtungen unter dem Aspekt der Sprachunabhängigkeit wurde die Tatsache ausgenutzt, dass die GPL in C♯ ebenfalls bereits für den FSTCopmposer umgesetzt ist.

Die Implementierung der Features lief nahezu identisch zu Java ab. Lediglich die FST-Struktur sieht bei C♯ ein wenig anders aus und die FST-Knotentypen sind natürlich anders benannt als in Java.

Keine Probleme entstanden beispielsweise durch die andere Behandlung von Methoden. Diese werden in C♯ nicht als Terminale behandelt, sondern noch weiter aufgeteilt in Modifier (z.B. public, private) und Rest. Die Umsetzung hat den Vorteil, dass die Inhalte der Modifikationen nicht auf einzelne FST-Knoten beschränkt sind, sondern von vornherein auf FSTs ausgelegt waren.

List. 5.3 enthält exemplarisch den getter aus dem Feature Coloring. Verglichen mit der Java-Variante in List. 5.1 sind die Unterschiede marginal.

```
<modification>
  <type>introduction</type>
  <traversal>..Edge:*..*:class_body||..Vertex:*..*:
     class_body
  </traversal>
  <content>
    <parsed>
      <plainText>
        <text>public int getColor() {return color;}</text>
        <tType>cSharp.method</tType>
      </plainText>
    </parsed>
  </content>
</modification>
```

Listing 5.3: Modifikation, GPL C♯, Coloring Feature, getter

## 5.2 Berkeley DB

Die *Berkeley DB*[2] ist ein Datenbanksystem und vom Umfang her als großes Software-Projekt zu betrachten. Die folgenden Betrachtungen konzentrieren sich deshalb weniger auf Semantik, und mehr auf Konzepte für Modifikationen.

Entwickelt wurde die Berkeley DB in einem Feature-fremden Paradigma. Die Studien hier basieren auf refaktorisierten Versionen.

### 5.2.1 Tracing Feature

Zielsetzung war es hier, ein simples Feature Tracing zu realisieren, dass in jeder Klasse ein Feld vom Typ Tracer einbaut, das und alle Methoden-Ein- bzw. Austritte mitgelogt.

List. 5.4 enthält die Modifikation die jeder Klasse ein Feld vom Typ Tracer hinzufügt. Hervorzuheben ist an dieser Stelle der Einsatz der bedingten Adressierung im Ausdruck der Auswahlfunktion. Das bedeutet, dass alle Klassen gewählt werden, die auch tatsächlich Klassen, d.h. keine Interfaces sind. Das Einfügen von getter/setter passiert analog.

Man sieht hier, dass umfassende Auswahlfunktionen in großen Software-Projekten vom Ausdruck her recht simpel aufgebaut werden können, wenn die Sprache mächtig genug dafür ist. Ohne den Operator der bedingten Adressierung gäbe es keine Möglichkeit genau diese Auswahl zu treffen, da sich die Information über die Einteilung in Interface und Klasse in der Hierarchie des FST bei Java unerhalb der FST-Knoten der Interfaces bzw. Klassen selbst befindet (siehe List. 5.5).

Für das Tracing selbst, d.h. das Mitlogen von Methoden-Ein- bzw. Austritt mussten ebenfalls Erweiterungen an der Implementierung der Auswahl- und Änderungsfunktion durchgeführt werden.

[2] http://www.oracle.com/database/berkeley-db/index.html

```
<modification>
  <type>introduction</type>
  <traversal>..*:ClassDeclaration[..*:ClassOrInterface1]
  </traversal>
  <content>
    <parsed>
      <plainText>
        <text>private Tracer t = new Tracer();</text>
        <tType>java.field</tType>
      </plainText>
    </parsed>
  </content>
</modification>
<modification>
```

Listing 5.4: Modifikation, Berkeley DB Java, Tracing Feature, getter

```
//interface
[NT -> SomeInterface.java : Java-File]
    [NT -> SomeInterface : CompilationUnit]
      [T -> - : PackageDeclaration "package p;" compose:
         Replacement]
      [NT -> SomeInterface : ClassDeclaration]
        [T -> - : Modifiers "public" compose:
           ModifierListSpecialization]
        [T -> - : ClassOrInterface2 "interface" compose:
           Replacement]
//class
[NT -> SomeClass.java : Java-File]
    [NT -> SomeClass : CompilationUnit]
      [T -> - : PackageDeclaration "package p;" compose:
         Replacement]
      [NT -> SomeClass : ClassDeclaration]
        [T -> - : Modifiers "public" compose:
           ModifierListSpecialization]
        [T -> - : ClassOrInterface1 "class" compose:
           Replacement]
```

Listing 5.5: Modifikation, Berkeley DB Java, Tracing Feature, Unterscheidung in Klasse bzw. Interface

```
<modification>
  <type>javaMethodBodyOverriding</type>
  <traversal>..*:MethodDecl -- sib+ ..*:ClassOrInterface2
  </traversal>
  <content>
    <parsed>
      <plainText>
        <text> { t.in(Thread.currentThread().getStackTrace()
           [1].toString());
          original(); t.out(Thread.currentThread().
             getStackTrace()[1].toString()); }
        </text>
        <tType>java.methodBody</tType>
      </plainText>
    </parsed>
  </content>
</modification>
```

