Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik

Masterarbeit

Typsicherheit in Feature-orientierten Software-Produktlinien in FeatureIDE

Autor:

Sönke Holthusen

02. Juli, 2012

Betreuer:

Prof. Dr. habil. Gunter Saake
Institut für Technische und Betriebliche Informationssysteme, OvGU Magdeburg

Dipl.-Inform. Thomas Thüm
Institut für Technische und Betriebliche Informationssysteme, OvGU Magdeburg

Prof. Dr.-Ing. Ina Schaefer
Institut für Softwaretechnik und Fahrzeuginformatik, TU Braunschweig

Holthusen, Sönke:
Typsicherheit in Feature-orientierten Software-Produktlinien in FeatureIDE
Masterarbeit, Otto-von-Guericke-Universität Magdeburg, 2012.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung

Soll aus dem vorhandenen Quelltext einer Software ein ähnliches Produkt entwickelt werden, kann auf die *clone and own* Strategie zurückgegriffen werden [CN01]. Hier wird der vorhandene Quelltext kopiert und anschließend so verändert, dass das neue Produkt den geänderten Anforderungen des Entwicklers entspricht. Wird es nötig den von den verschiedenen Produkten gemeinsam genutzten Quelltext zu ändern, zum Beispiel weil ein Fehler zu beheben ist, muss diese Änderung für alle Produkte vorgenommen werden. Mit steigender Anzahl an Kopien des Quelltextes werden solche Änderungen zunehmend aufwendiger und die Wartung wird schwierig. Eine Möglichkeit dieses Problem der Wartbarkeit zu umgehen, ist die Verwendung von Software-Produktlinien. Eine Software-Produktlinie ist eine Familie von Software-Produkten, die auf einer Quelltextbasis basieren [PBL05]. Das Nutzen einer Software-Produktlinie macht den Quelltext leichter wartbar und ermöglicht das einfachere Hinzufügen von neuen Produkten zur Familie.

Eine Software-Produktlinie kann durch verschiedene Methodiken umgesetzt werden [CE00, AK09]. Eine ist die Feature-orientierte Programmierung (FOP) [Pre97]. Hier wird Funktionalität in Feature-Module gekapselt. Diese können, zum Beispiel für eine Objekt-orientierte Sprache, Klassen einführen und die Struktur von eingeführten Klassen verändern. Durch die Kombination verschiedener Feature-Module wird ein Produkt erzeugt.

Um die Entwicklung von Software-Produktlinien zu erleichtern und zu verbessern, ist es notwendig, automatische Analysen durchzuführen [TAK+12]. Ein Typchecker stellt dabei sicher, dass das Typsystem der zugrunde liegenden Programmiersprache umgesetzt wird [ALSU08, Pie02]. So wird zum Beispiel bei der Typprüfung ein Fehler gemeldet, wenn eine benötigte Klasse nicht vorhanden ist. Dieses Vorgehen schützt den Anwender vor Laufzeitfehlern. Durch die Möglichkeit Feature-Module zu kombinieren ergeben sich einige zusätzliche Fehlerquellen, die es zu beachten gilt. Für eine einzelne Variante ist es eindeutig, ob ein Typ vorhanden ist oder nicht. Fehlt eine verwendete Klasse, kann dies meist auf nicht eingebundene Bibliotheken oder einen Tippfehler zurückgeführt werden. In der Feature-orientierten Programmierung hingegen kann ein solcher Fehler zusätzlich noch durch eine nicht gültige

Kombination von Feature-Modulen verursacht werden. Wird eine Klasse nicht gefunden, kann dies ebenfalls an einer nicht eingebundenen Bibliothek oder einem Schreibfehler liegen. Zusätzlich kann auch eine fehlende Feature-Abhängigkeit für eine nicht erreichbare Klasse sorgen.

Formale Ansätze für Typsysteme für Feature-orientierte Software-Produktlinien existieren bereits [AKGL10]. Diese werden allerdings im Allgemeinen nur für eine funktionsreduzierte Programmiersprache, zum Beispiel auf Basis von Java, beschrieben. Für den Entwickler von Feature-orientierten Software-Produktlinien, zum Beispiel mit der Unterstützung von FeatureIDE, stellt sich also immer noch die Frage ob für jede gültige Konfiguration auch ein wohlgetyptes Produkt erzeugt wird. FeatureIDE[1] ist eine integrierte Entwicklungsumgebung zur Entwicklung von Software-Produktlinien mittels Feature-orientierter Programmierung [KTS+09].

Verfahren zur Analyse von Software-Produktlinien können in verschiedene Kategorien unterteilt werden [TAK+12]. Die am einfachsten umzusetzenden Analysen sind dabei *Produkt-basierte* Ansätze. Jedes Produkt einer Software-Produktlinie wird dabei erzeugt und die vorhandenen Test-Werkzeuge genutzt. Für die Typprüfung einer in Java geschriebenen Software-Produktlinie kann für eine Produkt-basierte Analyse zum Beispiel der normale Java-Compiler genutzt werden. Das Erzeugen und anschließende Kompilieren des Quelltextes ist allerdings sehr zeitaufwendig. Die Anzahl an Produkten wächst im schlechtesten Falle exponentiell mit der Anzahl der Features, was diesen Ansatz nicht skalieren lässt. Sinnvoll sind Produkt-basierte Ansätze zum Vergleich von Korrektheit, Vollständigkeit und Effizienz mit anderen Verfahren [TAK+12].

Eine weitere Kategorie von Analysen sind *Familien-basierte* Ansätze [TAK+12]. Diese nutzen das Wissen über gültige Kombinationen von Feature-Modulen und arbeiten nur mit den Feature-Modulen, nicht dem erzeugten Produkt-Quelltext. Da für einen Familien-basierten Ansatz die Features nicht für alle Produkte in denen sie verwendet werden überprüft werden müssen, verspricht dieser eine bessere Performance als ein Produkt-basierter Ansatz.

Im Zuge dieser Arbeit soll ein Familien-basierter Typchecker für FeatureIDE implementiert werden und die Performance mit der eines Produkt-basierten Ansatzes verglichen werden.

Gliederung der Arbeit

In Kapitel 2 werden die zum Verständnis dieser Arbeit benötigten Grundlagen eingeführt. Kapitel 3 behandelt das Konzept eines Typcheckers für Feature-orientierte Software-Produktlinien, während Kapitel 4 die Umsetzung dieses Konzeptes in einen Prototypen beschreibt. Die Evaluierung des Prototyps erfolgt in Kapitel 5. Verwandte Arbeiten werden in Kapitel 6 vorgestellt. Abgeschlossen wird diese Arbeit mit einer Zusammenfassung und einem Ausblick in Kapitel 7.

[1] http://fosd.net/featureide

2. Grundlagen

Dieses Kapitel beschreibt die für diese Arbeit benötigten Grundlagen. Hierfür werden Software-Produktlinien vorgestellt, auf Feature-orientierte Programmierung eingegangen und im letzten Abschnitt Grundlagen für Typsysteme und Typchecker erklärt.

2.1 Software-Produktlinien

Eine Software-Produktlinie (SPL) ist eine Menge von Software-Produkten, die aus einer gemeinsamen Quelltextbasis erzeugt werden [PBL05]. Produkte einer Software-Produktlinie unterscheiden sich dabei in der Auswahl der Features.

Feature

In der Literatur finden sich viele unterschiedliche Definitionen für Feature [AK09]. Eine der ersten Definitionen ist die Folgende von Kang et al. [KCH+90]:

> „a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems“

Features sind also herausstechende oder unterscheidbare Aspekte, Eigenschaften oder Charakteristika eines Software-Systems.

Je nach Anwendungsgebiet kann der Begriff Feature anders belegt sein [AK09]. In der Modellierung von Software-Produktlinien werden Features genutzt, um die Variabilität der Software-Produktlinie zu beschreiben und um die Kommunikation mit allen Beteiligten (Entwickler, Kunden, Vorgesetzten) zu vereinfachen. Eine solche Nutzung ist zunächst unabhängig von einer Implementierung.

Graphen Produktlinie

Um die Grundlagen zu verdeutlichen und im weiteren Verlauf die Beschreibung des Konzeptes dieser Arbeit zu erleichtern, wird eine vereinfachte Version der Graphen Produktlinie (GPL) eingeführt, die FeatureIDE[1] als Beispiel beiliegt. Die GPL ist eine SPL von Bibliotheken, die für die Arbeit mit Graphen entwickelt wurde. Jedes Produkt dieser SPL soll mit Kanten umgehen können, die entweder gewichtet oder ungewichtet, und entweder gerichtet oder ungerichtet sind. Außerdem werden verschiedene Algorithmen bereitgestellt, wie etwa das Durchnummerieren aller Knoten, das Finden von Zyklen und das Bestimmen des kürzesten Pfades zwischen zwei Knoten. Diese Anforderungen führen zu diversen Abhängigkeiten, die sich aus den Algorithmen ergeben. So werden zur Bestimmung der Länge eines Pfades gewichtete Kanten und zum Ermitteln von Zyklen gerichtete Kanten benötigt.

Nehmen wir an, dass eine Graphen-Bibliothek implementiert werden soll, die Knoten durchnummerieren, Zyklen erkennen und mit gewichteten Graphen arbeiten kann. Es steht keine entsprechende SPL zur Verfügung, es ist allerdings eine Bibliothek für ungewichtete, gerichtete Graphen vorhanden, die bereits eine Zyklenerkennung bereit stellt. Diese stellt durch *clone-and-own* die Grundlage der neuen Bibliothek. Nachdem die Entwicklung abgeschlossen ist, wird ein Fehler in der Zyklenerkennung festgestellt und muss in beiden Versionen des Quelltextes behoben werden. Mit jeder zusätzlichen Version wird die Pflege des Quelltextes aufwendiger und fehleranfälliger. Eine Möglichkeit diese Probleme zu umgehen ist es, die Software zu zerlegen und die Funktionalität in Features zu kapseln. Diese Features können dann um neue Funktionalität erweitert werden, indem diese ebenfalls als Feature zur Verfügung gestellt wird.

Um die Spezifikationen der GPL in eine Software-Produktlinie zu überführen, bietet es sich an, die unterschiedlichen Algorithmen als jeweils ein Feature, Shortest, Number und Cycle (siehe Tabelle 2.1) zu modellieren. Die Kanten-Eigenschaften *gerichtet* und *gewichtet* können als jeweils zwei Features gesehen werden. Dies führt zu den insgesamt vier Features Directed, Undirected, Weighted und Unweighted. Für die Basis-Funktionalität, die unabhängig von den Kanten und den Algorithmen ist, ist es notwendig, noch ein weiteres Feature Graph hinzuzufügen. Tabelle 2.1 fasst die Features, die sich aus der Anforderung ergeben, zusammen.

Diese Menge an Features alleine reicht allerdings noch nicht aus um eine Software-Produktlinie zu beschreiben, da sich Features in Kombination, hier zum Beispiel ungerichteter und gerichteter Graph, gegenseitig ausschließen.

Feature-Modell

Um die Software-Produktlinie zu beschreiben, wird ein *Feature-Modell* benötigt. Das Feature-Modell ist eine hierarchische Repräsentation der Software-Produktlinie, in der alle Features in einem Baum dargestellt werden [Bat05]. Das Feature-Modell beschreibt die gültigen *Varianten* (auch *Produkte*) einer SPL. Jedes Feature, ausgehend von der Wurzel, kann dabei Kindfeatures haben, die in unterschiedlichen Beziehungen zum Elternfeature stehen und Einschränkungen in der Feature-Auswahl ermöglichen [Bat05]. Tabelle 2.2 beschreibt die verschiedenen Arten von Verbindungen zwischen Eltern- und Kindfeatures.

[1] http://fosd.net/featureide

Feature-Name	Beschreibung
Graph	Die Basis jeder Graphen-Bibliothek
Directed	gerichtete Kanten
Undirected	ungerichtete Kanten
Weighted	gewichtete Kanten
Unweighted	ungewichtete Kanten
Number	Algorithmus zur Nummerierung von Knoten
Shortest	Algorithmus zur Bestimmung des kürzesten Pfades
Cycle	Algorithmus zur Zyklenerkennung

Tabelle 2.1: Beschreibung der einzelnen Features der GPL

Verbindung	Bedeutung
Obligatorisch	K muss gewählt sein
Optional	K kann gewählt sein
Oder	eine beliebige Anzahl an Kindfeatures K_i kann gewählt werden
Alternative	genau ein Kindfeature K_i ist auszuwählen

Tabelle 2.2: Bedeutung der Verbindungen zwischen Elternfeature E und Kindfeatures K_i wenn E gewählt ist [Bat05]

Für die graphische Repräsentation eines Feature-Modells einer Software-Produktlinie kann ein Feature-Diagramm genutzt werden [KCH+90]. Abbildung 2.1 zeigt das Feature-Diagramm der Graphen Produktlinie. Das Feature Graph wird immer benötigt, auch eine Art von Kante ist notwendig. Dabei kann zwischen den vier möglichen Kombinationen

- gewichtet und gerichtet
- gewichtet und ungerichtet
- ungewichtet und gerichtet
- ungewichtet und ungerichtet

gewählt werden. Die Wahl von Algorithmen ist optional.

Die Verbindung zwischen Weight und den Kindfeatures Weighted und Unweighted ist eine *Alternative*. Es kann nur eines der Kindfeatures gewählt werden. Die *Oder*-Verbindung des Features Algorithms zu den einzelnen Kindfeatures erlaubt es, einen oder mehr der zur Verfügung stehenden Algorithmen zu nutzen. Die Verbindung zu den Features Graph und Edges ist *Obligatorisch*, diese Features müssen gewählt werden. Die *Optional*-Verbindung zwischen dem Wurzelfeature und Algorithms ermöglicht es dieses Feature zu wählen, es muss aber nicht ausgewählt sein.

Für weitere Einschränkungen, die nicht, oder nur sehr schwer, in dem Feature-Modell darstellbar sind, können dem Feature-Modell *Constraints* oder *Zusatzbedingungen*

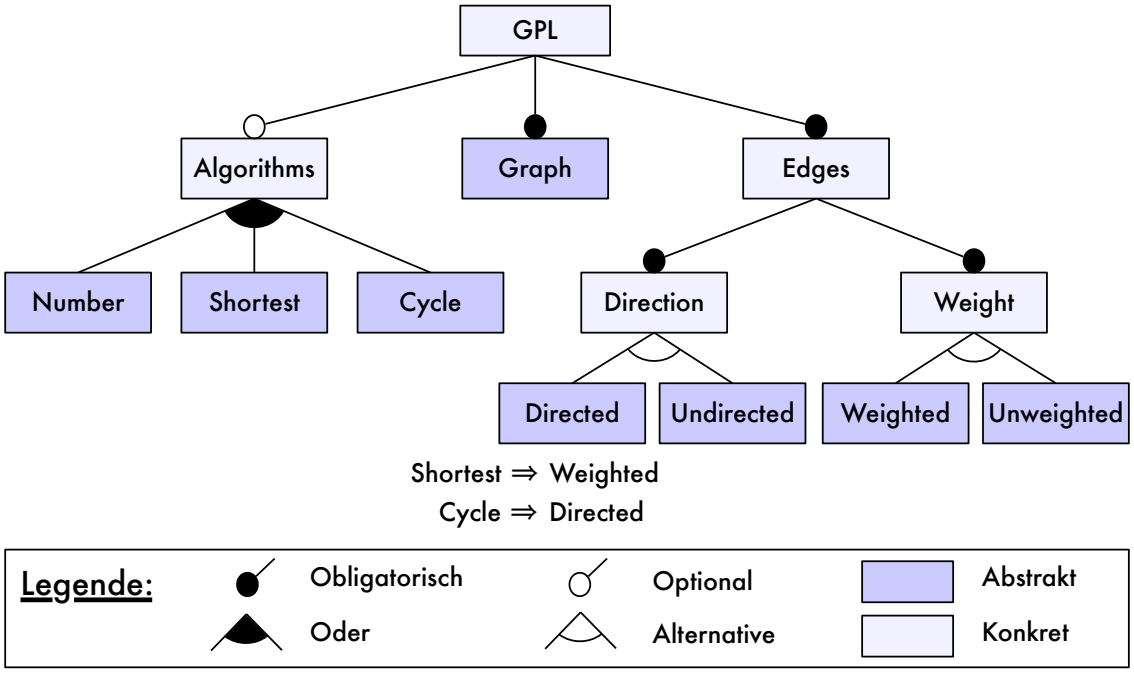


Abbildung 2.1: Ein Feature-Diagramm der Graphen Produktlinie

in Form eines aussagenlogischen Ausdrucks hinzugefügt werden [Bat05]. So wird mit $Shortest \Rightarrow Weighted$ gefordert, dass die Wahl von Feature *Shortest* eine Wahl von Feature *Weighted* impliziert. Dies macht es in Abbildung 2.1 notwendig, bei der Wahl des Features für die Zyklenerkennung, auch das Feature für einen gerichteten Graphen zu wählen. In einem Feature-Diagramm werden Constraints ebenfalls aufgelistet.

Es wird zwischen zwei verschiedenen Arten von Features unterschieden [TKES11]. Zum einen gibt es *konkrete* Features, denen eine Implementierung zu Grunde liegt. Der zweite Typ von Features sind *abstrakte* Features, die hauptsächlich der einfacheren Modellierung und Strukturierung dienen und bei der Feature-Auswahl ignoriert werden. Ohne die explizite Nutzung von abstrakten Features kann es gültige, unterschiedliche Mengen von Features geben, aus denen das gleiche Produkt erzeugt wird. Abbildung 2.1 zeigt, dass zur einfacheren Modellierung die Features GPL, Edges, Weight, Direction und Algorithms abstrakt sind. Die Unterscheidung der verschiedenen, möglichen Kantentypen wird so deutlich vereinfacht.

Ein Feature-Modell kann als eine Menge von Mengen von Features dargestellt werden [TKES11]. Jede dieser Mengen stellt eine gültige Feature-Kombination da. Eine beliebige Untermenge von konkreten Features wird als *Konfiguration* bezeichnet. Eine Konfiguration ist gültig, wenn sie Element des Feature-Modells ist. Aus einer gültigen Konfiguration lässt sich eine *Variante* erzeugen, die einem Produkt der Software-Produktlinie entspricht.

Aussagenlogischer Ausdruck

Eine weitere Möglichkeit ein Feature-Modell zu beschreiben sind *aussagenlogische Ausdrücke* [Man02]. Hierbei wird für jede Verbindung von Knoten des Feature-

Feature-Modell	aussagenlogischer Ausdruck
Optionales Feature C_i	$C_i \Rightarrow P$
Obligatorisches Feature C_i	$(C_i \Rightarrow P) \wedge (P \Rightarrow C_i)$
Oder Gruppe	$P \Leftrightarrow V_{1 \leq i \leq n} C_i$
Alternative Gruppe	$(P \Leftrightarrow V_{1 \leq i \leq n} C_i) \wedge \bigwedge_{i<j} (\neg C_i \vee \neg C_j)$

Tabelle 2.3: Übersetzung von Verbindungen im Feature-Modell zu aussagenlogischen Ausdrücken. P ist dabei ein Elternfeature mit den Kindfeatures $C_1, C_2, ..., C_n$ [TKES11]

$$
\begin{aligned}
&(G \Rightarrow GPL) \wedge (GPL \Rightarrow G) \wedge (A \Rightarrow GPL) \wedge (A \Leftrightarrow C \vee S \vee N) \\
&\wedge (E \Rightarrow GPL) \wedge (GPL \Rightarrow E) \wedge (W \Leftrightarrow WW \vee UW) \wedge (\neg WW \vee \neg UW) \\
&\wedge (D \Leftrightarrow DD \vee UD) \wedge (\neg DD \vee \neg UD) \wedge (S \Rightarrow WW) \wedge (C \Rightarrow DD) \wedge GPL
\end{aligned}
$$

Abbildung 2.2: Das Feature-Modell der GPL aus Abbildung 2.1 als aussagenlogischer Ausdruck nach Tabelle 2.3. G = Graph, A = Algorithms, C = Cycle, S = Shortest, N = Number, E = Edges, W = Weight, WW = Weighted, UW = Unweighted, D = Direction, DD = Directed, UD = Undirected

Modells mit Hilfe der in Tabelle 2.3 gezeigten Regeln in aussagenlogische Ausdrücke übersetzt. Diese werden durch eine Konjunktion verbunden und durch die Constraints und den Wurzelknoten des Feature-Modells ergänzt. Wird dieses Vorgehen auf die GPL (Abbildung 2.1) angewendet, ergibt sich der in Abbildung 2.2 dargestellte aussagenlogische Ausdruck.

Liegt das Feature-Modell in einem solchen aussagenlogischen Ausdruck vor, können verschiedene Anfragen gestellt werden, die mit Hilfe eines SAT-Solvers ausgewertet werden. Ein SAT-Solver (von englisch satisfiability = Erfüllbarkeit) prüft, ob für einen aussagenlogischen Ausdruck eine Belegung existiert, für die der Ausdruck wahr wird.