Listing 5.6: Modifikation, Berkeley DB Java, Tracing Feature, Erweitern aller Methoden

Zunächst musste die Ausdruckssprache um den Operator *Addition von Geschwistern* erweitert werden. Die Problemstellung ist der o.g. sehr ähnlich. Es ging darum, alle Methoden aller Klassen auszuwählen, d.h. die Methodenköpfe der Interfaces explizit nicht. Da sich die FST-Knoten der Methoden in Java auf derselben Ebene befinden wie die Information über die Einteilung in Interface und Klasse bot sich dieser Operator an. Der Ausdruck der Auswahlfunktion in List. 5.6 gibt nun alle Methoden-Knoten zurück, wenn die kapselde Einheit vom Typ class ist.

Desweiteren mussten alle Methodenrümpfe erweitert werden, ohne Kenntnisse über den Methodenkopf. Die Kompositionsregeln für Java-Methoden im FST Composer verlangen als eingabe zwei valide und kompatible FST-Knoten vom Typ MethodDecl. Auch das wäre ohne eine Erweiterung der Veränderungsfunktion um eine weiter Ausprägung nicht möglich gewesen. Die eingeführte Ausprägung (javaMethodBodyOverriding) übergeht diese Einschränkung, indem sie vor der Komposition die Informationen über den Methodenkopf von der zu alterierenden Methode in die FST-Struktur der Methode aus dem Inhalt der Modifikation (java.methodBody1) übernimmt (siehe List 5.7). List. 5.6 enthält auch diese Erweiterung.

Zusammenfassend kann man bei der Umsetzung des Tracing-Features wohl keine eindeutige Aussage über Sprachunabhängigkeit machen. Die gewollten Modifikationen konnten zwar umgesetzt werden, jedoch nicht ohne die bestehende Implementation zu erweitern. Es könnte sein, dass sich bei ähnlich großen Projekten auch ähnlich viele neue Schwierigkeiten ergeben, zu deren Lösung die Implementation aufs Neue um sehr spezielle Sprachangepasste Konstrukte wie javaMethodBodyOverriding erweitert werden müsste.

```
public void apply(FSTNode root) {
  [...]
  //copy header info in FST Node
    FSTTerminal contentFST = (FSTTerminal) getContent().
       getFST();
    contentFST.setName(node.getName());
    contentFST.setType(node.getType());
    contentFST.setCompositionMechanism(((FSTTerminal) node)
      .getCompositionMechanism());
    String newBody = ((FSTTerminal) node).getBody().split
       ("[{]")[0];
    newBody = newBody + contentFST.getBody();
    contentFST.setBody(newBody);
  [...]
}
```

Listing 5.7: Java, Typ einer Änderungsfunktion, javaMethodBodyOverriding

```
after(Cleaner cleaner): (call(void DIN.updateDupCountLNRef(
   long)) || call(void BIN.updateEntry(int,long))) && this(
   cleaner) {
  cleaner.nLNsMigrated++;
}
```

Listing 5.8: AspectJ, BerkeleyDB, Beispiel A

### 5.2.2 AspectJ Umsetzung

Studien zur Umsetzung von AspectJ sollten die Flexibilität der Quantifizierung bzw. der Modifikationen bemessen. Hierbei wurden aus der als AspectJ-Projekt refaktorisierten Berkeley DB ausgewählte homogene querschneidende Belange betrachtet und wenn möglich in Modifikationen konvertiert. Im Folgenden werden exemplarisch einige *pointcuts* und *advices* aufgeführt und wenn möglich provisorisch in Modifikationen umgesetzt.

#### Beispiel A

List. 5.8 kann nicht umgesetzt werden, da bei der Quantifizierung Methodenaufrufe innerhalb einer Methode gewählt werden müssten, was bei der Granularität des FST-Generators für Java nicht geht. Der Generator behandelt Methoden als geschlossene Konstrukte.

Denkbar wäre auch hier die Einführung einer weiteren Ausprägung der Änderungsfunktion, die genau diese Funktionalität bereitstellt. Dennoch ist es wohl Aufgabe der Quantifizierung und damit der Auswahlfunktion, die richtigen FST Knoten für die Än-

```
after(MemoryBudget mb): execution(void MemoryBudget.update*
   MemoryUsage(..)) && this(mb) {
  if (mb.getCacheMemoryUsage() > mb.cacheBudget) {
      mb.envImpl.alertEvictorDaemon();
  }
}
```

Listing 5.9: AspectJ, BerkeleyDB, Beispiel B

```
<modification>
  <type>superimposition</type>
  <traversal>..MemoryBudget.ClassDeclaration..
     updateMemoryUsage:MethodDecl</traversal>
  <content>
    <parsed>
      <plainText>
        <text>
          original();
          if (getCacheMemoryUsage() > cacheBudget) {
                envImpl.alertEvictorDaemon();
            }
        </text>
        <tType>java.method</tType>
      </plainText>
    </parsed>
  </content>
</modification>
```

Listing 5.10: Modifikation, BerkeleyDB, Beispiel B

derungsfunktion zu finden und wird deshalb als weniger sinnvoll betrachtet.

### Beispiel B

List. 5.9 kann umgesetzt werden als Modifikation in List. 5.10. Die bestimmten Methodenaufrufen vorangestellten Variablen aus dem Kopf des Advice werden weggelassen, da diese sich beim Einfügen in den Quellcode der Klassen bereits auf die richtigen Methoden beziehen.

### Beispiel C

Der pointcut in List. 5.11 kann als Auswahlfunkionsausdruck in List. 5.12 umgesetzt werden. An diesem letzten Beispiel kann man erkennen, wie ähnlich sich die beiden

```
pointcut assertLatched(IN in): (
  (execution(* IN.findParent(..)) && within(IN)
  || execution(* BIN.addCursor(..))
  || execution(* BIN.removeCursor(..))
  || execution(* BIN.adjustCursors(..))
  || execution(* BIN.adjustCursorsForInsert(..))
  || execution(* BIN.adjustCursorsForMutation(..))
  || execution(* BIN.evictLNs(..)))
  && this(in);
```

Listing 5.11: AspectJ, BerkeleyDB, Beispiel C

```
<traversal>
  ..IN.ClassDeclaration..findParent*:MethodDecl
  || ..BIN.ClassDeclaration..addCursor*:MethodDecl
  || ..BIN.ClassDeclaration..removeCursor*:MethodDecl
  || ..BIN.ClassDeclaration..adjustCursors*:MethodDecl
  || ..BIN.ClassDeclaration..adjustCursorsForInsert*:
     MethodDecl
  || ..BIN.ClassDeclaration..adjustCursorsForMutation*:
     MethodDecl
  || ..BIN.ClassDeclaration..evictLNs*:MethodDecl
</traversal>
```

Listing 5.12: Modifikation, BerkeleyDB, Beispiel C

Konzepte der pointcuts und der Quantifizierung sind.

### Fazit

Während der kompletten Studie wurden 16 homogene querschnittliche Belange betrachtet, sechs davon konnten umgesetzt werden. Die allermeisten nicht umsetzbaren könnten umgesetzt werden, wenn es eine Möglichkeit gäbe, die FST Granularität so anzuheben, dass Methoden nicht mehr als geschlossene Konstrukte, sondern beispielsweise als Aggregat aus einzelnen Deklarationen, Zuweisungen, Kontrollstrukturen usw. betrachtet werden könnten

# Abbildungsverzeichnis

# Listings

# Literaturverzeichnis


[AL08] Apel, Sven ; Lengauer, Christian: Superimposition: A Language-Independent Approach to Software Composition. In: Pautasso, Cesare (Hrsg.) ; Tanter, Éric (Hrsg.): *Software Composition* 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.): *AMAST* Bd. 5140, Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–79979–5, 36–50

[Ape08] Apel, Sven: *Moderne Programmier Paradigmen.* Vorlesung Universität Passau - Fakultät für Informatik und Mathematik, 2007/2008

[Bat05] Batory, Don S.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, J. H. (Hrsg.) ; Pohl, Klaus (Hrsg.): *SPLC* Bd. 3714, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–28936–4, 7–20

[LHA07] Lopez-Herrejon, Roberto E. ; Apel, Sven: Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies. In: Dwyer, Matthew B. (Hrsg.) ; Lopes, Antónia (Hrsg.): *FASE* Bd. 4422, Springer, 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–71288–6, 423–437

[Sne07] Snelting, Gregor: *Software Engineering.* Vorlesung Universität Passau - Fakultät für Informatik und Mathematik, WS2006/2007

# Enthaltene Software

Diese Arbeit beinhaltet eine CD-ROM die Folgendes beinhaltet:

- die entwickelte Software in Form eines kompletten workspace für die Integrierte Entwicklungsumgebung *eclipse*[3]
- eine digitale Version dieser Ausarbeitung im Format pdf
- ein Textdokument, das die Untersuchungen zur Umsetzbarkeit der AspectJ-Strukturen wiederspiegelt.

Die Startparameter für den FST Composer (Main class composer.FSTGenComposer) zur Betrachtung der Testfälle und Fallstudien in eclipse sind:

**einfache konstruierte Testfälle**

–expression examples/Modification/ConstructedSimpleTests/Out.expression

**GPL Java**

–expression examples/Modification/GPL/EdgeColoringTestOut.expression

**GPL C♯**

–expression examples/Modification/GPLCSharp/EdgeColoringTestOut.expression

**Berkeley DB**

–expression examples/Modification/BerkeleyDB/project.equation

[3] www.eclipse.org

# Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit 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 die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Boxleitner Stefan