Die Überprüfung der Gültigkeit einer Konfiguration kann mit Hilfe eines SAT-Solvers geprüft werden. Dabei werden alle gewählten Features mit dem aussagenlogischen Ausdruck des Feature-Models durch Konjugationen verknüpft und dieser Gesamtausdruck mit dem SAT-Solver auf Erfüllbarkeit geprüft. Dabei versucht der SAT-Solver eine Belegung zu finden, für die der Ausdruck gültig ist. Eine mögliche Konfiguration wäre Graph, Unweighted, Directed, Cycle, Shortest, die zu einem Ausdruck ($FM \wedge$ Graph $\wedge$ Unweighted $\wedge$ Directed $\wedge$ Cycle $\wedge$ Shortest) für den SAT-Solver führt. Dieser findet keine gültige Belegung für diesen Ausdruck und gibt zurück, dass der Ausdruck nicht erfüllbar ist. Ein Blick auf das Feature-Modell zeigt, dass das Feature Shortest das Feature Weighted benötigt, diese Bedingung in der gegebenen Konfiguration aber nicht erfüllt ist. Der SAT-Solver ist allerdings nicht in der Lage herauszufinden, warum eine Konfiguration nicht gültig ist. Eine gültige Konfiguration lässt sich aus dem einleitenden Beispiel ableiten: Base, Unweighted, Directed, Cycle.

Soll festgestellt werden, ob eine Tautologie vorliegt, das heißt, dass ein aussagenlogischer Ausdruck für jede Belegung *wahr* ist, darf *keine* Belegung existieren, für die die Negation des Ausdruckes erfüllbar ist. Existiert eine Belegung, für die die Negation erfüllbar ist, liegt keine Tautologie vor. Das heißt, ein Prädikat $taut(X)$, welches feststellen soll, ob der aussagenlogische Ausdruck X eine Tautologie ist, prüft ob $\neg X$ erfüllbar, und damit ob $SAT(\neg X)$ *wahr* ist. Ist dies der Fall, liegt keine Tautologie vor. Existiert keine Belegung, für die $\neg X$ erfüllbar ist, das heißt $SAT(\neg X)$ ist *falsch*, liegt eine Tautologie vor. Das Ergebnis der Erfüllbarkeit muss also negiert werden. Daraus folgt $(taut(X) \Leftrightarrow \neg SAT(\neg X))$.

Soll weiterhin festgestellt werden, ob ein aussagenlogischer Ausdruck, unter der Annahme, dass das Feature-Modell gilt, immer erfüllbar ist, kann das gerade eingeführte Prädikat $taut(X)$ genutzt werden. Ist zu prüfen, ob das Feature NUMBER immer gewählt ist, wenn auch das Feature CYCLE gewählt ist, muss festgestellt werden, ob der Ausdruck $FM \Rightarrow (\text{CYCLE} \Rightarrow \text{NUMBER})$ eine Tautologie ist, also ob $taut(FM \Rightarrow (\text{CYCLE} \Rightarrow \text{NUMBER}))$ beziehungsweise ob $\neg SAT(\neg(FM \Rightarrow (\text{CYCLE} \Rightarrow \text{NUMBER})))$ *wahr* ist. Ist dieser Ausdruck *wahr*, ist das Feature NUMBER immer gewählt wenn das Feature CYCLE gewählt ist. Ist der Ausdruck *falsch*, gibt es mindestens eine Belegung, für die diese Annahme nicht gilt.

Implementierungen

Ist eine Software-Produktlinie modelliert, geht es an die Implementierung. Hier stehen verschiedene Verfahren zur Verfügung [AK09]. Angefangen von dem bedingten Kompilieren durch Annotation, zum Beispiel mit Präprozessor-Anweisungen (#ifdef), über Software-Frameworks [JF88] und Komponenten [Szy02], zu Aspekt- [KLM+97], Feature- [Pre97] und Delta-orientierter Programmierung [SBB+10]. In jedem Fall findet eine Zuordnung von Features zu Implementierungseinheiten statt. Hierunter fallen zum Beispiel in der bedingten Kompilierung die #ifdefs, in der Aspekt-orientierten Programmierung die Aspekte und in der Feature-orientierten Programmierung die Feature-Module. Da sich diese Arbeit mit einem Typchecker für eine Feature-orientierte Produktlinie beschäftigt, wird im Weiteren näher auf dieses Verfahren eingegangen.

2.2 Feature-orientierte Programmierung

Feature-orientierte Programmierung (FOP) ist eine Möglichkeit, Software variabel zu gestalten. Dies geschieht durch die Implementierung von Funktionalität in Feature-Modulen und der anschließenden Generierung eines Programms aus diesen Modulen. Es gibt verschiedene Werkzeuge, die FOP unterstützen, darunter sind AHEAD und FeatureHouse [BSR03, AKL09]. Feature-Module in AHEAD werden in JAK und die von FeatureHouse in einer modifizierten Variante von Java geschrieben. Beide Werkzeuge sind in der Lage gültigen Java-Quelltext zu erzeugen. Die Programmiersprache, zu der die Feature-Module kombiniert werden, wird Basissprache (engl. *host language*) genannt und die Werkzeuge Composer. AHEAD und FeatureHouse unterstützen auch weitere Basissprachen für Feature-orientierte Programmierung. Im weiteren Verlauf wird FOP allerdings, aufgrund des Schwerpunktes dieser Arbeit, mit Hilfe von FeatureHouse und Java erklärt.

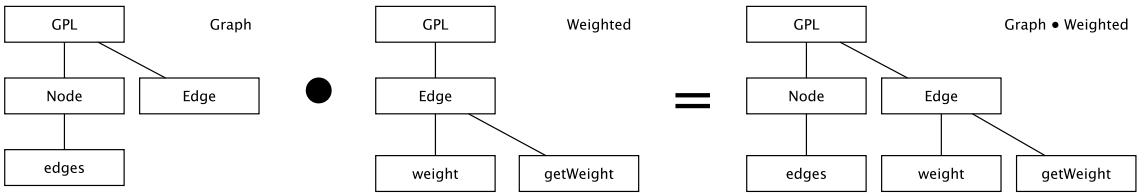


Abbildung 2.3: Superimposition von Feature-Strukturbäumen der in Abbildung 2.4 eingeführten Features mit Hilfe des Superimpositionsoperators (•)

Feature-Modul

Ein *Feature-Modul* ist das Konstrukt, in das die Funktionalität in Feature-orientierter Programmierung gekapselt wird. In Verbindung mit einer Objekt-orientierten Basissprache kann ein Feature-Modul Klassen einführen oder die Klassenstruktur bereits eingeführter Klassen verändern. Hierunter fallen das Hinzufügen oder Überschreiben von Methoden und das Hinzufügen von Feldern. Die Menge der Klasseneinführungen und Verfeinerungen eines Feature-Moduls wird als *Kollaboration* bezeichnet. Ein Element einer Kollaboration ist die *Rolle*, die eine Klasse in einem Feature inne hat. Der Nutzer wählt eine Menge von Feature-Modulen aus und der Quelltext des Produktes wird generiert. Dieser Vorgang des Anwendens von Feature-Modulen wird komponieren genannt. Der komponierte Quelltext kann anschließend mit einem Standard-Compiler kompiliert werden.

Jedes Feature kann als Feature-Strukturbaum dargestellt werden. Der Wurzelknoten eines solchen Baumes besteht dabei aus der Packetstruktur des Feature-Moduls. Die Kindknoten sind die eingeführten und erweiterten Klassen, die wiederum eingeführte und erweiterte Methoden und eingeführte Felder als Kindknoten besitzen. FeatureHouse bietet die Möglichkeit, Features durch die Superimposition von Feature-Modulen zu komponieren [AKL09]. Um zwei Feature-Module durch Superimposition zu komponieren, werden die Bäume der einzelnen Feature-Module übereinander gelegt. Dabei werden Klassen kombiniert und Methoden überschrieben. Abbildung 2.3 zeigt die Superimposition zweier Feature-Module der GPL. Wenn es nötig ist, eine Version einer Methode vor ihrer Erweiterung aufzurufen, kann dies durch das Schlüsselwort `original()` geschehen. Das Schlüsselwort `original()` verhält sich dabei ähnlich dem aus der Objekt-orientierten Programmierung bekannten Schlüsselwort `super()`. Das Schlüsselwort `super()` ruft die gleiche Methode der Superklasse auf, während `original()` die Version der Methode aufruft, bevor diese vom aktuellen Feature verändert wurde. Von dem Schlüsselwort `original()` abgesehen, handelt es sich bei FeatureHouse-kompatiblem Quelltext um Standard-Java.

Der Quelltext Abbildung 2.4 zeigt einen Ausschnitt des Quelltextes der Features Graph und Weighted der GPL. In Graph werden die zwei Klassen `Node` zur Repräsentation der Knoten und `Edge` für die Kanten eingeführt. Die Klasse `Edge` hat zu diesem Zeitpunkt noch keine Informationen darüber, mit was für einem Graphen, bezogen auf die Kanten, letztendlich gearbeitet wird. Die Klasse `Edge` wird durch das Feature Weighted um Felder und Methoden erweitert, die es möglich machen mit einem gewichteten Graphen zu arbeiten.

GRAPH

```
1  package GPL;
2  public class Node {
3      private Edge[] edges;
4  }
5
6  public class Edge {
7  }
```

WEIGHTED

```
8  package GPL;
9  public class Edge {
10     private int weight;
11     public int getWeight(){return weight;}
12 }
```

Abbildung 2.4: Feature-Module der GPL

```
1  public class Edge extends Objekt{
2      int x = 10.0;
3      public Edge(int x){
4          weight = x;
5          initialize(x);
6      }
7      private void initialize(){ /*...*/ }
8  }
```

Abbildung 2.5: Beispiel für durch ein Typsystem in Java erkannte Fehler

Feature-orientierte Programmierung alleine stellt noch keine Software-Produktlinie dar. Wendet man die in Abschnitt 2.1 vorgestellten Techniken zu Software-Produktlinien an, ergibt sich eine Feature-orientierte Software-Produktlinie. Hierfür müssen vor allem die Features der SPL den Feature-Modulen der Feature-orientierten Programmierung zugeordnet werden.

2.3 Typsysteme

Ein Typsystem ist eine Menge an Inferenz-Regeln, mit denen sichergestellt werden kann, dass ein bestimmtes Verhalten eines Programms beim Ausführen nicht auftritt [ALSU08]. Der Quelltext wird in Typausdrücke zerlegt und es wird geprüft, ob sich alle Ausdrücke aus den Regeln des Typsystems ableiten lassen können. Das Werkzeug, das die Einhaltung dieser Regeln überprüft, wird als Typchecker bezeichnet [Pie02].

Ein Typsystem ist eine statische Analysemethode. Im Gegensatz zu dynamischen Analysen werden statische Analysen nicht zur Laufzeit vorgenommen und arbeiten, wie im Beispiel eines Typsystems, auf dem Quelltext eines Programms. Ein Programm ist wohlgetypt, wenn es allen Regeln des Typsystems entspricht und somit die Typprüfung ohne Fehler besteht [Pie02].

Zur Verdeutlichung werden im Quelltext in Abbildung 2.5 einige Fehler aufgezeigt, die ein Typchecker für Java findet. Der erste Fehler ist bei der verwendeten Superklasse zu finden. In der Zeile 1 wird diese `Objekt` statt `Object` geschrieben. Das

Typsystem stellt sicher, dass jede verwendete Klasse auch vorhanden ist. Der zweite Fehler ist in Zeile 2 zu finden. Es wird versucht einer Variablen vom Typ `int` einen Wert vom Typ `float` zuzuweisen. Das Typsystem stellt fest, dass nicht automatisch von `int` nach `float` gecastet werden kann und meldet einen Fehler. Der dritte Fehler zeigt sich in Zeile 4. Es wird versucht, der nicht deklarierten Variablen `weight` einen Wert zuzuweisen. Das Typsystem stellt sicher, dass alle Variablen, auf die zugegriffen wird auch vorhanden sind. Ein letzter Fehler kann in Zeile 5 gefunden werden. Es wird eine Methode `initialize(int)` aufgerufen. Es existiert zwar eine Methode mit dem gleichen Namen, die Signatur unterscheidet sich allerdings. Das Typsystem prüft, ob alle aufgerufenen Methoden auch vorhanden sind.

3. Typchecker für Feature-orientierte Software-Produktlinien

In diesem Kapitel wird ein Konzept für einen Typchecker für Feature-orientierte Software-Produktlinien beschrieben. Dazu wird zunächst auf verschiedene Arten von Fehlern eingegangen, die in einer Software-Produktlinie auftreten können, und ein formales Typsystem für Feature Featherweight Java betrachtet. Im Weiteren wird auf das eigentliche Konzept eines Typcheckers eingegangen und die verschiedenen Phasen beschrieben.

3.1 Fehler in Software-Produktlinien

In diesem Abschnitt werden verschiedene Fehler beschrieben, die in Software-Produktlinien auftreten können. Dies sind zum einen Composerfehler, die zu Fehlern beim Komponieren mehrerer Feature-Module führen. Weiterhin wird auf Compilerfehler eingegangen, die auch von dem Typchecker der Basissprache erkannt werden können.

3.1.1 Composerfehler

Composerfehler sind Fehler, die das Komponieren einzelner Feature-Module zu einer Variante beeinflussen oder verhindern. Dies kann durch Syntaxfehler im Quelltext oder durch zueinander inkompatiblen Feature-Modulen passieren.

Syntaxfehler

Unterschiedliche Composer erwarten eine unterschiedliche Syntax für den der SPL zugrunde liegenden Quelltext. So erwartet AHEAD JAK-Dateien und FeatureHouse benötigt nur leicht angepassten Java Quelltext (siehe Abschnitt 2.2). Entspricht der Quelltext nicht dieser Syntax (siehe Abbildung 3.1), können die Feature-Module nicht komponiert werden und es kommt zu einem Fehler.

Da es sich um Fehler in der Syntax handelt, kann diese Art von Fehlern durch ein Modifizieren des Quelltextes behoben werden.

```
1 public class Node
2     public Node(){ /* ... */ }
3 }
```

Abbildung 3.1: Beispiel für einen Syntaxfehler: Der Quelltext kann nicht geparst werden, da die geschweifte Klammer nach dem Klassennamen fehlt.

DIRECTED
```
1 class Edge {
2     int getX(){ /* ... */ }
3 }
```

UNDIRECTED
```
4 class Edge {
5     double getX(){ /* ... */ }
6 }
```

Abbildung 3.2: Beispiel für inkompatible Feature-Module durch inkompatible Rückgabewerte.

Inkompatible Feature-Module

Verschiedene Fehler können dazu führen, dass einzelne Feature-Module inkompatibel zueinander werden.

Bei der Konzipierung einer Software-Produktlinie kann es sinnvoll sein, in alternativen Features Methoden mit gleicher Signatur, verschiedene Rückgabetypen zu geben. Erwartet ein Feature-Modul einen bestimmten Rückgabetyp von einer Methode, kommt es durch das Komponieren mit einem Feature-Modul, in dem diese Methode einen anderen Rückgabetyp hat, zu einem Fehler.

Der Quelltext in Abbildung 3.2 demonstriert, wie es zu inkompatiblen Features durch inkompatible Rückgabewerte kommen kann. Feature-Modul DIRECTED führt in der Klasse `Edge` eine Methode `getX()` mit dem Rückgabetyp `int` ein. Feature-Modul UNDIRECTED führt in der Klasse `Edge` eine Methode `getX()` mit `double` als Rückgabetyp ein. Werden UNDIRECTED und DIRECTED komponiert, kommt es zu einem Fehler, da versucht wird, eine Methode mit dem Rückgabetyp `int` mit einer Methode mit dem Rückgabetyp `double` zu überschreiben.

Ein ähnliches Problem ergibt sich mit Feldern. Wird in zwei unterschiedlichen Feature-Modulen ein Feld mit gleichem Namen, aber von unterschiedlichem Typ eingeführt, führt das Komponieren dieser Feature-Module zu einem Fehler.

In FEATUREHOUSE kann ein Feld zwar mehrfach mit dem gleichen Typ definiert, aber nur mit einem Wert initialisiert werden. Im Gegensatz zu Methoden ist es nicht möglich ein Feld einfach zu überschreiben. Ein mehrfaches Initialisieren eines Feldes mit unterschiedlichen Werten in einer Kombination von Feature-Modulen führt zu einem Fehler. Diese beiden Fehler werden im Quelltext in Abbildung 3.3 veranschaulicht. Die Feature-Module DIRECTED und UNDIRECTED führen jeweils ein Feld mit dem Namen `x` ein, allerdings mit unterschiedlichen Typen. Ein Komponieren dieser beiden Feature-Module führt zu einem Fehler.

Erbt eine Klasse in verschiedenen Feature-Modulen von unterschiedlichen Superklassen, ergibt sich ebenfalls eine Situation, in der die betroffenen Feature-Module

```
1 class Edge {                                                    DIRECTED
2     int x = 5;
3 }
```

```
4 class Edge {                                                    UNDIRECTED
5     double x = 4.5;
6 }
```

Abbildung 3.3: Beispiel für inkompatible Feature-Module durch inkompatible Feld-Deklarationen

```
1 class Edge extends ArrayList {}                                 DIRECTED
```

```
2 class Edge extends LinkedList {}                                UNDIRECTED
```

Abbildung 3.4: Beispiel für inkompatible Feature-Module durch Erben von unterschiedlichen Klassen in unterschiedlichen Feature-Modulen

nicht kompatibel sind. Der Quelltext Abbildung 3.4 veranschaulicht dieses Problem. Die Feature-Module DIRECTED und UNDIRECTED definieren eine Klasse `Edge`, die jeweils eine andere Superklasse erweitert. Werden beide Features zusammen gewählt, kann es zu einem Fehler kommen, wenn auf Methoden oder Felder zugegriffen wird, die in der einen Superklasse vorhanden sind, in der anderen aber nicht.

Es lassen sich folgende Gründe für inkompatible Features zusammenfassen:

- Methoden einer Klasse mit gleicher Signatur, aber unterschiedlichem Rückgabetyp, deklariert in verschiedenen Feature-Modulen
- Felder einer Klasse mit gleichem Namen und Typ, die in verschiedenen Feature-Modulen mit einem anderen Wert initialisiert sind
- Felder einer Klasse mit gleichem Namen aber unterschiedlichem Typ, deklariert in verschiedenen Feature-Modulen
- Zwei Rollen einer Klasse erben in verschiedenen Feature-Modulen von verschiedenen Superklassen

3.1.2 Compilerfehler

Compilerfehler sind Fehler, die der Compiler der Basissprache zusätzlich zu den Composerfehlern erkennen würde, wenn alle Produkte generiert und geprüft werden (bei Java zum Beispiel *javac*). Diese beinhalten Fehler, die der Typchecker des Compilers findet und Syntaxfehler. Da Syntaxfehler bereits beim Komponieren gefunden werden (siehe auch Abschnitt 3.1.1), werden diese in dieser Arbeit nicht als Compilerfehler betrachtet. Compilerfehler erhalten durch das Komponieren von Features in einer Software-Produktlinie eine neue Dimension, da es passieren kann, dass diese nur in bestimmten Feature-Kombinationen auftreten.

Im Weiteren wird genauer auf Fehler im Zusammenhang mit Superklassen, Typpreferenzen sowie Methoden- und Feldzugriffen eingegangen.

```
1 class Edge { /* ... */}                                    BASE
2 class DirectedEdge extends Edge { /* ... */ }              DIRECTED
```

Abbildung 3.5: Beispiel für Compilerfehler durch Vererbung

Superklassen

Eine Eigenschaft objektorientierter Programmierung ist die Möglichkeit der Vererbung. Hierbei werden die Eigenschaften einer Klasse in eine neue Klasse übernommen und den Bedürfnissen des Entwicklers angepasst. Hierzu gehört das Hinzufügen von Feldern und Methoden, sowie das Erweitern und Überschreiben von Methoden. Durch das Komponieren von Feature-Modulen kann es passieren, dass eine Superklasse nicht in allen Varianten verfügbar ist.

In Abbildung 3.5 wird im Feature-Modul BASE eine Klasse `Edge` und im Feature-Modul DIRECTED die Klasse `DirectedEdge` eingeführt. `DirectedEdge` erbt von `Edge`. Wird das Feature-Modul DIRECTED ohne das Feature-Modul BASE gewählt, kommt es zu einem Fehler, da die Superklasse der Klasse `DirectedEdge` nicht vorhanden ist. Da dieser Fehler in einer beliebigen Variante auftreten kann, müsste zum Erkennen solcher Fehler jede Variante erzeugt und geprüft werden.

Typreferenzen

Der Fehler einer fehlenden Superklasse lässt sich auf Typreferenzen verallgemeinern. Jeder Typ, auf den in einem Feature-Modul zugegriffen wird, muss in jeder Variante, die das Feature-Modul enthält, deklariert sein. Dies führt zu einer Reihe von Fehlerquellen im Bezug auf Typzugriffe:

- Nicht vorhandene Superklassen (siehe Abschnitt 3.1.2) oder Interfaces
- Nicht vorhandener Typ bei Deklaration eines Feldes oder einer Variablen
- Nicht vorhandener Rückgabe- oder Parametertyp einer Methode

Abbildung 3.6 gibt einige Beispiele für Typreferenzfehler. Das Feature-Modul GRAPH führt eine Klasse `Node` ein, die ein Feld vom Typ `Edge` deklariert und eine Methode, die den Typ `Edge` als Rückgabetyp hat. Die Klasse `Edge` wird im Feature-Modul WEIGHTED eingeführt. Ist das Feature-Modul GRAPH ohne WEIGHTED ausgewählt, kommt es zu einem Compilerfehler, da die Klasse `Edge` nicht erreichbar ist. Das Feature-Modul SHORTEST führt eine Klasse `Shortest` ein, die in ihrem Konstruktor auf das statische Feld vom Typen `Edge` zurückgreift. Hier kommt es zu einem Compilerfehler, wenn das Feature-Modul SHORTEST ohne das Feature-Modul WEIGHTED ausgewählt ist, da die Klasse `Edge` ebenfalls nicht erreichbar ist.

Methoden- und Feldzugriffe

Wird in einem Programm eine nicht deklarierte Methode aufgerufen oder auf ein nicht deklariertes Feld zugegriffen, führt dies zu einem Compilerfehler. Auch jede aus

GRAPH
```
1  public class Node {
2      Edge[] edges;
3      public Edge[] getEdges(){ return edges; }
4  }
```

WEIGHTED
```
5  public class Edge {
6      int weight;
7      getWeight(){ return weight; }
8      public static int defaultWeight = 1;
9  }
```

SHORTEST
```
10 public class Shortest {
11     Shortest(){
12         int defaultWeight = Edge.defaultWeight;
13     }
14 }
```

Abbildung 3.6: Beispiel für Compilerfehler durch nicht vorhandene Typen

Feature-Modulen erzeugte Variante einer Software-Produktlinie ist ein Programm in der Basissprache, für das diese Eigenschaft gilt. Ein Typchecker für eine SPL hat also sicherzustellen, dass alle Methoden und Felder, auf die in einem Feature-Modul zugegriffen werden, auch in jeder gültigen Variante mit diesem Feature-Modul deklariert sind.

Abbildung 3.7 zeigt ein Beispiel für nicht vorhandene Methoden. Das Feature-Modul GRAPH führt eine Klasse `Edge` ein, die durch das Feature-Modul WEIGHTED um Funktionalität für einen gewichteten Graphen erweitert wird. Das Feature-Modul SHORTEST greift auf diese Funktionalität zurück. Wird SHORTEST ohne WEIGHTED gewählt, kommt es zu einem Compilerfehler, da die Methode `getWeight()` der Klasse `Edge` nicht vorhanden ist.

Der Quelltext in Abbildung 3.8 zeigt ein Beispiel für einen Compilerfehler durch ein nicht vorhandenes Feld. In den Feature-Modulen GRAPH und WEIGHTED wird die Klasse `Edge` eingeführt, beziehungsweise um die Funktionalität eines gewichteten Graphen erweitert. Das Feature-Modul SHORTEST greift auf das Feld `weight` der Klasse `Edge` zu. Damit das Feld `weight` vorhanden ist, muss bei einer Wahl von SHORTEST auch das Feature-Modul WEIGHTED gewählt sein. Ist dies nicht der Fall, kommt es zu einem Compilerfehler, da das Feld `weight` nicht vorhanden ist.

3.1.3 Korrekturmaßnahmen

Um die vorgestellten Fehler zu beheben, werden hier zwei Maßnahmen vorgeschlagen. Zum einen kann der Quelltext des Feature-Moduls angepasst werden, zum anderen das Feature-Modell verändert werden. Auch eine Kombination kann eine mögliche Lösung für einen Fehler sein.

Einen Syntaxfehler, wie die fehlende, geschweifte Klammer in Abbildung 3.1 lässt sich nur durch das Hinzufügen der Klammer, und damit durch eine Änderung im Quelltext, beheben. Eine Änderung des Feature-Modells hilft bei einem solchen Fehler nicht.

```
1  public class Edge {}                                                    GRAPH
```

```
2  public class Edge {                                                     WEIGHTED
3      int weight;
4      int getWeight(){ return weight; }
5  }
```

```
6  public class Shortest {                                                 SHORTEST
7      Shortest(){
8          /* ... */
9          int weight = edge.getWeight();
10         /* ... */
11     }
12 }
```

Abbildung 3.7: Beispiel für Compilerfehler bei nicht vorhandenen Methoden

```
1  public class Edge {}                                                    GRAPH
```

```
2  public class Edge {                                                     WEIGHTED
3      public int weight;
4  }
```

```
5  public class Shortest {                                                 SHORTEST
6      Shortest(){
7          /* ... */
8          int weight = edge.weight;
9          /* ... */
10     }
11 }
```

Abbildung 3.8: Beispiel für Compilerfehler bei nicht vorhandenen Feldern

Inkompatible Features lassen sich am einfachsten durch eine Änderung am Feature-Modell beheben. Ein Hinzufügen des Constraints (DIRECTED ⇔ ¬UNDIRECTED) verhindert, dass die beiden Features UNDIRECTED und DIRECTED gleichzeitig in einer Variante präsent sind. Dies umgeht den Umstand, dass in Abbildung 3.2 zwei Methoden mit gleicher Signatur, aber unterschiedlichem Rückgabetyp, deklariert werden. Eine Änderung des Rückgabetyps von `double` nach `int`, oder das Umbenennen der Methode, zum Beispiel in `getXX()` im Feature UNDIRECTED löst das Problem ebenfalls, allerdings durch Änderungen im Quelltext. Analog behebt der Constraint (DIRECTED ⇔ ¬UNDIRECTED) den Umstand in Abbildung 3.3, das zwei Felder mit gleichem Namen, aber von unterschiedlichem Typ deklariert sind. Durch das Ändern des Typs von Feld `x` in dem Feature UNDIRECTED von `double` in `int` kann der Fehler durch Anpassungen des Quelltextes behoben werden. Eine weitere Maßnahme, die Änderungen am Quelltext nach sich zieht, ist das Umbenennen des Feldes `x`, zum Beispiel in `y`.

Kann in einem Feature nicht auf eine Klasse zugegriffen werden, gibt es ebenfalls mehrere Möglichkeiten dies zu korrigieren. Ist die Klasse in einem anderen Feature-Modul deklariert, wie es in Abbildung 3.6 der Fall ist, kann durch das Hinzufügen des Constraints (GRAPH ⇒ WEIGHTED) sichergestellt werden, dass die Klasse `Edge`

aus dem Feature WEIGHTED immer in der Klasse `Node` des Features GRAPH zur Verfügung steht. Wird die benötigte Klasse nicht gefunden, kann geprüft werden, ob in dem Feature des Zugriffes eine Klasse mit ähnlichem Namen deklariert wurde. Gibt es im Feature GRAPH zum Beispiel eine Klasse `Edde`, kann der Fehler durch eine Umbenennung von `Edge` nach `Edde` behoben werden. Wird die Klasse mit ähnlichem Namen in einem anderen Feature eingeführt, kann es notwendig sein, sicherzustellen, dass dieses Feature erreichbar ist. Führt das Feature WEIGHTED die Klasse `Edde` ein, kann `Edge` in `Edde` umbenannt werden. Es muss weiterhin sichergestellt werden, dass das Feature WEIGHTED immer präsent ist, wenn GRAPH gewählt ist.

Sehr ähnlich sieht es auch bei Compilerfehlern durch fehlende Methoden und Felder aus. Im Beispiel in Abbildung 3.7 wird im Feature SHORTEST die Methode `getWeight()` aufgerufen. Da diese im Feature WEIGHTED deklariert ist, kann ein Constraint (SHORTEST $\Rightarrow$ WEIGHTED) sicherstellen, dass die Methode `getWeight()` immer deklariert ist, wenn SHORTEST gewählt ist. Lässt sich keine Methode mit dem Namen `getWeight()` finden, kann nach Methoden mit ähnlichem Namen gesucht werden. Wird im Feature SHORTEST die Klasse `Edge` mit einer Methode `getHeight()` deklariert, kann der Fehler durch das Umbenennen von `getWeight()` nach `getHeight()` behoben werden. Existiert eine Methode mit ähnlichem Namen in einem anderen Feature, zum Beispiel eine Methode `Edge.getHeight()` im Feature WEIGHTED, kann es, neben der Umbenennung von `getWeight()` nach `getHeight()`, auch notwendig sein, durch ein Constraint (SHORTEST $\Rightarrow$ WEIGHTED) sicherzustellen, dass die Methode `getHeight()` immer verfügbar ist, wenn das Feature SHORTEST gewählt ist. Diese Korrekturen gelten genauso für Felder.

Zur Berechnung der Ähnlichkeit von Zeichenketten gibt es unterschiedliche Algorithmen. Einer ist die Levenshtein-Distanz [Lev65]. Diese berechnet, wie viele Einfüge-, Lösch- und Ersetzoperationen nötig sind, um die eine Zeichenkette in die andere zu überführen. Die Methodennamen `getWeight()` und `getHeight()` haben eine Levenshtein-Distanz von 1, da nur das Ersetzen eines `W` durch ein `H` nötig ist, um `getWeight()` in `getHeight()` zu überführen.

Tabelle 3.1 zeigt eine Zusammenfassung der vorgestellten Korrekturmaßnahmen. Syntaxfehler lassen sich nur durch Änderungen am Quelltext beheben, während sich alle anderen Fehler durch Änderungen am Quelltext oder am Feature-Modell beheben lassen. Bei den vorgestellten Compilerfehlern kann auch eine Kombination aus Quelltext- und Feature-Modell-Änderungen zur Lösung führen.

3.2 Typprüfung in Feature Featherweight Java

Feature Featherweight Java (FFJ) ist eine Feature-orientierte, Java-ähnliche Programmiersprache [AKGL10]. FFJ kann zum Implementieren von Software-Produktlinien genutzt werden. Das Typsystem ist in der Lage sicherzustellen, dass jede Variante der SPL wohlgetypt ist. In diesem Kapitel wird die Syntax von FFJ beschrieben und auf das Typsystem und die damit erkennbaren Fehler eingegangen. Des Weiteren wird diskutiert, welche der vorgestellten Hilfsstrukturen und Anfragen auch für einen Typchecker für FeatureHouse-kompatible Software-Produktlinien genutzt werden können.

Fehler	Fehlertyp	Korrektur durch Änderung von		
		Implementierung	Feature-Modell	Kombination
Syntaxfehler	Composer	+	-	-
Inkompatible Features	Composer	+	+	-
Nicht vorhandener Typ	Compiler	+	+	+
Nicht vorhandene Methode	Compiler	+	+	+
Nicht vorhandenes Feld	Compiler	+	+	+

Tabelle 3.1: Mögliche Korrekturansätze zum Beheben der in Abschnitt 3.1 vorgestellten Fehler. +: Korrektur ist möglich, -: Korrektur ist nicht möglich

3.2.1 Feature Featherweight Java

Feature Featherweight Java (FFJ) ist von Featherweight Java ([IPW01]) inspiriert und wird genutzt, um Typsicherheit für Software-Produktlinien zu beweisen [AKGL10]. Es handelt sich um eine im Funktionsumfang verringerte, funktionale Variante von Java, die durch zusätzliche Schlüsselwörter für Feature-orientierte Programmierung erweitert wurde.

Es können Klassen eingeführt (engl. *introduce*) und verfeinert (engl. *refine*) werden. Für jede eingeführte Klasse muss die Superklasse angegeben werden, auch wenn es sich dabei um `Object` handelt. Die Verfeinerung einer Klasse benötigt das Schlüsselwort `refines`. Dies erlaubt eine explizite Verfeinerung von Klassen und macht es möglich zwischen Einführungen und Verfeinerungen zu unterscheiden. Eine Verfeinerung kann einer Klasse Felder und Methoden hinzufügen oder Methoden verfeinern. Verfeinerten Methoden wird das Schlüsselwort `overrides` vorangestellt. Die explizite Unterscheidung von neu eingeführten und überschriebenen Methoden gibt die Möglichkeit festzustellen, ob eine Methode unbeabsichtigt überschrieben wurde und ob jede überschriebene Methode auch existiert. Es ist nicht möglich bereits eingeführte Felder zu überschreiben. Der Methodenrumpf in FFJ besteht aus `return` und einem einzigen Ausdruck, also nicht aus einer Folge von Anweisungen. Ein Ausdruck wiederum kann in Form einer Variablen, eines Feldzugriffes, eines Methodenaufrufs, einer Objekterzeugung oder eines Castes vorliegen.

In FFJ werden Klassen und Verfeinerungen Features zugeordnet. Eine Menge von Klassen und Verfeinerungen für ein Feature wird Feature-Modul genannt. Die Reihenfolge, in der Verfeinerungen von Klassen durchgeführt werden, wird Verfeinerungskette (engl. *refinement chain*) genannt und beginnt mit der ersten Verfeinerung einer Klasse. Diese Reihenfolge bestimmt sich durch die Komponierreihenfolge der jeweiligen Features.

Im Gegensatz zu FFJ kennt FeatureHouse kein explizites Verfeinern von Klassen und Methoden, da es keine Schlüsselworte gibt, die `refines` oder `overrides` von FFJ entsprechen. Ohne weitere Analysen des Quelltextes ist es nicht möglich festzustellen, ob eine Klasse in einem Feature eingeführt oder verfeinert wurde. Hinweise darauf können Methoden und Felder der Klasse geben, auf die zugegriffen wird, ohne das diese in dem Feature eingeführt wurden. Das Schlüsselwort `original()` setzt ebenfalls voraus, dass die aufrufende Methode und damit auch die Klasse, bereits in einem anderen Feature eingeführt wurde. Aus diesem Grunde ist es nicht möglich, eine eindeutige Verfeinerungskette zu erstellen.

EmailClient
```
1  class Msg extends Object {
2      String serialize () { ... }
3  }
4  class Trans extends Object {
5      Bool send(Msg m) { ... }
6  }
```

SSL
```
7  class SSL extends Object {
8      Trans trans;
9      Bool send(Msg m) { ... }
10 }
11 refines class Trans {
12     Key key;
13     overrides Bool send(Msg m) {
14         return new SSL(this).send(m);
15     }
16 }
```

Abbildung 3.9: Ein Feature-orientierter E-Mail-Client mit Unterstützung für SSL Verschlüsselung in FFJ [AKGL10]

Der Quelltext in Abbildung 3.9 zeigt einen Auszug eines E-Mail-Clients in FFJ. In dem Feature-Modul EmailClient werden die Klassen `Msg` und `Trans` eingeführt, die die Grundfunktionalität eines E-Mail-Clients bereit stellen. Beide Klassen erben von `Object`. Das Feature-Modul SSL führt eine neue Klasse `SSL` ein und verfeinert die Klasse `Trans`, indem das Feld `key` hinzugefügt und die Methode `send` verfeinert wird, um die Nachricht zu verschlüsseln.

3.2.2 FFJ Typsystem

Das Typsystem für FFJ wird zunächst für Feature-orientierte Programmierung, in der jeweils nur eine Variante betrachtet wird, beschrieben und dann für SPLs erweitert [AKGL10]. Für einen Typchecker ist es wichtig herausfinden zu können, welche Klassen, Felder und Methoden für eine bestimmte Feature-Auswahl deklariert sind. Um dies zu ermöglichen, werden zunächst entsprechende Hilfskonstrukte erzeugt und Methoden für diverse Anfragen definiert. Da es sich um ein Typsystem für eine Feature-orientierte Sprache handelt, ist zu prüfen, welche Konzepte für FeatureHouse übernommen werden können.

Klassentabelle

Während des Parsens eines FFJ-Programmes wird eine Klassentabelle CT (engl. *class table*) angelegt [AKGL10]. Es werden alle eingeführten Klassen und Verfeinerungen hinzugefügt. Anfragen an die Klassentabelle geben die Definition von eingeführten oder verfeinerten Klassen in einem Feature zurück. Die Anfrage CT(EmailClient.`Trans`) gibt zum Beispiel die Einführung der Klasse `Trans` im Feature-Modul EmailClient zurück, während CT(SSL.`Trans`) die Verfeinerung der Klasse `Trans` des Feature-Moduls SSL zurück gibt. Eine Klasse darf in dem gleichen Feature-Modul nicht eingeführt und verfeinert oder mehrmals eingeführt oder mehrmals verfeinert werden. Diese Anforderungen machen die Ergebnisse der

Anfragen an die Klassentabelle eindeutig. Es werden noch weitere Plausibilitätsanforderungen an die Klassentabelle gestellt. Jeder Eintrag muss entweder eine Klasse einführen, die von einer anderen Klasse erbt oder eine bereits eingeführte Klasse verfeinern. Kein Eintrag darf als Feature BASE oder als Klasse `Object` haben. Für jede Klasse `C`, die in der Klassentabelle zu finden ist, gibt es mindestens eine Einführung oder Verfeinerung und es gibt keine Zyklen in der Vererbung.

Verfeinerungstabelle

Die Verfeinerungstabelle RT (engl. *refinement table*) gibt für eine gegebene Klasse eine Liste von Feature-Modulen zurück, in denen diese Klasse eingeführt oder verfeinert wurde [AKGL10]. Das erste Element dieser Liste ist dabei das Feature-Modul, in dem die Klasse eingeführt wurde, gefolgt von null oder mehr Feature-Modulen, in denen die Klasse verfeinert wird. Die Reihenfolge ergibt sich dabei aus der Reihenfolge der Komposition, angefangen mit dem Feature-Modul, das die Klasse nach der Einführung als erstes verfeinert hat. Die Anfrage RT(`Trans`) gibt demnach die Liste EMAILCLIENT, SSL zurück.

Anfragen

Neben der Klassentabelle und der Verfeinerungstabelle werden noch zusätzliche Funktionen und Prädikate definiert [AKGL10]. Damit alle Anfragen terminieren wird ein Feature BASE eingeführt, das nur die Klasse `Object` einführt.

Die Funktion `fields`$(\Phi.C)$ gibt die Felder zurück, die von einer Klasse bereit gestellt werden. Dazu wird zunächst die Verfeinerungskette der zugehörigen Klasse von der letzten Verfeinerung bis zu der Einführung der Klasse durchgegangen und alle Felder, die eingeführt werden, in einer Liste gesammelt. Ist das erste Element der Verfeinerungskette erreicht (das Feature-Modul, in dem die Klasse eingeführt wird), wird die Suche bei dem letzten Element der Verfeinerungskette der Superklasse fortgeführt. Die Suche terminiert, wenn BASE.`Object` erreicht wird.

Die Funktion `mbody`$(m, \Phi.C)$ gibt den Methodenkörper zurück, der in der Verfeinerungskette als letztes eingeführt oder verfeinert wurde. Dazu werden wie bei `fields` die Verfeinerungskette und die Vererbungshierarchie durchgegangen. Bei der ersten gefundenen Einführung oder Verfeinerung der gesuchten Methode wird der Methodenkörper zurückgegeben. Die Funktion `mtype`$(m, \Phi.C)$ sucht wie `mbody` in der Verfeinerungskette und Vererbungshierarchie, ignoriert allerdings Methodenverfeinerungen und gibt den Typ einer Methode bei der ersten passenden Methodeneinführung zurück.

Weiterhin werden die Prädikate $\texttt{introduce}_{class}(\Phi.C)$, $\texttt{introduce}_{method}(f, \Phi.C)$ und $\texttt{introduce}_{field}(m, \Phi.C)$ von Apel et al. eingeführt. Diese stellen fest, ob die Klassen-, Methoden- und Feldeinführungen gültig sind, das heißt, ob sie nicht bereits in anderen Features eingeführt wurden. Das Prädikat `refine`$(\Phi.C)$ stellt fest, ob die Verfeinerung einer Klasse auf einer bereits eingeführten Klasse durchgeführt wird. Das Prädikat `override`$(m, \Phi.C, \overline{C} \to C_0)$ stellt fest, ob eine zu verfeinernde Methode bereits eingeführt wurde und die Signaturen übereinstimmen [AKGL10].

```
1  refines class Trans {                                                    TEXT
2      Unit receive(Msg msg) {
3          return /* ... */ new Display().render(msg);
4      }
5  }
6  class Display {
7      Unit render(Msg msg) {/* zeige Nachricht im text Format an */}
8  }
```

```
9  refines class Display {                                                  MOZILLA
10     MozillaRenderer renderer;
11     overrides Unit render(Msg msg) {/* HTML Nachricht mit Mozilla */}
12 }
```

```
13 refines class Display {                                                  SAFARI
14     SafariRenderer renderer;
15     overrides Unit render(Msg msg) {/* HTML Nachricht mit Safari */}
16 }
```

Abbildung 3.10: Beispiel für FFJ_{PL} [AKGL10]

Typsicherheit

Ein FFJ-Programm besteht aus einer Klassentabelle, einer Verfeinerungstabelle und einem Term. Um die Typsicherheit eines FFJ-Programms sicherzustellen, muss der Term wohlgetypt sein, alle Klassen und Verfeinerungen müssen wohlgeformt sein und die Klassentabelle des Programms die Plausibilitätsbedingungen erfüllen [AKGL10]. Die Regeln, nach denen wohlgetypte Terme und die Wohlgeformtheit von Klassen und Verfeinerungen sichergestellt werden, können [AKGL10] entnommen werden.

3.2.3 FFJ_{PL} Typsystem

Nachdem das Typsystem für FFJ beschrieben wurde, wird nun auf das Typsystem der FFJ Erweiterung für Software-Produktlinien, FFJ_{PL} [AKGL10], eingegangen. Eine Software-Produktlinie auf Wohlgetyptheit zu testen, bedeutet sicherzustellen, dass alle Produkte wohlgetypt sind. Dazu ist es notwendig, Informationen über gültige Feature-Kombinationen in die Typprüfung einzubeziehen. Wichtig ist zu beachten, dass es Features geben kann, die optional sind oder sich gegenseitig ausschließen.

Erweiterte Hilfsstrukturen

Um das Typsystem für Software-Produktlinien zu erweitern, müssen die Klassentabelle und die Verfeinerungstabelle angepasst und eine Einführungstabelle IT (engl. *introduction table*) erzeugt werden [AKGL10]. Grundsätzlich muss die Klassentabelle bei FFJ_{PL} die gleichen Bedingungen erfüllen wie bei FFJ. Eine Ausnahme betreffen Zyklen in der Vererbungshierarchie. Während FFJ keine Zyklen in der Vererbungshierarchie in der Klassentabelle erlaubt, sind diese in der Klassentabelle von FFJ_{PL} erlaubt, solange keine Varianten mit Zyklen in der Vererbungshierarchie möglich sind. Die Einführungstabelle gibt für eine Klasse die Features zurück, in denen diese Klasse eingeführt wird. Die Features müssen sich gegenseitig ausschließen,

da eine Klasse in einer Variante nur einmal eingeführt werden kann. Eine Anfrage IT(`Display`) an die erweiterte E-Mail-Client SPL in Abbildung 3.10 würde das Feature-Modul TEXT zurückgeben. Auch Anfragen nach Feldern oder Methoden sind möglich. Die Anfrage IT(`Display.renderer`) wird MOZILLA, SAFARI zurückgeben. Die Verfeinerungstabelle wird so angepasst, dass bei einer Anfrage neben den Features, die eine Klasse verfeinern, auch die Features zurückgegeben werden, die eine Klasse einführen. Die Reihenfolge der Features, die durch die Einführungstabelle und die Verfeinerungstabelle zurückgegeben werden, richtet sich nach der Reihenfolge der Features im Feature-Modell [AKGL10].

Das Konzept der Klassentabelle kann genauso für FeatureHouse übernommen werden, da in der Schnittstelle zum Typsystem nicht zwischen Einführung und Verfeinerung einer Klasse unterschieden wird. Auch eine Verfeinerungstabelle, wie für FFJ_{PL} beschrieben, kann übernommen werden, da im Gegensatz zu der Verfeinerungstabelle von FFJ ebenfalls nicht zwischen Einführung und Verfeinerung unterschieden wird. Die Anfragen nach der Einführung von Methoden und Feldern der Einführungstabelle können ebenfalls übernommen werden. Da nicht zwischen Einführungen und Verfeinerungen unterschieden wird, kann eine Anfrage nach Features, die eine Klasse einführen, nicht bearbeitet werden.

Anfragen an das Feature-Modell

Anfragen an das Feature-Modell sind notwendig, um festzustellen, ob Features immer, manchmal oder gar nicht in einer Kombination mit anderen Features auftreten [AKGL10]. Dafür definieren Apel et al. zum einen das Prädikat $\texttt{sometimes}(\overline{\Omega}, \Phi)$, das immer dann wahr ist, wenn es eine gültige Feature-Kombination gibt, in dem das Feature Φ mit den Features $\overline{\Omega}$ vorkommt. Die Menge von Features $\overline{\Omega}$ kann auch als Kontext bezeichnet werden, für den geprüft wird, ob Φ immer präsent ist. Das Prädikat `sometimes` kann durch eine Negation auch als das Prädikat $\texttt{never}(\overline{\Omega}, \Phi)$ verwendet werden, das immer dann wahr ist, wenn das Feature Φ nie im Feature-Kontext $\overline{\Omega}$ vorkommt ($\texttt{never}(\overline{\Omega}, \Phi) = \neg\ \texttt{sometimes}(\overline{\Omega}, \Phi)$).

Die Funktion $\texttt{always}(\overline{\Omega}, \Phi)$ wird genutzt um zu prüfen, ob ein Feature Φ immer im Kontext der Features $\overline{\Omega}$ vorhanden ist, entweder alleine, oder als Teil einer alternativen Gruppe von Features [AKGL10]. Aus einer alternativen Gruppe von Features kann nur ein Feature gewählt sein. Die Wahl eines Features der Gruppe schließt alle anderen Features der Gruppe aus. Für die Funktion `always` sind drei Fälle zu unterscheiden.

1. Ist Φ immer im Kontext vorhanden wird Φ zurückgegeben.
2. Ist Φ nicht immer im Kontext vorhanden, gehört aber zu der kleinsten Gruppe sich gegenseitig ausschließender Features, von denen immer eins im Kontext gewählt ist, gibt `always` diese Gruppe inklusive Φ zurück.
3. Trifft keiner der ersten beiden Fälle zu wird eine leere Liste zurückgegeben.

Um festzustellen, ob ein Programm-Element in einem bestimmten Feature-Kontext $\overline{\Omega}$ immer erreichbar ist, werden weitere Anfragen benötigt. Hierfür werden die Prädikate $\texttt{validref}_{class}(\overline{\Omega}, C)$, $\texttt{validref}_{method}(\overline{\Omega}, C.m)$ und $\texttt{validref}_{field}(\overline{\Omega}, C.f)$ für Klassen, Methoden und Felder eingeführt.

Felder und Methoden

Die Funktion `fields`$(\overline{\Omega}, \Phi.C)$ gibt eine Liste von Listen der in der Klasse $\Phi.C$ im Feature-Kontext $\overline{\Omega}$ eingeführten Felder zurück. Eine Liste von Listen ist nötig, da alternative Feature-Gruppen zu unterschiedlichen Listen von eingeführten Feldern führen können. Wie für die `fields` Funktion in FFJ wird die Verfeinerungs- und Vererbungshierarchie durchgegangen. Felder, die in Features eingeführt werden, die immer in dem gegebenen Kontext vorhanden sind, werden der Liste hinzugefügt, Felder aus Features, die nie vorhanden sind, werden ignoriert. Felder aus Features, die vorkommen können, werden für das Typsystem mit einem @ gekennzeichnet, um auf den Umstand hinzuweisen, dass ein Feld nicht in jeder Feature-Kombination vorhanden sein muss. Wird eine alternative Gruppe gefunden, dupliziert `fields` die Ergebnisliste, fügt die Felder, die jedes einzelne der alternativen Features einführt, hinzu und die Suche wird mit allen Listen fortgeführt [AKGL10].

Die Funktion `mtype`$(\overline{\Omega}, m, \Phi.C)$ gibt eine Liste aller Signaturen der Methode `m` der Klasse $\Phi.C$ zurück, die im Feature-Kontext $\overline{\Omega}$ eingeführt werden. Durch alternative Features kann es dazu kommen, dass es mehr als eine Signatur gibt. Dies macht es notwendig eine Liste zurückzugeben [AKGL10].

Das Prädikat $\texttt{introduce}_{class}(\overline{\Omega}, \Phi.C)$ zeigt, dass die Klasse $\Phi.C$ in dem gegebenen Kontext noch in keinem Feature eingeführt wurde. Die Prädikate $\texttt{introduce}_{field}(\overline{\Omega}, f, \Phi.C)$ und $\texttt{introduce}_{method}(\overline{\Omega}, m, \Phi.C)$ stellen sicher, dass ein Feld oder eine Methode für $\Phi.C$ im gegebenen Feature-Kontext noch nicht eingeführt wurde [AKGL10].

Das Prädikat `refine`$(\overline{\Omega}, \Phi.C)$ stellt fest, ob die Klasse $\Phi.C$ im Kontext der Features $\overline{\Omega}$ bereits eingeführt wurde und immer erreichbar ist, also verfeinert werden kann. Das Prädikat `override`$(\overline{\Omega}, m, \Phi.C, \overline{C} \to C_0)$ prüft, ob eine Methode mit gegebener Signatur in gegebenem Kontext eingeführt und immer erreichbar ist [AKGL10].

Typsicherheit

Ein FFJ_{PL}-Programm besteht aus einer Klassentabelle, einer Verfeinerungstabelle, einer Einführungstabelle und einem Term [AKGL10]. Eine Software-Produktlinie in FFJ_{PL} ist wohlgetypt, wenn der Term wohlgetypt ist, alle Klassen und Verfeinerungen wohlgeformt sind und die Klassentabelle die Plausibilitätsanforderungen erfüllt. Die Regeln, nach denen wohlgetypte Terme und die Wohlgeformtheit von Klassen und Verfeinerungen sichergestellt werden, können [AKGL10] entnommen werden.

Die Typsysteme von FFJ und FFJ_{PL} stellen sicher, dass alle Klassen, Methoden und Felder auf die zugegriffen wird in jeder Variante vorhanden sind. Es werden also alle Fehler die in Abschnitt 3.1.2 vorgestellt wurden und in FFJ möglich wären adressiert.

3.3 Typchecker für Feature-orientierte SPLs

Im letzten Abschnitt wurde ein bestehendes Typsystem für FFJ_{PL} beschrieben, das es ermöglicht sicherzustellen, das eine Software-Produktlinie typsicher ist. Feature Featherweight Java ist in der Funktionalität allerdings sehr eingeschränkt. Die

gesamte Funktionalität von Java in einen Typchecker für Software-Produktlinien umzusetzen ist sehr aufwendig, da die Spezifikation für JavaSE7[1] mehr als 600 Seiten umfasst.

In diesem Abschnitt wird ein Typchecker vorgeschlagen, der während der Typprüfung einzelne Typchecks auf einer Software-Produktlinie durchführen kann. Die Typchecks haben Zugriff auf die Feature-Module und das Feature-Modell der SPL. Der Typchecker arbeitet in drei Phasen, die jeweils auf die Ergebnisse der letzten Phase zurückgreifen. Abbildung 3.11 zeigt den Ablauf des Typcheckers. In der *Vorverarbeitung des Quelltextes* wird der Quelltext aller vorhandener Feature-Module geparst und in einen abstrakten Syntaxbaum (engl. *abstract syntax tree*) überführt. Aus diesem abstrakten Syntaxbaum wird eine Klassentabelle erzeugt, die einen vereinfachten Zugriff auf Informationen zu den Feature-Modulen ermöglicht. Mit Hilfe dieser Klassentabelle und dem Feature-Modell werden in der Phase *Typprüfung und Problemsammlung* die Typchecks durchgeführt. Die Ergebnisse der Typprüfung werden in der Phase *Berichten und Korrigieren* genutzt um dem Anwender Vorschläge zum Beheben möglicher Probleme zu geben. Dies umfasst im Wesentlichen Anpassungen des Feature-Modells und Änderungen des Quelltextes von Feature-Modulen. Änderungen am Quelltext oder dem Feature-Modell machen eine erneute Prüfung notwendig, die wieder mit der Vorverarbeitung startet.

In den nächsten Abschnitten wird anhand des Quelltextes in Abbildung 3.12 genauer auf die einzelnen Phasen eingegangen.

3.3.1 Vorverarbeitung des Quelltextes

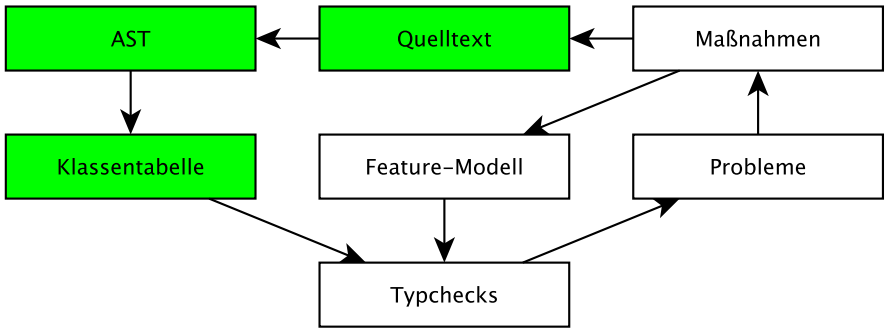
In der ersten Phase des Typcheckers werden die vorhandenen Informationen zu der Software-Produktlinie in einer Weise aufbereitet, mit der es im weiteren Verlauf einfacher ist auf diese zuzugreifen. Dazu wird zunächst aus dem Quelltext der Feature-Module der SPL ein abstrakter Syntaxbaum erzeugt und die so gewonnenen Informationen in die Klassentabelle überführt.

Abstrakter Syntaxbaum

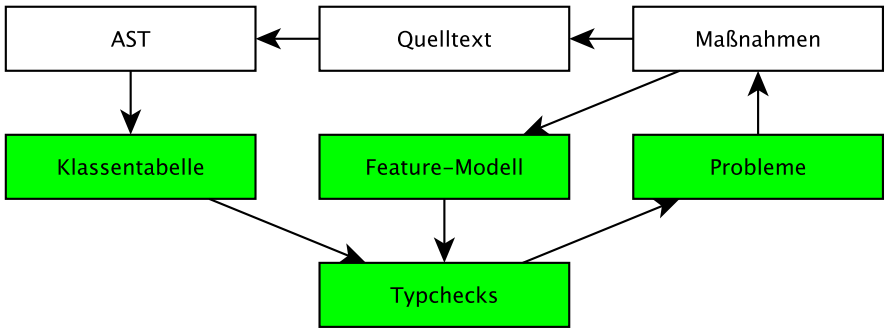
Ein abstrakter Syntaxbaum ist eine abstrakte Repräsentation von Quelltext. Jedes Programm-Element hat ein Baum-Element, das ihn repräsentiert. Semantisch wichtige Elemente eines Knotens werden als Kindknoten organisiert [ALSU08]. Ein Java-Programm hat als Kinder einen Knoten für jede Quelltext-Datei. In einer Quelltext-Datei können unter anderem Klassen und Interfaces deklariert werden, die dann Kindknoten einer solchen Datei sind. Eine Klassendeklaration wiederum kann zum Beispiel Feld- und Methodendeklarationen, sowie Konstruktoren als Kindknoten haben.

Ein abstrakter Syntaxbaum kann unterschiedliche Detailgrade haben. Während der eine Parser bei der Erstellung eines abstrakten Syntaxbaums nur bis zu Methodendeklarationen parst, geht ein anderer auch in die Methodenkörper und parst jeden Ausdruck. Um auf alle Informationen, die im Quelltext vorhanden sind, zurückgreifen zu können, ist es für den Typchchecker nötig, einen möglichst hohen Detailgrad zu haben und einen Parser zu nutzen, der alles erfassen kann. Zusätzlich können durch das Parsen vorhandene Syntaxfehler erkannt werden.

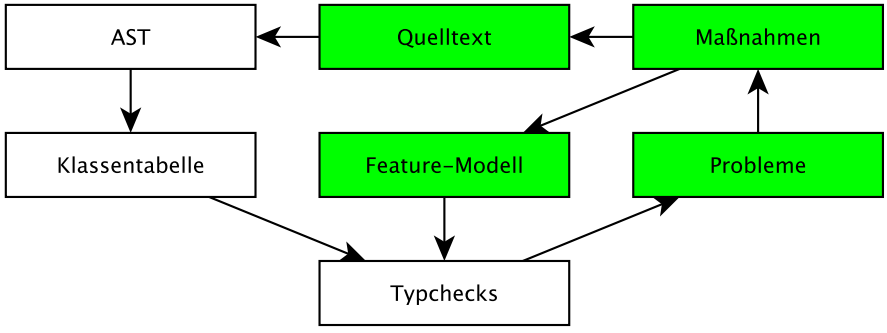
[1] http://docs.oracle.com/javase/specs/



(a) Phase *Vorverarbeitung des Quelltextes*: aus dem Quelltext wird der abstrakte Syntaxbaum erzeugt, mit dessen Hilfe die Klassentabelle erstellt wird.



(b) Phase *Typprüfung und Problemsammlung*: mit Hilfe der Klassentabelle und dem Feature-Modell werden Typchecks durchgeführt und erkannte Probleme gesammelt.



(c) Phase *Berichten und Korrigieren*: Probleme werden dem Nutzer gemeldet und es werden Korrekturmaßnahmen vorgeschlagen, die den Quelltext und das Feature-Modell ändern können.

Abbildung 3.11: Ablaufplan des Typcheckers und der in den einzelnen Phasen genutzten Komponenten (grün hervorgehoben)

```
1  public class Node {                                              Graph
2      String name;
3      Edge[] edges;
4      public Edge[] getEdges() { return edges; }
5  }
```

```
6  public class Edge {                                              Directed
7      Node start, end;
8      public Node getStartNode(){ return start; }
9  }
```

```
10 public class Edge {                                              Weighted
11     int weight;
12     public in getWeight() { return weight; }
13 }
```

Abbildung 3.12: Quelltext der Features Graph, Directed und Weighted der GPL zum Veranschaulichen des Konzeptes

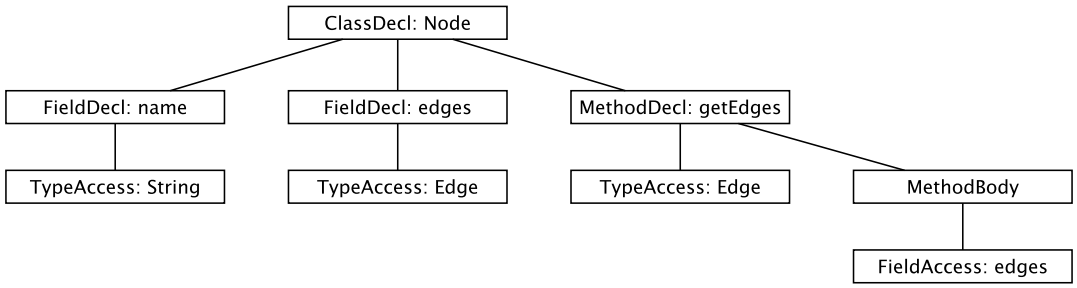


Abbildung 3.13: Abstrakter Syntaxbaum des Features-Moduls Graph der GPL

Abbildung 3.13 zeigt einen abstrakten Syntaxbaum für die Klasse `Node` des Features Graph für den in Abbildung 3.12 gezeigten Quelltext. Es wird die Klasse `Node` deklariert, die zwei Felder `name` und `edges` und die Methode `getEdges()` enthält. Die Felddeklarationen haben einen Kindknoten vom Knoten-Typ `TypeAccess`, das den Typ des Feldes festhält. Der Rückgabetyp der Methode `getEdges()` wird durch den `TypeAccess` wiedergegeben und der Methoden-Körper enthält den Feldzugriff auf das Feld `edges`. Mit Hilfe des abstrakten Syntaxbaums ist es möglich, sich leichter durch die Programm-Elemente eines Programms zu bewegen, als es zum Beispiel auf dem Quelltext selber möglich wäre.

Klassentabelle

Der abstrakte Syntaxbaum stellt alle Daten zur Verfügung, die in dem Quelltext der Feature-Module vorhanden sind. Um einen effizienten Zugriff auf diese Daten zu gewährleisten, wird eine Klassentabelle angelegt. Diese bietet Zugriff auf Klassen, Methoden und Felder. Eine mögliche Anfrage ist CT(`C`), die alle Features zurück gibt, in denen die Klasse `C` eingeführt oder erweitert wird. Dies entspricht einer Anfrage RT(`C`) an die Verfeinerungstabelle RT von FFJ_{PL}. Da FeatureHouse kompatibles Java keine explizite Verfeinerungen wie das `refines` Schlüsselwort in FFJ kennt, kann ohne weitere Analysen nicht sicher gesagt werden, ob eine Klassendefinition eine Einführung oder eine Verfeinerung ist (siehe Abschnitt 3.2.1). Eine weitere

Anfrage ist CT(Φ), die alle Klassendefinitionen zurück gibt, die in einem Feature Φ eingeführt werden. CT(`C.m`) liefert alle Features, die eine Methode `m` der Klasse `C` einführen, CT(`C.f`) alle Features, die das Feld `f` von `C` einführen. Diese Funktionalität entspricht den Anfragen IT(`C.m`) und IT(`C.f`) der Einführungstabelle IT von FFJ_{PL}.

Auf das Beispiel in Abbildung 3.12 bezogen liefert eine Anfrage CT(`Edge`) die Features Weighted und Directed, da in beiden eine Rolle für die Klasse `Edge` definiert ist. Das Ergebnis von CT(`Node.name`) ist Graph und das Ergebnis von CT(`Edge.getStartNode()`) ist Directed.

Zusammenfassung

In der Phase der *Vorverarbeitung des Quelltextes* wird der Quelltext der Software-Produktlinie in Datenstrukturen überführt, die es ermöglichen sollen, eine möglichst effektive Typprüfung durchführen zu können. Zum einen wird eine Klassentabelle erzeugt, die Informationen zu Klassen und deren Methoden und Feldern bereitstellt, sowie in welchen Features diese eingeführt oder erweitert werden. Zum anderen kann zusätzlich auch auf den erzeugten abstrakten Syntaxbaum zugegriffen werden, sollten die Informationen in der Klassentabelle nicht ausreichend sein.

Im nächsten Abschnitt wird die Phase der *Typprüfung und Problemsammlung* beschrieben, in der mit Hilfe der in der Vorverarbeitung erstellen Datenstrukturen die Typprüfung durchgeführt wird.

3.3.2 Typprüfung und Problemsammlung

In der zweiten Phase wird die Typprüfung durchgeführt. Die einzelnen Typchecks haben dabei Zugriff auf die im vorhergehenden Schritt erzeugte Klassentabelle und das Feature-Modell der Software-Produktlinie. Die in dieser Phase festgestellten Probleme werden gesammelt und in der nächsten Phase ausgewertet.

Feature-Modell-Anfragen

Um zu prüfen, ob Features immer in der Präsenz anderer Features gewählt sind, ist es notwendig, ein Prädikat $implies(\Phi, \overline{\Omega})$ zu definieren. Dies gibt *wahr* zurück, wenn bei der Wahl von Feature Φ immer eines der Features in $\overline{\Omega}$ präsent ist. Um diese Anfrage zu beantworten, kann ein SAT-Solver verwendet werden (siehe Abschnitt 2.1).

Für $implies(\Phi, \overline{\Omega})$ ist festzustellen, ob, wenn das Feature-Modell gilt, auch die zusätzliche Anforderung $(\Phi \Rightarrow (\Omega_1 \vee ... \vee \Omega_n))$ gilt, das heißt der gesamte Ausdruck eine Tautologie ist. Das Feature-Modell kann als aussagenlogischer Ausdruck FM dargestellt werden, wie in Abschnitt 2.1 gezeigt. Die Aussage, das, wenn FM *wahr* ist, auch $(\Phi \Rightarrow (\Omega_1 \vee ...\Omega_n))$ *wahr* sein muss, lässt sich mit einer Implikation ausdrücken und führt zu dem Ausdruck $(FM \Rightarrow (\Phi \Rightarrow (\Omega_1 \vee ... \vee \Omega_n)))$. Dieser Ausdruck muss darauf geprüft werden, ob es sich um eine Tautologie handelt, was wiederum zum Ausdruck

$$\neg SAT(\neg(FM \Rightarrow (\Phi \Rightarrow (\Omega_1 \vee ... \vee \Omega_n))))$$

führt (siehe Abschnitt 2.1), der dem gewünschten Prädikat $implies(\Phi, \overline{\Omega})$ entspricht.

Beispiel Typzugriffe

Als Beispiel soll ein Typcheck dienen, der prüft, ob alle genutzten Klassen in allen Varianten der GPL aus Abbildung 3.12 immer vorhanden sind. Das Feature GRAPH nutzt in der Klasse `Node` die Klasse `String`. Diese gehört zur Systembibliothek und ist aus diesem Grund immer vorhanden. Des Weiteren wird auf die Klasse `Edge` zugegriffen. Das Feature GRAPH hat für diese Klasse keine Rolle. Die Anfrage CT(`Edge`) an die Klassentabelle liefert als Ergebnis, dass die Features DIRECTED und WEIGHTED eine Klasse `Edge` einführen oder verfeinern. Es wird mit

$$implies(\text{GRAPH}, \{\text{WEIGHTED}, \text{DIRECTED}\})$$

eine Anfrage an das Feature-Modell gestellt, die feststellt, ob immer eines der Features DIRECTED oder WEIGHTED gewählt ist, wenn GRAPH gewählt ist. Ist es möglich, GRAPH ohne DIRECTED oder WEIGHTED zu wählen, kommt es in bestimmten Varianten mit GRAPH, aber ohne DIRECTED oder WEIGHTED zu einem Typfehler. Das Prädikat $implies(\text{GRAPH}, \{\text{WEIGHTED}, \text{DIRECTED}\})$ prüft, ob es sich bei dem Ausdruck $(FM \Rightarrow (\text{GRAPH} \Rightarrow (\text{WEIGHTED} \vee \text{DIRECTED})))$ um eine Tautologie handelt.

$$implies(\text{GRAPH}, \{\text{WEIGHTED}, \text{DIRECTED}\}) \Leftrightarrow$$
$$\neg SAT(\neg(FM \Rightarrow (\text{GRAPH} \Rightarrow (\text{WEIGHTED} \vee \text{DIRECTED}))))$$

Ist das Ergebnis des SAT-Solvers *wahr*, ist das Ergebnis der `implies` Anfrage *falsch* und es kommt in bestimmten Varianten zu einer fehlenden Klasse `Edge` im Feature GRAPH. Ist das Ergebnis der Erfüllbarkeitsanfrage *falsch*, so ist das Ergebnis von `implies` wahr und es ist somit sichergestellt, dass für jede Variante mit dem Feature GRAPH eine Klasse `Edge` erreichbar ist.

Probleme

Alle erkannten Fehler werden als Problem, das es zu lösen gilt, an die *Berichten und Korrigieren* Phase übergeben. Ein Problem enthält die Informationen, die benötigt werden, um Korrekturvorschläge zu machen und dem Anwender das Problem zu melden. Folgende Fragen sollen durch diese Informationen beantwortet werden können:

- Um was für einen Fehler handelt es sich? (zum Beispiel fehlende Klasse oder Syntaxfehler)
- Wenn etwas fehlt, wie ist der Name? (der Klasse, der Methode, des Feldes)
- Wo im Quelltext ist der Fehler bemerkt worden? (Feature, Datei, Position in der Datei)
- Welche Features könnten ein nicht erreichbares Element bereitstellen?

Diese Fragen lassen sich für den Fehler des Typzugriff-Beispiels beantworten:

- Es handelt sich um ein Problem der Art *fehlende Klasse*,
- Es fehlt eine Klasse mit dem Namen `Edge`.
- Der Fehler tritt in Feature GRAPH in der Datei *Node.java* in Zeile `3` auf.
- Features, die eine Klasse mit dem Namen `Edge` bereitstellen sind DIRECTED und WEIGHTED.

Alle Probleme, die sich während der Typprüfung feststellen lassen, werden gesammelt. In der nächsten Phase wird versucht, Korrekturmaßnahmen auf Basis dieser Probleme zu erarbeiten.

Zusammenfassung

In der Phase *Typprüfung und Problemsammlung* wird die eigentliche Typprüfung ausgeführt. Dazu werden einzelne Typchecks auf der Software-Produktlinie durchgeführt. Um festzustellen, ob benötigte Abhängigkeiten zwischen einzelnen Features im Feature-Modell bereits vorhanden sind, wurde das Prädikat $implies(\Phi, \overline{\Omega})$ eingeführt. Das Prädikat *implies* ist wahr, wenn bei der Wahl von Feature Φ mindestens eines der Features in $\overline{\Omega}$ ebenfalls gewählt ist. Um in der Lage zu sein, dem Anwender hilfreiche Hinweise auf einen Fehler und zur Korrektur dieses Fehlers zu geben, wird ein Problem als eine Menge an Informationen zu einem Fehler definiert. Alle gefundenen Fehler werden durch ein Problem repräsentiert und an die nächste Phase *Berichten und Korrigieren* übergeben.

3.3.3 Berichten und Korrigieren

Die letzte Phase des Typcheckers ist das *Berichten und Korrigieren*. Alle in der Typprüfung gefundenen Probleme werden analysiert, mögliche Korrekturvorschläge ermittelt und dem Anwender mitgeteilt.

Der erste Schritt in dieser Phase ist das Generieren einer dem Fehler angepassten Problemmeldung, die es dem Anwender ermöglicht, das Problem zu verstehen. Für jeden der in Abschnitt 3.1 beschriebenen Fehler wird im Folgenden eine Problemmeldung vorgeschlagen und mögliche konkrete Korrekturmaßnahmen (vergleiche Abschnitt 3.1.3) beschrieben.

Korrekturen für Syntaxfehler

Die Problemmeldung des Syntaxfehlers aus Abbildung 3.1 kann wie folgt aussehen:

```
Vor dem Token "public" wurde eine fehlerhafte Syntax festgestellt.
```

Liegt ein Syntaxfehler vor, ist die Korrektur auf das Ändern des Quelltextes beschränkt. Mit einem entsprechend fortgeschrittenen Parser ist es möglich festzustellen, welche Token erwartet worden wären. So gibt FEATUREHOUSE unter anderem `implements` und `{` als mögliche Lösungen bei der fehlenden Klammer des Syntaxfehler-Beispiels.

Korrekturen für inkompatible Features

Eine Problemmeldung für das Beispiel aus Abbildung 3.3 kann wie folgt formuliert werden:

```
Das Feld "x" der Klasse "Edge" hat unterschiedliche Typen in den
Features "Directed" (Typ: int) und "Undirected" (Typ: double).
```

Fehler durch inkompatible Features lassen sich nach Abschnitt 3.1.3 durch das Anpassen des Feature-Modells und das Verändern des Quelltextes verhindern. Für den konkreten Fehler ergeben sich daraus folgende Korrekturmaßnahmen:

1. Hinzufügen des Constraints (DIRECTED $\Leftrightarrow \neg$UNDIRECTED).
2. Ändern des Typs von Feld `x` in Klasse `Edge` im Feature DIRECTED von `int` nach `double`.
3. Ändern des Typs von Feld `x` in Klasse `Edge` im Feature UNDIRECTED von `double` nach `int`.
4. Umbenennen des Feldes `x` in Klasse `Edge` im Feature DIRECTED.
5. Umbenennen des Feldes `x` in Klasse `Edge` im Feature UNDIRECTED.

Korrekturen für fehlende Typen

Eine Problemmeldung für das in Abbildung 3.6 gezeigte Beispiel sieht wie folgt aus:

```
Auf die Klasse "Edge" kann aus dem Feature "Graph" nicht zugegriffen
werden. Eine Klasse "Edge" wird im Feature "Weighted" deklariert.
```

Ein Fehler durch nicht vorhandenen Typen kann mehrere Lösungen haben. Zum einen kann die gesuchte Klasse in einem anderen Feature deklariert sein, zum anderen kann eine Klasse mit ähnlichem Namen existieren. Eine dritte Möglichkeit ist, dass der Name der Klasse gar nicht aufgelöst werden kann. An dem erweiterten Beispiel der nicht vorhandenen Typen aus Abschnitt 3.1.3, ergeben sich je nach Situation unterschiedliche Korrekturmöglichkeiten.

1. Die Klasse `Edge` ist im Feature WEIGHTED deklariert:
   - Hinzufügen des Constraints (GRAPH $\Rightarrow$ WEIGHTED).
2. Eine Klasse `Edde` ist im Feature GRAPH oder WEIGHTED deklariert:
   - Ändern des Typs von `edges` im Feature GRAPH von `Edge` in `Edde`.
3. Es existiert in keinem Feature eine Klasse, die `Edge` ähnlich genug ist:
   - Erstellen einer neuen Klasse `Edge` im Feature GRAPH.

Treffen die beiden Situationen 1 und 2 gleichzeitig zu, können die Korrekturvorschläge kombiniert werden. Wird der Typ von `edges` in `Edde` aus dem Feature WEIGHTED umbenannt, kann mit `implies(GRAPH, WEIGHTED)` geprüft werden, ob das Feature WEIGHTED immer gewählt ist, wenn Feature GRAPH gewählt ist. Diese Abhängigkeit wird allerdings auch beim nächsten Typchecker-Durchgang erkannt und behoben, eine Prüfung zu diesem Zeitpunkt kann gegebenenfalls ausgelassen werden.

Korrekturen für fehlende Methoden und Felder

Eine Problemmeldung für das Beispiel nicht vorhandener Methoden aus Abbildung 3.7 sieht wie folgt aus:

```
Die Methode "getWeight()" der Klasse "Edge" ist im Feature "Shortest"
nicht erreichbar. Sie wird in Klasse "Edge" im Feature "Weighted"
deklariert.
```

Unter der Voraussetzung, dass die Klasse der Methode oder des Feldes vorhanden ist, ergeben sich, wie bei den Typreferenzen, drei unterschiedliche Situationen. Die gesuchte Methode oder das gesuchte Feld können in einem anderen Feature deklariert sein, eine Methode oder ein Feld mit ähnlichem Namen ist in einem Feature deklariert, oder es existiert keine Methode oder kein Feld mit einem Namen, der ähnlich genug ist (siehe Abschnitt 3.1.3).

Auch bei dem erweiterten Beispiel der nicht vorhandenen Methoden aus Abschnitt 3.1.3 ergeben sich, je nach Situation, unterschiedliche Korrekturmöglichkeiten.

1. Die Methode `getWeight()` der Klasse `Edge` ist im Feature WEIGHTED deklariert:
   - Hinzufügen des Constraints (SHORTEST ⇒ WEIGHTED).
2. Eine Methode `getHeight()` der Klasse `Edge` ist im Feature SHORTEST oder WEIGHTED deklariert:
   - Ändern des Namens des Methodenaufrufs `getWeight()` in Klasse `Edge` im Feature SHORTEST in `getHeight()`.
3. Es existiert in keinem Feature eine Methode, die `getWeight()` der Klasse `Edge` ähnlich genug ist:
   - Erstellen einer neuen Methode `getWeight()` in Klasse `Edge` im Feature GRAPH.

Treffen die beiden Situationen 1 und 2 gleichzeitig zu, werden auch beide Korrekturmöglichkeiten vorgeschlagen. Wird der Name des Methodenaufrufs `getWeight()` der Klasse `Edge` auf `getHeight()` der Klasse `Edge` des Features WEIGHTED geändert, muss sichergestellt sein, dass das Feature WEIGHTED immer präsent ist, wenn das Feature SHORTEST gewählt ist. Dies kann ebenfalls durch einen Aufruf von `implies(SHORTEST, WEIGHTED)` geprüft oder dem Typchecker im nächsten Durchlauf überlassen werden.

Eine Problemmeldung für das Beispiel nicht vorhandener Felder aus Abbildung 3.8 sieht wie folgt aus:

```
Das Feld "weight" der Klasse "Edge" ist im Feature "Shortest" nicht
erreichbar. Es wird in Klasse "Edge" im Feature "Weighted" deklariert.
```

Die Korrekturmöglichkeiten bei der nicht vorhandenen Felder ergeben sich analog zu den vorher beschriebenen für nicht vorhandene Methoden.

Bewerten der Korrekturmöglichkeiten

Jedes Problem hat eine eigene Liste mit Korrekturmaßnahmen. Neben diesen lokalen Listen gibt es eine globale Liste, in der die Korrekturmaßnahmen aller Probleme des aktuellen Durchgangs gesammelt sind. Um in größeren Projekten mit vielen Fehlern die Übersicht zu bewahren und Korrekturmaßnahmen hervorzuheben, die die Probleme wahrscheinlich am besten lösen, werden diese sortiert. Möglichkeiten der Sortierung sind die Anzahl von Problemen, die eine Korrekturmaßnahme behebt oder die Art der Maßnahme.

Der Vorgang des Bewertens der Korrekturmaßnahmen wird an einem Beispiel verdeutlicht. Bei einem Durchgang des Typcheckers sind drei Probleme aufgetreten, für die folgende Korrekturmaßnahmen vorgeschlagen werden:

- Auf die Klasse "Edge" kann aus dem Feature "Shortest" nicht zugegriffen werden. Eine Klasse "Edge" wird im Feature "Graph" deklariert
  - Hinzufügen des Constraint (Shortest ⇒ Graph).
  - Ändern des Typs von Edge nach Edde in Klasse `Shortest` im Feature Shortest.
- Auf die Klasse "Node" kann aus dem Feature "Shortest" nicht zugegriffen werden. Eine Klasse "Node" wird im Feature "Graph" deklariert.
  - Hinzufügen des Constraint (Shortest ⇒ Graph).
  - Erzeugen neuer Klasse `Node` im Feature Shortest.
- Die Methode "getWeight()" der Klasse "Edge" ist im Feature "Shortest" nicht erreichbar. Sie wird in Klasse "Edge" im Feature "Weighted" deklariert.
  - Hinzufügen von Constraint (Shortest ⇒ Weighted).
  - Umbenennen des Methodenaufrufs in Klasse `Shortest` im Feature Shortest von `getWeight()` in `getHeight()`.

Aus den Korrekturmaßnahmen kann eine globale Liste von Korrekturmaßnahmen generiert werden. Diese Liste wird zunächst nach der Häufigkeit des Vorkommens der einzelnen Korrekturmaßnahmen sortiert. Im Beispiel kommt die Maßnahme zum Hinzufügen des Constraints (Shortest ⇒ Graph) zweimal in der Liste vor, alle anderen nur einmal. Bei gleicher Anzahl von Vorkommen, werden erst die Änderungen am Quelltext, dann das Hinzufügen von nicht vorhandenen Programm-Elementen und schließlich Änderungen am Feature-Modell der Liste hinzugefügt.

Diese Sortierreihenfolge liegt darin begründet, dass bei einer großen Anzahl von gleichen Korrekturmaßnahmen die Wahrscheinlichkeit hoch ist, dass diese Korrektur viele Fehler behebt. Mit der ersten Korrekturmaßnahme lassen sich in dem Beispiel möglicherweise zwei der drei Probleme beheben. Vorschläge zum Ändern des Quelltextes werden in der Sortierung höher bewertet als Vorschläge zum Ändern des Feature-Modells, da angenommen werden kann, dass manuelle Änderungen am Feature-Modell bedachter vorgenommen werden, als manuelle Veränderungen

Vorkommen	Korrekturmaßnahme
2	Hinzufügen des Constraint (SHORTEST ⇒ GRAPH)
1	Ändern des Typs von Edge nach Edde in Klasse `Shortest` in Feature SHORTEST
1	Umbenennen des Methodenaufrufs in Klasse `Shortest` in Feature SHORTEST von `getWeight()` in `getHeight()`
1	Erzeugen neuer Klasse `Node` in Feature SHORTEST
1	Hinzufügen des Constraint (SHORTEST ⇒ WEIGHTED)

Tabelle 3.2: Die nach Relevanz sortierte, globale Liste von Korrekturvorschlägen

am Quelltext. Wird dieses Vorgehen auf die Liste von Korrekturmaßnahmen angewandt, ergibt sich die in Tabelle 3.2 gezeigt globale Liste von Korrekturmaßnahmen.

Die lokalen Listen von Korrekturmaßnahmen der einzelnen Probleme werden nach der globalen Liste sortiert. Da die Listen der ersten beiden Probleme bereits nach der globalen Liste sortiert sind, ist eine Umsortierung dieser Listen nicht nicht nötig. Das Hinzufügen des Constraint (SHORTEST ⇒ WEIGHTED) steht in der globalen Liste unter dem Umbenennen des Methodenaufrufs in Klasse `Shortest` im Feature SHORTEST von `getWeight()` in `getHeight()`, deshalb werden beide Korrekturmaßnahmen getauscht, so dass sich folgende lokale Liste für das dritte Problem ergibt:

- Die Methode “getWeight()” der Klasse “Edge” ist im Feature “Shortest” nicht erreichbar. Sie wird in Klasse “Edge” im Feature “Weighted” deklariert.
  - Umbenennen des Methodenaufrufs in Klasse `Shortest` im Feature SHORTEST von `getWeight()` in `getHeight()`.
  - Hinzufügen des Constraint (SHORTEST ⇒ WEIGHTED).

3.4 Zusammenfassung

In diesem Kapitel wurden mit Composer- und Compilerfehlern verschiedene Arten von Fehlern vorgestellt, die ein Typchecker für eine Software-Produktlinie erkennen sollte. Des Weiteren wurden verschiedene Möglichkeiten vorgestellt, wie diese Fehler korrigiert werden können. Syntaxfehler lassen sich nur durch Änderungen am Quelltext beheben, alle anderen Fehler zusätzlich durch das Hinzufügen von Constraints zum Feature-Modell. Die vorgestellten Compilerfehler können sich auch durch eine Kombination von Änderungen am Quelltext und am Feature-Modell beheben lassen.

Des Weiteren wurde mit Feature Featherweight Java (FFJ) beschrieben. FFJ ist eine Java ähnliche, funktionale, Feature-orientierte Programmiersprache, mit einem, im Vergleich zu Java, deutlich reduzierten Funktionsumfang und zusätzlichen Schlüsselwörtern zum Verfeinern von Klassen und Methoden [AKGL10]. Das Typsystem von FFJ stellt sicher, dass ein FFJ-Programm wohlgetypt ist. Die Erweiterung von FFJ zu FFJ_{PL} ermöglicht es, auch Software-Produktlinien zu erstellen und einer Typprüfung zu unterziehen. Die Prüfung wird auf der Basis der Feature-Module und des Feature-Modells durchgeführt. Es wird nicht jedes mögliche Produkt erzeugt und

geprüft. Die Hilfskonstrukte, die für das Typsystem für FFJ_{PL} vorgestellt wurden, können in ähnlicher Weise auch für das Konzept des Typcheckers für FeatureHouse verwendet werden. Es werden Teile der Funktionalität der Klassentabelle, der Einführungstabelle und der Verfeinerungstabelle übernommen.

Im letzten Abschnitt dieses Kapitels wurde das Konzept für einen Typchecker für Feature-orientierte Software-Produktlinien vorgestellt. Der Typchecker arbeitet in drei Phasen. In der ersten Phase, der *Vorverarbeitung des Quelltextes*, wird aus dem Quelltext der SPL ein abstrakter Syntaxbaum erzeugt. Der abstrakte Syntaxbaum wird genutzt, um eine Klassentabelle zu erstellen, die es ermöglicht effizient an Informationen zu eingeführten Klassen, Methoden und Feldern zu gelangen. In der zweiten Phase, der *Typprüfung und Problemsammlung*, werden die eigentlichen Tests durchgeführt. Es wird eine Möglichkeit vorgestellt, auf das Feature-Modell zuzugreifen und es wird eine Möglichkeit eingeführt, Fehler als Probleme mit zusätzlichen Informationen an die nächste Phase weiterzugeben. In der dritten Phase, dem *Berichten und Korrigieren*, werden diese Probleme an den Anwender weitergegeben und gegebenenfalls Maßnahmen zur Korrektur der Probleme vorgeschlagen. Hierfür wurden Meldungen für die verschiedenen Arten von Problemen eingeführt und mögliche Korrekturmaßnamen an einem konkreten Beispiel beschrieben. Außerdem wurde eine Methode vorgestellt, wie die Korrekturmaßnahmen sortiert werden können, um bei großen Projekten mit vielen Fehlern die Übersicht zu bewahren und die Korrekturmaßnahmen mit der größten Relevanz hervorzuheben.

4. Prototypische Implementierung eines Typcheckers

In diesem Kapitel wird die prototypische Implementierung des im letzten Kapitel vorgestellten Typcheckers beschrieben. Hierfür werden zunächst die Werkzeuge vorgestellt auf denen der Prototyp basiert und auf deren Schnittstellen zurückgegriffen wird. Im Weiteren wird auf die Umsetzung der einzelnen Phasen des Konzeptes eingegangen. Der Typchecker ist so implementiert, dass neue Checks leicht als Check-Plugin eingebunden werden können. Das Kapitel schließt mit der Beschreibung ab, wie sich der Typchecker um neue Typchecks in Form eines Check-Plugins erweitern lässt.

4.1 Verwendete Werkzeuge

Der Prototyp nutzt Fuji für das Parsen der Feature-Module in abstrakte Syntaxbäume. Für die Interaktion mit dem Feature-Modell und des SAT-Solvers wird FeatureIDE genutzt. Auf beide Werkzeuge wird im Weiteren genauer eingegangen.

Fuji: ein erweiterbarer Compiler für Feature-orientierte Programmierung in Java

Fuji[1] ist ein Compiler für FeatureHouse-kompatible Software-Produktlinien auf Basis von JastAddJ[2]. JastAddJ ist ein erweiterbarer Compiler für Java, der wiederum auf JastAdd basiert, einem Meta-Compiler-System[3]. Bei anderen FOP Werkzeugen wie AHEAD und FeatureHouse ist es notwendig, aus einer gegebenen Auswahl an Feature-Modulen den Quelltext eines Produktes zu erzeugen, der im Anschluss durch den Java-Compiler kompiliert wird. Fuji umgeht diesen Schritt, indem die Feature-Module im Hauptspeicher komponiert werden und direkt aus dieser Komposition Java-Bytecode erzeugt wird.

[1] http://fosd.de/fuji
[2] http://jastadd.org/web/jastaddj/
[3] http://jastadd.org

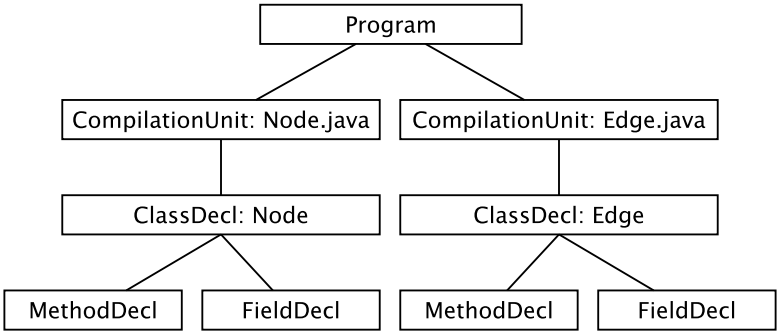


Abbildung 4.1: Aufbau eines durch Fuji erzeugten abstrakten Syntaxbaumes

Fuji stellt eine Schnittstelle bereit, mit der es möglich ist, eine beliebige Anzahl von FeatureHouse-kompatiblen Feature-Modulen zu einem abstrakten Syntaxbaum zu komponieren und auf den abstrakten Syntaxbaum der Komposition zuzugreifen. Der Schritt der Bytecode-Erzeugung kann übersprungen werden. Wie in Abbildung 4.1 zu sehen ist, hat jeder erzeugte abstrakte Syntaxbaum einen Knoten `Program` als Wurzel und Knoten vom Typ `CompilationUnit` als Kinder. Diese repräsentieren die einzelnen Java-Quelltext-Dateien der komponierten Feature-Module. Eine `CompilationUnit` kann zum Beispiel `ClassDecl`- oder `InterfaceDecl`-Knoten als Kinder haben, die für Klassen-Deklarationen beziehungsweise Interface-Deklarationen stehen. Eine Klassen-Deklaration wiederum hat Felder- und Methoden-Deklarationen. Zugriffe auf Klassen werden durch einen `TypeAccess` dargestellt. Um zu prüfen, ob der Typ existiert, auf den zugegriffen wird, muss eine passende `ClassDecl` gesucht werden.

Fuji wird für die Implementierung des Prototypen genutzt, da es ohne große Anpassungen möglich ist, ganze Feature-Module in einen abstrakten Syntaxbaum zu überführen und mit diesem zu arbeiten. Der abstrakte Syntaxbaum ist detailliert genug, um auf für Typchecks relevante Daten zuzugreifen. Während des Parsens findet Fuji außerdem Syntaxfehler, die dem Nutzer, durch die Informationen die Fuji liefert, detailliert präsentiert werden können.

FeatureIDE

Eclipse[4] ist eine integrierte Entwicklungsumgebung, die durch Plugins erweiterbar ist. Es werden offiziell verschiedene Programmiersprachen unterstützt, unter anderem Java und C/C++. FeatureIDE[5] ist eine Erweiterung für Eclipse, mit der Software-Produktlinien modelliert und implementiert werden können. Die Modellierung erfolgt durch eine graphische Benutzerschnittstelle mit der das Feature-Modell einer Software-Produktlinie erstellt und bearbeitet werden kann. FeatureIDE stellt eine Schnittstelle bereit, um das Feature-Modell auch unabhängig von der GUI bearbeiten zu können. Ein Feature-Modell kann neu erstellt oder aus verschiedenen Dateiformaten importiert werden. Es kann auf die Features und die Constraints zugegriffen werden, Konfigurationen erstellt und auf Gültigkeit geprüft werden. FeatureIDE stellt außerdem den Zugriff auf einen SAT-Solver bereit, mit dem es möglich

[4] http://www.eclipse.org
[5] http://fosd.net/fide

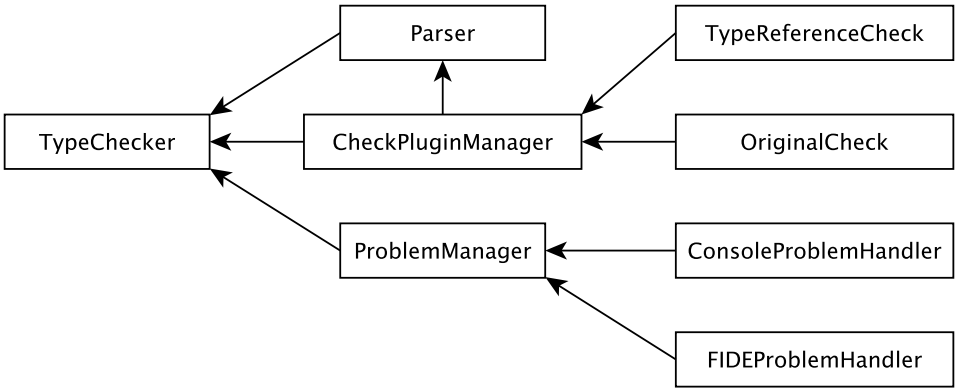


Abbildung 4.2: Aufbau und Abhängigkeiten des Prototypen

ist, eine Anfrage, wie das Prädikat $implies(\Phi, \overline{\Omega})$, das in Abschnitt 3.3.2 beschrieben ist, an das Feature-Modell zu stellen.

Für die Implementierung von Software-Produktlinien stehen in FeatureIDE verschiedene Werkzeuge zur Verfügung. Es werden unter anderem Feature-orientierte Software-Produktlinien mit AHEAD oder FeatureHouse, Aspekt-orientierte Software-Produktlinien mit AspektJ und Delta-orientierte Software-Produktlinien mit DeltaJ unterstützt.

FeatureIDE wird im Kern des Typcheckers zum einen für die Bereitstellung des SAT-Solvers und zum anderen für die Interaktion mit dem Feature-Modell verwendet. Der Typchecker kann allerdings auch vollständig in FeatureIDE integriert werden.

4.2 Aufbau des Prototypen

Die wichtigsten Komponenten des Typcheckers sind die Klassen `Parser`, `CheckPluginManager` und `ProblemManager`, sowie die Klasse `TypeChecker`, die für den Ablauf des Typcheckers zuständig ist. Abbildung 4.2 zeigt den Aufbau und die Abhängigkeiten der einzelnen Komponenten der Implementierung. Die einzelnen Typchecks sind als Plugins implementiert (`TypeReferenceCheck` und `OriginalCheck` in Abbildung 4.2). Dies erleichtert die Erweiterung des Typcheckers um neue Typchecks. Die Check-Plugins werden durch den Plugin-Manager verwaltet. Der Parser ist für die Überführung des Quelltextes in abstrakte Syntaxbäume zuständig. Der Plugin-Manager ist für die Durchführung der eigentlichen Typprüfung zuständig. Außerdem sammelt der Plugin-Manager die Probleme ein, die während der Typprüfung von den Typchecks gemeldet werden. Die gesammelten Probleme werden an den Problem-Manager übergeben, der mögliche Maßnahmen ermittelt und diese mit einer Fehlerbeschreibung auf verschiedenen Wegen ausgeben kann. Die Handhabung der Probleme erfolgt durch die Problem-Handler (`ConsoleProblemHandler` und `FIDEProblemHandler` in Abbildung 4.2).

4.2.1 Vorverarbeitung des Quelltextes

Die Phase *Vorverarbeitung des Quelltextes* (siehe Abschnitt 3.3.1) übernimmt im Prototyp im Wesentlichen die Klasse `Parser`. Diese erzeugt mit der Hilfe von Fuji

einen abstrakten Syntaxbaum für jedes Feature-Modul. Jedes Check-Plugin kann sich für bestimmte Knotentypen des abstrakten Syntaxbaumes registrieren, um während des Parsens Zugriff auf alle Knoten dieses Typs zu erhalten. Aus diesen Daten kann das Check-Plugin benötigte Hilfsstrukturen erzeugen.

Bei mehrfachem Aufruf des Typcheckers aus FeatureIDE heraus, ist der Parser, wenn von dem Anwender so gewählt, in der Lage, festzustellen, in welchen Feature-Modulen sich Dateien geändert haben. Dies macht es möglich, nur die Feature-Module neu parsen zu müssen, die sich geändert haben. Dieses Vorgehen bedeutet eine erhebliche Leistungssteigerung und ermöglicht es, den Typchecker nach einem einmaligen kompletten Typcheck auch bei jeder Änderung am Quelltext der Software-Produktlinie laufen zu lassen.

Fuji ist in der Lage Syntaxfehler in den Feature-Modulen festzustellen. Treten Syntaxfehler auf, werden diese vom Parser erkannt und der Typchecker hat die Möglichkeit auf diese zu reagieren. Sobald Syntaxfehler in den Feature-Modulen auftreten, werden diese durch den Problem-Manager behandelt und der Durchgang des Typcheckers wird abgebrochen.

4.2.2 Typprüfung und Problemsammlung

Die Klasse `CheckPluginManager` übernimmt die Rolle der im Konzept vorgestellten Phase *Typprüfung und Problemsammlung* (siehe Abschnitt 3.3.2). Bei der Initialisierung des Plugin-Managers können ein oder mehrere Check-Plugins übergeben werden. Diese Check-Plugins stellen die einzelnen Typchecks bereit, die während der Typprüfung ausgeführt werden sollen. Der Plugin-Manager hat, neben der Bereitstellung der Baumknoten in der ersten Phase zum Erzeugen der Hilfsstrukturen, die Aufgabe, die Check-Plugins zu verwalten und die eigentliche Typprüfung anzustoßen. Jeder Typcheck ist in zwei Phasen geteilt. In der *Initialisierung* wird dem Check-Plugin die Möglichkeit gegeben, auf Knoten zuzugreifen, die während des Parsens für das Check-Plugin gesammelt wurden. Dies ermöglicht dem Check-Plugin Hilfsstrukturen zum effizienten Zugriff auf diese Daten zu erstellen. Die zweite Phase ist die Ausführung des eigentlichen Typchecks. Alle während des Checks gefunden Fehler werden als Problem mit weiteren Informationen (Art des Fehlers, Informationen zu möglichen Lösungen, Position des Fehlers im Quelltext...) versehen und zur späteren Verarbeitung gespeichert.

4.2.3 Berichten und Korrigieren

Die Klasse `ProblemManager` ist für die Durchführung der dritten Phase des Konzeptes, dem *Berichten und Korrigieren* (siehe Abschnitt 3.3.3), zuständig. In dieser Phase werden alle gefundenen Probleme von dem Plugin-Manager übernommen. Das Check-Plugin, das ein Problem festgestellt hat, ist auch dafür zuständig, die Korrekturmaßnahmen zu ermitteln. Dieser Vorgang wird durch den Problem-Manager angestoßen. Um diese Probleme zu handhaben, können Problem-Handler, die das Interface `IProblemHandler` implementieren, genutzt werden. Die Aufgabe eines Problem-Handlers ist es, dem Anwender das Problem zu erklären und die Korrekturmaßnahmen vorzustellen. Ein Problem-Handler kann auch genutzt werden, um diese Korrekturmaßnahmen umzusetzen. In der aktuellen

Version liegen dem Prototypen die Problem-Handler `ConsoleProblemHandler` und `FIDEProblemHandler` bei. Der `ConsoleProblemHandler` gibt die Problemmeldung und die Korrekturmaßnahmen als Textnachricht aus, während der `FIDEProblemHandler` die Problemmeldung als Fehlermarker in FeatureIDE ausgibt.

Eine vollständige Meldung eines Fehlers an den Nutzer besteht aus der Problemmeldung einschließlich aller ermittelten Korrekturmaßnahmen. Für den Fehler einer nicht vorhandenen Klasse aus Abbildung 3.6 (Seite 17) sieht dies, wie in Abschnitt 3.3.3 beschrieben, wie folgt aus:

```
Auf die Klasse "Edge" kann aus dem Feature "Graph" nicht zugegriffen
werden. Eine Klasse "Edge" wird in Feature "Weighted" deklariert.

Folgende Maßnahmen könnten den Fehler beheben:
1. Hinzufügen des Constraints
    (Graph impliziert Weighted)
2. Ändern der genutzten Klasse von "Edge" in die vorhandene
    Klasse "Edde"
```

4.3 Aufruf des Prototypen

Der implementierte Typchecker lässt sich auf zwei unterschiedliche Arten aufrufen. Er ist zum einen in FeatureIDE integriert, lässt sich alternativ aber auch von der Kommandozeile starten. Für den Start des Typcheckers muss die Klasse `TypeChecker` instanziiert werden.

Klasse `TypeChecker`

Die Klasse `TypeChecker` ist für den Ablauf des Typcheckers zuständig. Um den Typchecker möglichst einfach erweiterbar zu machen, können der Klasse `TypeChecker` eine Liste an Check-Plugins und eine Liste mit Problem-Handlern übergeben werden. Dieses Vorgehen macht es möglich, die Funktionalität des Typcheckers zu erweitern, ohne den Quelltext der Klasse `TypeChecker` zu ändern. So können neu implementierte Check-Plugins hinzugefügt werden oder die Integration in eine integrierte Entwicklungsumgebung mit Hilfe eines Problem-Handlers vollzogen werden.

Diese Variabilität erlaubt es, nur die Typchecks auszuführen und nur die Problem-Handler zu verwenden, die gerade benötigt werden. Es ist möglich einzelne Typchecks oder eine beliebige Menge der zur Verfügung stehenden Typchecks auszuführen. Dies kann zum Beispiel nützlich sein, sollte ein bestimmtes Check-Plugin zu viel Zeit benötigen, oder der Typcheck für einen Durchgang uninteressant sein. Variable Problem-Handler machen es möglich, nur die Funktionalität zu laden, die auch benötigt wird. Wird der Typchecker von der Kommandozeile aufgerufen, ist es im Allgemeinen nicht sinnvoll, den Problem-Handler für die Integration in FeatureIDE zu laden.

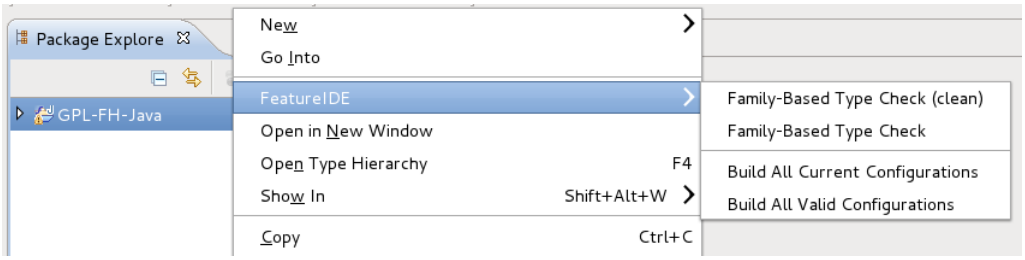


Abbildung 4.3: Aufruf des Typcheckers in FeatureIDE

4.3.1 FeatureIDE Plugin

Der Typchecker wurde als FeatureIDE-Plugin entwickelt und lässt sich auch in FeatureIDE nutzen. Der Aufruf erfolgt bei einem FeatureIDE-Projekt über das FeatureIDE-Kontextmenü des Projektes. Wie in Abbildung 4.3 zu sehen, stehen die zwei Befehle *Family-Based Type Check* und *Family-Based Type Check (clean)* zur Verfügung. Der Aufruf von *Family-Based Type Check* startet einen inkrementellen Durchgang des Typcheckers und parst nur Features, die geändert wurden. Dies kann bei großen Software-Produktlinien zu einer deutlich kürzeren Laufzeit führen, belegt allerdings Hauptspeicher, da alle Daten zu den Feature vorgehalten werden. Mit *Family-Based Type Check (clean)* werden alle Daten zu vorherigen Durchläufen des Typcheckers freigegeben und alle Features müssen erneut geparst werden.

Die Einbindung von gefundenen Problemen in FeatureIDE beschränkt sich auf einfache Fehlermarker in der Quelltext-Ansicht von FeatureIDE. In FeatureIDE wäre es weiterhin möglich, dem Nutzer Korrekturmaßnahmen am Quelltext vorzuschlagen. Diese Vorschläge fänden sich direkt an der Stelle im Quelltext, an der der Fehler auftritt und könnten durch einen Klick auf den Vorschlag übernommen werden. Diese Funktionalität ließe sich auch für die FeatureIDE-Integration des Typcheckers implementieren.

4.3.2 Aufruf ohne graphische Benutzeroberfläche

Eine weitere Möglichkeit den Typchecker aufzurufen, ist die von der Kommandozeile. Der Typchecker kann mit allen Abhängigkeiten aus Eclipse als ausführbare .jar-Datei exportiert werden. Nach dem Export kann der Typchecker mit JAVA von der Kommandozeile gestartet werden. Nötige Parameter sind die .jar-Datei (in diesem Fall spl_typechecker.jar), das Feature-Modell der zu prüfenden Software-Produktlinie und der Pfad zu dem Ordner, in dem sich der Quelltext der einzelnen Feature-Module befindet. Liegt eine Feature-orientierte Software-Produktlinie in FeatureHouse-kompatiblem Java vor, deren Feature-Modell in der Datei *spl/model.xml* gespeichert ist und deren Feature-Module sich im Ordner *spl/features* befinden, sieht der Aufruf wie folgt aus:

```
java -jar spl_typecheck.jar spl/model.xml spl/features
```

Die Ausgabe möglicher Probleme erfolgt durch die Standardausgabe.

```
1 public TypeReferenceCheck(){
2     plugin_name = ¨TypeReferenceCheck Plugin¨;
3     registerNodeType(CompilationUnit.class);
4 }
```

Abbildung 4.4: Der Konstruktor des Check-Plugins `TypeReferenceCheck`

4.4 Implementierung eines Check-Plugins

Um die Implementierung eines Typchecks als Check-Plugin zu demonstrieren, wird hier die Implementierung des Check-Plugins `TypeReferenceCheck` besprochen. Dieser Typcheck prüft, ob alle Klassen, auf die in einem Feature-Modul zugegriffen werden, auch in jeder Variante vorhanden sind.

Um ein Check-Plugin zu implementieren muss die abstrakte Klasse `AbstractTypeCheckPlugin` erweitert und mindestens die Methoden `init()` und `invokeCheck()` implementiert werden. Die Methode `init()` wird dabei vor dem eigentlichen Check ausgeführt, um es dem Check-Plugin zu ermöglichen Hilfsstrukturen zu erstellen. Die Methode `invokeCheck()` enthält die eigentliche Funktionalität des Typchecks.

Im Konstruktor kann sich das Check-Plugin für bestimmte Knoten-Typen des abstrakten Syntaxbaums registrieren. Dafür wird die Methode `registerNodeType()` der Superklasse `AbstractTypeCheckPlugin` im Konstruktor des Check-Plugins mit dem `class`-Objekt des gewünschten Knotentyps als Parameter aufgerufen. Der Name, den das Check-Plugin mit der Methode `getName()` zurück gibt, kann durch das Ändern des Feldes `plugin_name` geändert werden. Der Name des Check-Plugins wird bei der Problemmeldung als Quelle des Problems benötigt.

Der Quelltext in Abbildung 4.4 zeigt den Konstruktor der Klasse `TypeReferenceCheck`. Der Name des Check-Plugins wird gesetzt und das Check-Plugin für die Knoten vom Typ `CompilationUnit` registriert. Hierfür wird das `class`-Objekt der Klasse `CompilationUnit` an die Methode `registerNodeType()` übergeben. Die Knoten vom Typ `CompilationUnit` werden benötigt um alle Typzugriffe und alle eingeführten Typen in einer Quelltext-Datei zu erhalten.

In den durch `AbstractTypeCheckPlugin` implementierten Methoden werden während des Parsens alle Knoten vom Typ `CompilationUnit` eingesammelt. Das Check-Plugin erhält in der Initialisierungsphase des Typchecks in der Methode `init()` die Möglichkeit, auf diese Knoten zuzugreifen. Der Zugriff erfolgt durch die Methode `getNodesByType()`, die von `AbstractTypeCheckPlugin` bereitgestellt wird. Der Aufruf `getNodesByType(CompilationUnit.class)` gibt alle gesammelten Knoten vom Knotentyp `CompilationUnit`, sortiert nach Feature, zurück. Um die Einführungstabelle zu erstellen, wird für jedes Feature jeder `CompilationUnit`-Knoten durchgegangen. Es werden alle Typdeklarationen eingesammelt, die vom Knotentyp `ReferenceType` sind. Der Knoten vom Knotentyp `ReferenceType` beinhaltet unter anderem Klassen- und Interface-Deklarationen. Die `ReferenceType`-Knoten werden, ebenfalls nach Feature sortiert, in der Einführungstabelle abgelegt. Da anonyme und lokale Klassen von außen nicht erreichbar sind, werden diese eben-

```
1  public void init() {
2      Map<Feature, List<CompilationUnit>> cumap =
3              getNodesByType(CompilationUnit.class);
4
5      for (Feature f : cumap.keySet()) {
6          for (CompilationUnit cu : cumap.get(f)) {
7              for (TypeDecl td : cu.getTypeDecls()) {
8                  if (td instanceof ReferenceType) {
9                      ReferenceType rt = (ReferenceType) td;
10                     if (!(rt.isAnonymous() || rt.isLocalClass() || rt
11                         .isArrayDecl())) {
12                         intros.get(f).add(rt);
13                     }
14                 }
15             }
16         }
17     }
18 }
```

Abbildung 4.5: Die Methode `init()` des Check-Plugins `TypeReferenceCheck`

```
1  protected Map<Feature, ReferenceType> providesType(String type) {
2      Map<Feature, ReferenceType> providing_features =
3              new HashMap<Feature, ReferenceType>();
4
5      for (Feature f : intros.keySet()) {
6          for (ReferenceType rt : intros.get(f)) {
7              if (rt.name().equals(type)) {
8                  providing_features.put(f, rt);
9              }
10         }
11     }
12     return providing_features;
13 }
```

Abbildung 4.6: Die Methode `providesType(String type)` des Check-Plugins `TypeReferenceCheck`

so ignoriert, wie Array-Deklarationen, da diese in der Vererbungshierarchie vom Knotentyp `ReferenceType` sind.

Um in der Check-Phase des Typchecks Anfragen an die Einführungstabelle stellen zu können, wird eine zusätzliche Methode `providesType(String type)` eingeführt, die im Quelltext in Abbildung 4.6 zu sehen ist. Die Methode `providesType` durchsucht die Einführungstabelle und gibt alle Features zurück, die einen Typ mit dem gesuchten Namen bereitstellen. Neben dem Feature, in dem der Typ bereitgestellt wird, wird auch die gefundene Typdeklaration zurückgegeben. Die Deklaration kann genutzt werden, um weitere Informationen über die gefundene Klasse zu erhalten.

Der Quelltext in Abbildung 4.7 zeigt die Implementierung der Methode `invokeCheck()` des Typchecks. Es wird zunächst über alle Features und alle `CompilationUnit`-Knoten iteriert, die während des Parsens gesammelt wurden. Im nächsten Schritt werden alle Kindknoten vom Knotentyp

```
1  public void invokeCheck(FeatureModel fm) {
2      Map<Feature, List<CompilationUnit>> cumap =
3              getNodesByType(CompilationUnit.class);
4
5      for (Feature f : cumap.keySet()) {
6          for (CompilationUnit cu : cumap.get(f)) {
7              for (TypeAccess ta : FujiWrapper.getChildNodesByType(cu,
8                  TypeAccess.class)) {
9                  if (ta.type() instanceof UnknownType) {
10                     Set<Feature> providing_features = providesType(
11                         ta.name()).keySet();
12                     if (!checkFeatureImplication(fm, f,
13                             providing_features)) {
14                         // melde ein neues Problem
15                     }
16                 }
17             }
18         }
19     }
20 }
```

Abbildung 4.7: Die Methode `invokeCheck(FeatureModel fm)` des Check-Plugins `TypeReferenceCheck`

`TypeAccess` einer jeder Klasse gesucht. Diese Suche wird mit Hilfe der Methode `FujiWrapper.getChildNodesByType(ASTNode, Class Knotentyp)` durchgeführt, die rekursiv alle Kindknoten eines Knotens durchsucht und die Kindknoten zurück gibt, die dem gesuchten Knotentyp entsprechen. Für jeden gefundenen Typzugriff wird geprüft, ob dieser bereits durch Fuji aufgelöst werden konnte. Ist dies der Fall, gibt die Methode `TypeAccess.type()` die von Fuji gefundene Typdeklaration zurück, andernfalls wird ein Knoten vom Typ `UnknownType` zurückgegeben. Im letzteren Fall kann auf den Namen der Klasse mit der Methode `name()` zugegriffen werden. Die Methode `providesType()` wird mit dem Namen des Typs aufgerufen, um an die Features zu gelangen, die den gesuchten Typ bereitstellen. Der letzte Schritt des Typchecks ist die Anfrage an das Feature-Modell, ob eines der Features, die den Typ bereitstellen können, immer gewählt ist, wenn das Feature mit der Typreferenz gewählt ist. Gibt es eine Variante, in der das Feature mit der Typreferenz ohne eines der Features gewählt ist, das den Typ bereit stellt, wird ein neues `CheckProblem` mit den nötigen Informationen zu dem Fehler erzeugt.

Weitere Check-Plugins

Neben dem im letzten Abschnitt vorgestellten Check-Plugin zum Prüfen von Typreferenzen wurde ein Check-Plugin zum Überprüfen von `original()`-Aufrufen implementiert. Das Schlüsselwort `original()` wird genutzt, um eine in der Komponierreihenfolge vor dem Feature-Modul des `original()`-Aufrufes eingeführte Version der aufrufenden Methode aufzurufen. Der original()-Check prüft, ob für jede Nutzung des Schlüsselwortes `original()` auch eine aufrufbare Methode existiert. Außerdem wird geprüft, ob eines der deklarierenden Feature-Module immer gewählt ist, wenn das Feature-Modul des `original()`-Aufrufes gewählt ist.

4.5 Zusammenfassung

In diesem Kapitel wurde die prototypische Implementierung eines Typcheckers nach dem in Kapitel 3 vorgestellten Konzept beschrieben. Es wurden mit Fuji und FeatureIDE die Werkzeuge vorgestellt, auf die der Typchecker aufbaut und deren Schnittstellen für die Implementierung genutzt werden. Die Implementierung übernimmt grundsätzlich die Teilung der Typprüfung in drei Phasen aus dem Konzept. Jede Phase wird durch eine Klasse ausgeführt. Die Klasse `Parser` parst den Quelltext der Feature-Module und erstellt die abstrakten Syntaxbäume (*Vorverarbeitung des Quelltextes*, siehe Abschnitt 3.3.1). In dem Prototypen wird jeder Typcheck in einem eigenen Check-Plugin implementiert. Die Verwaltung dieser Check-Plugins übernimmt die Klasse `CheckPluginManager`. Diese Klasse ist auch für das Durchführen der Typchecks und das Sammeln aller durch die Check-Plugins gemeldeten Probleme zuständig (*Typprüfung und Problemsammlung*, siehe Abschnitt 3.3.2). Die gesammelten Probleme werden von der Klasse `ProblemManager` ausgewertet und es werden Korrekturmaßnahmen ermittelt. Diese werden durch Problem-Handler gehandhabt, die ebenfalls als Plugins implementiert sind. Diese Problem-Handler können die Probleme über verschiedene Kanäle ausgeben und die Korrekturmaßnahmen umsetzen (*Berichten und Korrigieren*, siehe Abschnitt 3.3.3). Der Typchecker kann in FeatureIDE integriert werden, aber auch ohne eine graphische Benutzeroberfläche aufgerufen werden.

Durch die Implementierung von Typchecks und Problem-Handlern als Plugins, lässt sich der Prototyp zum einen leichter durch neue Funktionalität erweitern. Zum anderen wird der Ansatz sehr flexibel, da es möglich ist, nur die Plugins zu laden, die auch gebraucht werden. Wird der Typchecker von der Kommandozeile aufgerufen, ist die Funktionalität der Einbindung in FeatureIDE nicht notwendig. Der letzte Abschnitt dieses Kapitels hat sich damit beschäftigt zu zeigen, wie der Typchecker um eigene Checks erweitert werden kann.

5. Evaluierung

Im letzten Kapitel wurde die prototypische Implementierung des in dieser Arbeit vorgestellten Typcheckers beschrieben. In diesem Kapitel wird die Funktionsweise des Prototypen evaluiert und mit einem Produkt-basierten Ansatz verglichen.

5.1 Vorgehensweise

Das vorgestellte Konzept wird evaluiert, indem die Ergebnisse eines naiven, Produkt-basierten Vorgehens mit denen des Typcheckers verglichen werden. Der naive Ansatz eine Software-Produktlinie auf Typsicherheit zu prüfen, besteht aus dem Erzeugen aller gültiger Konfigurationen, dem Komponieren des Quelltextes für jede Variante und dem anschließenden Kompilieren jeder Variante. FeatureIDE stellt diese Funktionalität zur Verfügung. Für die Auswertung wird die Gesamtzeit, sowie die Zeit, die jeweils zum Komponieren und zum Kompilieren benötigt wird, gemessen. Die Differenz zwischen der Gesamtzeit und der Komponier- und Kompilierzeit ergibt sich im Wesentlichen durch das Erzeugen gültiger Konfigurationen. Um die Ergebnisse der Typprüfung vergleichen zu können, wird neben den Zeiten auch die Ausgabe des Compilers gespeichert.

Der Prototyp des in dieser Arbeit vorgestellten Typcheckers wird auf die gleichen Software-Produktlinien angewandt. Es werden die benötigte Zeit sowie die gefundenen Probleme aufgezeichnet. Die Zeiten, die der Typchecker benötigt, sind sehr klein. In diesen Dimensionen kann es leicht zu Ausreißern kommen, die die Ergebnisse verfälschen. Aus diesem Grund wird jeder Vorgang fünfmal gemessen und der Durchschnitt dieser Messungen festgehalten. Zum Vergleich mit dem naiven Ansatz werden neben den Zeiten auch die jeweils gemeldeten Probleme herangezogen.

Da der Oracle Java-Compiler nach den Java-Spezifikationen umgesetzt worden ist, kann davon ausgegangen werden, dass alle Typfehler in allen Varianten gefunden werden. Kann ein Typchecker alle Fehler eines Fehlertyps finden, es werden also keine Fehler *nicht* gefunden (engl. *false negatives*), wird der Typchecker als vollständig (engl. *complete*) bezeichnet. Sind alle von dem Typchecker gefundenen Fehler auch

SPL	Features (konkret)	Varianten	LOC
zipME	17 (13)	32	4988
GPL	38 (27)	840	2955
Notepad	16 (13)	1024	2727

Tabelle 5.1: Informationen über die für die Evaluierung genutzten Software-Produktlinien

wirklich Fehler, das heißt kein vom Typchecker gefundener Fehler wird vom Java-Compiler *nicht* gefunden (engl. *false positives*), arbeitet der Typchecker korrekt (engl. *sound*). Der Sun Java-Compiler erfüllt diese beiden Eigenschaften.

Ein Typchecker wird als ideal bezeichnet, wenn er vollständig und korrekt ist. Durch die Komplexität der Java-Spezifikationen ist dies allerdings nur schwer für Software-Produktlinien umsetzbar. Erstrebenswert ist allerdings, dass jeder für den Typchecker implementierte Check diese Eigenschaften erfüllt. Das bedeutet, der Check, der prüft, ob alle Typen auf die zugegriffen wird auch vorhanden sind, findet zwar *alle* nicht auflösbaren Typreferenzen, aber auch *nur* alle nicht auflösbaren Typreferenzen.

5.2 Testumgebung der Evaluierung

Als Testumgebung dient ein Rechner mit einer Dual-Core Intel Core i5 CPU mit einer Taktfrequenz von 3,33GHz, sowie 8GB Hauptspeicher. Als Betriebssystem wurde Windows 7 Professional und als Java-Laufzeitumgebung wurde die Oracle JRE in der Version 1.6.0_31 verwendet. Die Tests wurden mit einer aktuellen Version (Revision 1708) von FeatureIDE aus dem SVN-Repository ausgeführt. Beide Tests, der naive Ansatz und die Typprüfung mit der Implementierung des Typcheckers, können mit dieser Version durchgeführt werden. Die Software-Produktlinien liegen als FeatureIDE-Projekte in FeatureHouse-kompatiblem Java und mit einem Feature-Modell vor.

In Tabelle 5.1 sind die für die Evaluierung genutzten Software-Produktlinien aufgeführt. Die erste zu testende SPL ist eine zerlegte Version der Java-Pack-Bibliothek zipME[1]. Diese liegt Fuji als Beispiel bei und konnte ohne großen Aufwand als FeatureIDE-Projekt importiert werden. Die SPL hat 13 konkrete Features und das Feature-Modell erlaubt 32 Varianten. Als zweite SPL wird die Graphen-Produktlinie getestet, die FeatureIDE als Beispielprojekt beiliegt. Diese hat 27 konkrete Features und das Feature-Modell erlaubt 840 Varianten. Bei der dritten Software-Produktlinie handelt es sich um eine Notepad-SPL. Diese liegt ebenfalls FeatureIDE als Beispielprojekt (Notepad-FH-Java-2) bei. Die Notepad-SPL hat 13 konkrete Features und 1024 gültige Konfigurationen.

Für diese Arbeit wurden zwei Check-Plugins für den Typchecker implementiert. Der `TypeReferenceCheck` findet alle lokal nicht auflösbaren Typreferenzen in einem Feature-Modul, die auch, durch von anderen Feature-Modulen eingeführte Klassen, nicht in jeder Variante aufgelöst werden können. Dies schließt fehlende Importe der

[1] http://zipme.sourceforge.net/

SPL	Konfigurationen erzeugen	Komponieren	Kompilieren	Gesamt	Pro erzeugtes Produkt
zipMe	30s	48s	4s	1m22s	2577ms
GPL	41m09s	2m49s	1m4s	45m02s	3217ms
Notepad	40s	8m20s	1m59s	10m59s	664ms

Tabelle 5.2: Ergebnisse der Evaluierung des naiven Ansatzes

Systembibliothek und anderer externer Bibliotheken mit ein. Wird ein gefundener Fehler durch einen fehlenden Import verursacht, muss dieser allerdings nicht zu einem Fehler in der generierten Variante führen. Ist der Import in einer anderen Rolle der Klasse zu finden, ist dieser in der Komposition der Features trotzdem vorhanden. Der zweite Typcheck `OriginalCheck` prüft für jeden Aufruf von `original()`, ob die aufrufende Methode in der Komponierreihenfolge vor dem aufrufenden Feature deklariert und immer erreichbar ist.

Es werden Zeiten für beide Typchecks einzeln und in Kombination gemessen. Für jeden dieser drei Tests wird außerdem die Zeit gemessen, die der Typchecker benötigt, wenn inkrementell geparst werden kann. Dafür wird der Zeitstempel der letzten Modifikation der ersten Quelltext-Datei eines zufälligen Feature-Moduls einer SPL verändert. Dies sorgt dafür, dass nur das veränderte Feature-Modul erneut geparst werden muss.

5.3 Auswertung der Ergebnisse

Die Auswertung der Ergebnisse erfolgt in zwei Schritten. Im ersten Schritt wird auf die aufgezeichneten Zeiten eingegangen und der Zeitunterschied der unterschiedlichen Ansätze interpretiert. Der zweite Schritt besteht darin, die Fehlermeldungen des naiven Ansatzes mit den Problemmeldungen des Typcheckers zu vergleichen.

Vergleich der gemessenen Zeiten

Tabelle 5.2 zeigt die Zeiten, die für die verschiedenen Messungen des naiven Ansatzes aufgezeichnet wurden. Obwohl die zipME-SPL genauso viele konkrete Features wie die Notepad-SPL besitzt, erlaubt das Feature-Modell von der zipME-SPL nur 32 Varianten. Dies ist deutlich an den Zeiten zu sehen, die zum Komponieren und Kompilieren benötigt wurden. Die GPL benötigt mit 45 Minuten und 2 Sekunden am meisten Zeit, wobei der Großteil der benötigten Zeit auf das Suchen nach gültigen Konfigurationen zurückgeht. Dies ist auf die große Anzahl an Features und eine nicht optimierte Implementierung der Suche nach gültigen Konfigurationen mit Hilfe eines SAT-Solvers zurückzuführen. Die Notepad-SPL benötigt den Großteil der Zeit zum Komponieren der Features. Da die meisten Features optional sind und nur wenige Constraints vorhanden sind, hat die Zeit, die zum Suchen gültiger Konfigurationen benötigt wurde, einen deutlich kleineren Anteil als bei der zipME-SPL und der GPL.

Die in Tabelle 5.3 für den Typchecker gemessenen Zeiten unterscheiden sich deutlich von denen des naiven Ansatzes in Tabelle 5.2. Jedes Feature-Modul wird bei

SPL	Typreferenz-Check	Original-Check	Typreferenz- & Original-Check (% des naiven Ansatzes)
zipMe	516ms	535ms	547ms (0,66%)
GPL	1060ms	1078ms	1152ms (0,04%)
Notepad	523ms	491ms	666ms (0,1%)

Tabelle 5.3: Ergebnisse der Evaluierung des Typcheckers

SPL	Typreferenz-Check	Original-Check	Typreferenz- & Original-Check (% des kompletten Durchgangs)
zipMe	70ms	67ms	79ms (14,4%)
GPL	80ms	67ms	101ms (8,8%)
Notepad	70ms	76ms	85ms (12,8%)

Tabelle 5.4: Ergebnisse der Evaluierung des Typcheckers bei inkrementellem Parsen

jedem Durchgang des Typcheckers maximal einmal geparst. Die Typchecks laufen dann auf Basis der Datenstrukturen, die im Hauptspeicher liegen. Das Parsen der Feature-Module und das Erzeugen der Datenstrukturen nimmt auch den Großteil der benötigten Zeit in Anspruch. Der Typchecker benötigt mit den beiden implementierten Tests maximal 0,66% (für die zipME SPL) der Zeit des naiven Ansatzes. Abbildung 5.1 vergleicht die Zeiten, die der naive Ansatz und der Typchecker (mit dem Typreferenz- und dem Original-Check), benötigt haben. Da die gemessenen Zeiten des naiven Ansatzes bis in die Millionen reichen und die Zeiten für den Typchecker nur bis in die Tausende, wurde eine logarithmische Darstellung gewählt.

Wird der Typchecker inkrementell genutzt, kann der Zeitgewinn noch deutlich größer sein. Tabelle 5.4 führt die Zeiten auf, die der Typchecker benötigt, wenn nur ein Feature-Modul neu geparst werden muss. Für diesen Test wurde die erste Quelltext-Datei eines zufälligen Feature-Moduls verändert, damit das Feature-Modul neu geparst werden muss. Der Zeitgewinn gegenüber einem kompletten Durchganges lässt sich in Abbildung 5.2 deutlich erkennen. So ist der Typcheckers, inkrementell ausgeführt, für die GPL 11,4-mal so schnell wie ein kompletter Durchgang des Typcheckers.

Die Unterschiede zwischen den Laufzeiten des Typcheckers, mit jeweils nur dem Typreferenz-Check beziehungsweise dem Original-Check, sind mit maximal 32 Millisekunden sehr klein. Der Typreferenz-Check ist in der zipME-SPL und der GPL schneller, während der Original-Check schneller beim Prüfen der Notepad-SPL ist. In der Kombination sind die Check-Plugins für alle SPLs langsamer als die beiden einzelnen Check-Plugins. Der Unterschied ist für die GPL und die Notepad-SPL deutlich, für die zipME-SPL allerdings kaum sichtbar.

Vergleich der gefundenen Fehler

Für alle drei verwendeten Beispiel-Software-Produktlinien wurden bei dem naiven Ansatz Varianten erzeugt, für die der Java-Compiler Fehler gemeldet hat.

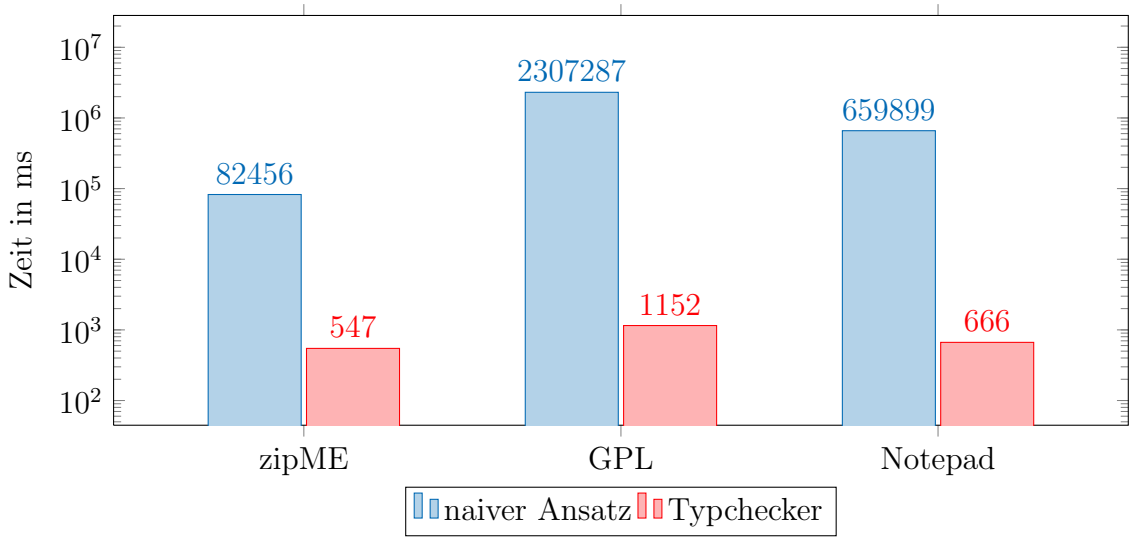


Abbildung 5.1: Ausführungszeiten des naiven Ansatzes und des Typcheckers mit Typreferenz- und Original-Check in logarithmischer Darstellung

zipME-SPL

Für die Software-Produktlinie zipME wurden für 8 der 32 Varianten Fehler vom Compiler gemeldet. Bei allen Fehlern handelt es sich um die Klasse `CRC32`, auf die in zwei Klassen zugegriffen wird, die in den fehlerhaften Varianten aber nicht erreichbar ist. Dieser Fehler wird bei der Typprüfung mit dem Typchecker von dem Typcheck für Typreferenzen gefunden:

```
Auf die Klasse "CRC32" kann aus dem Feature "GZIP" nicht zugegriffen
werden. Eine Klasse "CRC32" wird im Feature "CRC" deklariert.

Auf die Klasse "CRC32" kann aus dem Feature "DerivativeCompressGZIP"
nicht zugegriffen werden. Eine Klasse "CRC32" wird im Feature "CRC"
deklariert.
```

Insgesamt wurden für die zipME-SPL 48 Probleme gemeldet. Neben den beiden Typreferenz-Fehlern, wurden 46 fehlende Importe festgestellt. Diese führen allerdings nicht zu den durch den Java-Compiler gemeldeten Fehlern in den erzeugten Varianten.

GPL

Von den 840 erzeugten Varianten der GPL hat der Java-Compiler für 494 Varianten Fehler geliefert. In allen fehlerhaften Varianten wird die Methode `original()` nicht gefunden. Dieser Fehler kann durch eine in bestimmten Varianten nicht erreichbare oder gar nicht existierende Methode zurückgeführt werden, auf die versucht wird mit `original()` zuzugreifen. Ein Grund, warum die Methode nicht erreichbar ist, kann sein, dass das die Methode einführende Feature in der Komponierreihenfolge nach

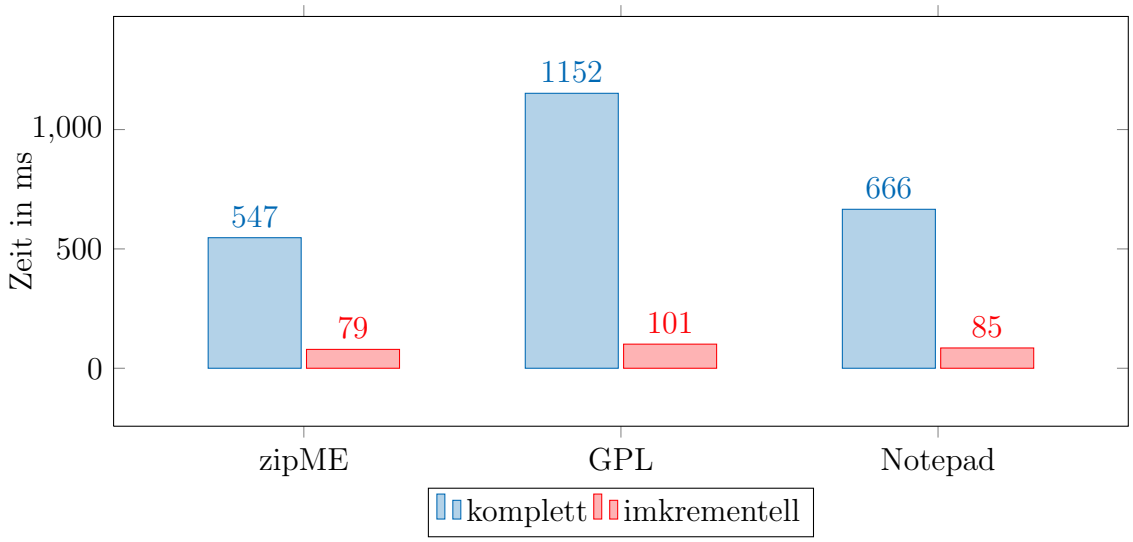


Abbildung 5.2: Vergleich der Ausführungszeiten eines kompletten Durchlaufs des Typcheckers mit der eines inkrementellen Durchlaufs

dem Feature kommt, das die Methode aufruft. Ist die Position des einführenden Features in der Komponierreihenfolge korrekt, kann es auch sein, dass das einführende Feature nicht in allen Varianten gewählt ist, in denen das aufrufende Feature gewählt ist. Bei der Komposition der Features kann das Schlüsselwort `original()` dann nicht durch den Aufruf der gesuchten Methode ersetzt werden. Der Java-Compiler findet keine Methode mit dem Namen `original()` und meldet einen Fehler. Auch das Check-Plugin, das `original()`-Aufrufe prüft, meldet Probleme. Es wird für jeden Aufruf des Schlüsselwortes `original()` eine entsprechende, vorher eingeführte Methode gefunden, allerdings sind die einführenden Features nicht immer erreichbar. Eines der gemeldeten Probleme ist das folgende:

```
Auf die Methode "Edge.adjustAdorns(EdgeIfc)" kann aus dem Feature
"WeightedWithEdges" nicht zugegriffen werden.
Die Methode "Edge.adjustAdorns(EdgeIfc)" wird in den
Features "DirectedWithEdges" und "UndirectedWithEdges" eingeführt.
```

Neben den Problemen, die der Original-Check liefert, wird vom Typreferenz-Check ein weiteres Problem wegen eines fehlenden Imports gemeldet. Dieser fehlende Import ist allerdings für keine durch den Java-Compiler gemeldeten Fehler verantwortlich.

Notepad-SPL

Für die Notepad-SPL wurden 1024 Varianten erzeugt. 512 von diesen sind fehlerhaft und alle dieser 512 fehlerhaften Varianten haben das Feature FullStyled in der Konfiguration. Die Probleme ergeben sich aus dem Umstand, dass Konstruktoren in FeatureHouse nicht überschrieben werden wie Methoden, sondern zusammengefügt werden. Da das Feature FullStyled vorhandene Konstruktoren überschreiben will, kommt es zu unterschiedlichen, Konstruktor-spezifischen, Fehlern. So

wird in den fehlerhaften Varianten versucht mehrere lokale Variablen mit dem gleichen Namen zu deklarieren oder zweimal `super()` in einem Konstruktor aufzurufen. Alle gefunden Fehler stehen im Zusammenhang mit der Behandlung von Konstruktoren in FeatureHouse. Keiner dieser Fehler wurde durch die beiden implementierten Typchecks gefunden. Der Typreferenz-Check hat allerdings 178 Probleme wegen fehlender Importe gemeldet. Diese fehlenden Importe haben nicht zu fehlerhaften Varianten geführt.

5.4 Diskussion der Ergebnisse

Der Unterschied in der Ausführungszeit zwischen dem naiven, Produkt-basierten Ansatz und der Typprüfung mit dem in dieser Arbeit vorgestellten Typchecker ist sehr groß (siehe Abbildung 5.1). Dieser Geschwindigkeitsvorteil wird vor allem dadurch erkauft, dass die durchgeführte Typprüfung nicht vollständig ist. Daher ist es wichtig, Check-Plugins bereitzustellen, die die am häufigsten auftretenden Fehler (zum Beispiel nicht immer vorhandene Klassen, Methoden und Felder) finden können. Es kann allerdings mitunter sehr aufwendig sein, diese zu implementieren, da Check-Plugins vollständig und korrekt sein sollten.

Der Typreferenz-Check hat in der zipME-SPL zwei Probleme im Zusammenhang mit fehlenden Klassen gefunden. Gleichzeitig hat der naive Ansatz 8 fehlerhafte Varianten erzeugt, die jeweils Fehler aufgrund einer fehlenden Klasse liefern. Aus den gemeldeten Probleme kann abgeleitet werden, dass der Constraint (GZIP $\Rightarrow$ CRC) das erste gefundene Problem lösen könnte. Wird dieser Constraint dem Feature-Modell hinzugefügt, ist, durch bereits im Feature-Modell vorhandene Constraints, auch das zweite Problem gelöst. Werden, nach Hinzufügen des Constraints, erneut alle Varianten erzeugt, sind alle Programme fehlerfrei und auch der Typchecker gibt keine Meldung mehr zu diesem Fehler aus. Die Typprüfung mit dem Typchecker hat dabei mit beiden verfügbaren Check-Plugins 547ms gedauert. Dies entspricht 0,66% der Zeit, die der naive Ansatz benötigt hat um alle Varianten zu testen. Negativ fällt dagegen auf, dass der gleiche Test in allen getesteten Software-Produktlinien falsche Fehler, in diesen Fällen auf Grund fehlender Importe in den Feature-Modulen, findet.

Werden mehr Typchecks implementiert, steigt unweigerlich auch die Zeit, die für die Durchführung der Typprüfung benötigt wird. Die Typprüfung hat allerdings nur einen kleinen Anteil an der Zeit, die für einen kompletten Durchgang des Typcheckers benötigt wird. Der Großteil der Zeit wird für das Parsen benötigt. Auch das Parsen wird durch das Hinzufügen von neuen Check-Plugins etwas verlängert, der Großteil des Zeitzuwachses ergibt sich allerdings aus der Prüfungsphase. Durch den Umstand, dass das Parsen einen Großteil der Zeit benötigt, ist der erste Typcheck sehr zeitintensiv. Für jeden weiteren Typcheck kann auf den gleichen abstrakten Syntaxbaum zugegriffen werden, nur das Erstellen der Hilfskonstrukte benötigt zusätzliche Zeit. Dies führt dazu, dass die durchschnittliche Zeit, die ein Typcheck benötigt, mit jedem weiteren Typcheck sinkt.

Die gesamte Ausführungszeit des Typcheckers wird mit einer zunehmenden Anzahl an Check-Plugins zunehmen. Trotzdem können mit diesem Ansatz viele Fehler, die eventuell nur in wenigen Varianten auftreten, gefunden werden und dies in, im Vergleich zu dem Komponieren und Kompilieren aller Varianten, sehr kurzer Zeit.

In den durchgeführten Tests haben alle Durchgänge des Typcheckers unter einem Prozent der Zeit benötigt, die der naive Ansatz benötigte. Diese Zeit kann durch die Möglichkeit, die Typprüfung inkrementell durchzuführen, zusätzlich noch deutlich verkürzt werden, da das Parsen der Feature-Module die meiste Zeit benötigt. Die Verwendung des Typcheckers stellt damit einen guten Kompromiss aus Geschwindigkeit und Fehlererkennung dar und ist damit eine gute Alternative zur Typprüfung von Software-Produktlinien. Allerdings sollten noch zusätzliche Typcheck implementierte werden.

6. Verwandte Arbeiten

Es gibt mehrere Arbeiten, die sich mit Typsystemen für Software-Produktlinien oder für Feature-orientierte Programmierung auseinandersetzen. Den meisten ist gemein, dass sie auf, in der Funktion reduzierten, Programmiersprachen aufsetzen.

Thaker et al. stellen ein System zur sicheren Komposition für Feature-orientierte Software-Produktlinien in AHEAD vor [TBKC07]. Zum Implementieren der Features wird Jak genutzt, eine um Features erweiterte Variante von Java. Für jedes Feature werden Constraints erzeugt, die zum Beispiel das Vorhandensein eines bestimmten Typs für ein Feature fordern. Diese Constraints müssen erfüllt sein, damit die SPL typsicher ist und können dem Feature-Modell hinzugefügt werden. Auf diesem Weg ist es nur noch möglich, typsichere Varianten zu erzeugen. Die Idee hinter dem Erzeugen der Constraints ist im Wesentlichen die gleiche, die auch für Typchecks des vorgestellten Typcheckers genutzt werden kann. Um sicherzustellen, dass eine Klasse auf die zugegriffen wird immer erreichbar ist, erstellt der Typreferenz-Check des vorgestellten Typcheckers zum Beispiel auch einen Constraint, den das Feature-Modell erfüllen muss.

Delaware et al. beschreiben ein System, das die sichere Komposition von Feature-Modulen erlaubt [DCB09]. Sichere Komposition bedeutet, dass ein Programm, das durch die Komposition von Feature-Modulen erzeugt wird, typsicher ist. Dazu wird *Lightweight Feature Java* (LFJ) eingeführt, das auf *Lightweight Java* basiert und um Features erweitert wurde. *Lightweight Java* ist eine funktionsreduzierte Untermenge von Java. Um die Typsicherheit sicherzustellen, wird eine Menge von Constraints für die Features einer Software-Produktlinie erstellt. Ein abgeleitetes Programm muss diese Constraints erfüllen, um typsicher zu sein. Die Constraints können in einen aussagenlogischen Ausdruck überführt werden, der alle Programme darstellt, die typsicher sind. Um die Typsicherheit für eine Software-Produktlinie sicherzustellen, wird aus dem Feature-Modell ebenfalls ein aussagenlogischer Ausdruck erzeugt und aus beiden Ausdrücken ein Ausdruck erzeugt, dessen Erfüllbarkeit bedeutet, dass alle Produkte der Software-Produktlinie und damit auch die Software-Produktlinie an sich typsicher sind. Dieser Ansatz lässt sich mit Hilfe eines SAT-Solvers realisieren. Für die größte getestete Software-Produktlinie benötigt eine Implementierung des

Typsystems für LFJ 30 Sekunden und damit weniger als für das Erzeugen und Kompilieren einer Variante der SPL [DCB09]. Für das Typsystem für LFJ ist bewiesen, dass es alle Fehler erkennt und nur typsichere Varianten erzeugt werden können. Die zugrunde liegende Sprache ist allerdings in den Funktionen eingeschränkt.

Ein ähnliches System für Delta-orientierte Programmierung beschreiben Schaefer et al. [SBD11]. Es wird ein Typsystem für Delta-orientierte Programmierung für *Imperative Featherweight Δ Java* (IFΔJ) vorgestellt. IFΔJ basiert auf *Imperative Featherweight Java*, einer minimalen, imperativen Variante von Java. Für jedes Delta-Modul wird eine abstrakte Repräsentation erzeugt, die aus der Signatur des Delta-Moduls und Constraints bestehen, die für ein Delta-Modul erfüllt sein müssen. Die Signatur eines Delta-Moduls besteht dabei aus dem Delta-Modul ohne Methodenrümpfe. Für alle gültigen Konfigurationen kann aus den Signaturen der Delta-Module, unter Einhaltung der Constraints, eine Klassen-Signatur-Tabelle erzeugt werden. Dies gelänge nur, wenn auch die Erzeugung des Produktes aus den Delta-Modulen gelungen wäre. Das von Schaefer et al. vorgestellte Typsystem erkennt für eine Software-Produktlinie in einer in der Funktionalität reduzierte Variante von Java alle Fehler sicher. Für jede Variante einer Software-Produktlinie muss für die Typprüfung allerdings eine Konfiguration erzeugt werden. Das Konzept des Typcherckers, das in dieser Arbeit vorgestellt wird, soll die häufigsten Fehler in einer Software-Produktlinie in FeatureHouse-kompatiblem Java finden, ohne dass es nötig ist für jede Variante eine Konfiguration zu erzeugen.

Apel et al. stellen einen Ansatz für ein Typsystem für Feature-orientierte Software-Produktlinien vor, das ohne das Erzeugen von Konfigurationen für jede einzelne Variante auskommt [AKGL10]. Als Basis wird *Feature Featherweight Java* beziehungsweise FFJ_{PL} genutzt, das auf *Featherweight Java* basiert, einer funktionsreduzierten Variante von Java. Die Interaktion mit dem Feature-Modell geschieht direkt durch das Typsystem. Das FFJ_{PL} Typsystem unterstützt sich gegenseitig ausschließende Features. Dies schließt die sich daraus ergebenden Konsequenzen ein. So ist es möglich, dass ein Term mehr als einen Typ haben kann. Das Prüfen einer SPL in FFJ_{PL} wird nicht modular durchgeführt, die gesamte Software-Produktlinie wird gleichzeitig getestet. Das Typsystem für FFJ_{PL} stellt ebenfalls sicher, dass eine Software-Produktlinie und damit auch jede Variante der SPL typsicher ist. Allerdings ist FFJ_{PL} in der Funktionalität reduziert und nicht Java-kompatibel. Zusätzlich wird bei jeder Änderung eines Feature-Moduls die ganze SPL erneut geprüft. Die prototypische Implementierung des vorgestellten Typcheckers kann die Informationen über ein Feature-Modul in einem weiteren Durchgang wiederverwenden und macht so ein erneutes Parsen des Feature-Moduls nur im Falle von Änderungen nötig.

Ein Typsystem für annotierte Software-Produktlinien stellen Kästner et al. für *Colored Featherweight Java* (CFJ) vor [KATS11]. CFJ speichert die Annotationen von Quelltextelementen nicht direkt im Quelltext, sondern in einer Annotationstabelle. Das Typsystem von CFJ stellt sicher, dass eine in CFJ implementierte Software-Produktlinie typsicher ist. Das formal für CFJ bewiesene Typsystem wurde praktisch um Typchecks erweitert, die es ermöglichen, auch annotierte Software-Produktlinien in Java zu testen und die wichtigsten Fehler zu finden. Damit ähnelt das Typsystem von Kästner et al. dem Ansatz dieser Arbeit grundsätzlich, beschäftigt sich

aber im Gegensatz zu dieser Arbeit mit annotierten Software-Produktlinien, nicht mit Feature-orientierten. Des Weiteren bauen die Typchecks auf CIDE, einer Erweiterung für Eclipse ähnlich FeatureIDE, auf. Der hier vorgestellte Typchecker läuft grundsätzlich auch unabhängig von FeatureIDE, lässt sich aber in FeatureIDE integrieren. Die Überprüfung einer annotierten Software-Produktlinie dauert etwa 10-mal solange wie die Überprüfung einer einzelnen Variante [KATS11]. Die prototypische Implementierung des in dieser Arbeit vorgestellten Konzeptes brauchte, je nach getesteter Software-Produktlinie, 1-5-mal solange für die Überprüfung der ganzen SPL, als für die Prüfung einer einzelnen Variante.

Mit *g*Deep stellen Apel und Hutchins ein System vor, mit dem es möglich ist, Artefakte unterschiedlicher Sprachen als Module in Feature-orientierter Programmierung einheitlich zu behandeln [AH08]. So lässt sich Java in *g*Deep-Modulen darstellen, komponieren und zurück nach Java transferieren. *g*Deep stellt ein Typsystem zur Verfügung, in das die Programmiersprache der Module ebenfalls eingebunden werden kann. Mit geringem Aufwand kann so sichergestellt werden, dass Module in Java, die vor der Komposition wohlgetypt sind, auch nach der Komposition wohlgetypt sind. *g*Deep kann allerdings nur einzelne Produkte komponieren und die Typsicherheit dieses Produktes sicherstellen. Um festzustellen, ob eine Software-Produktlinie typsicher ist, muss jede Variante komponiert werden. Das Konzept der Modul-Komposition wurde allerdings in FeatureHouse umgesetzt und findet damit auch Anwendung in dem Prototypen des in dieser Arbeit vorgestellten Konzeptes.

7. Zusammenfassung und Ausblick

Im Zuge dieser Arbeit sollte ein Familien-basierter Typchecker für FeatureIDE implementiert werden und die Performance mit der eines Produkt-basierten Ansatzes verglichen werden.

Um in der Lage zu sein, Typfehler in Software-Produktlinien zu erkennen, wurden zunächst verschiedene Arten von Fehlern vorgestellt. Composerfehler sind Fehler, die der Composer beim Komponieren der Feature-Module ausgibt. Dazu gehören Syntaxfehler und verschiedene Arten von inkompatiblen Features. Eine zweite Klasse von Fehlern sind Compilerfehler, die erst durch den Compiler der Basissprache gefunden werden. Hierunter fallen etwa benutzte, aber nicht in jeder Variante einer Software-Produktlinie erreichbare Klassen, Methoden und Felder.

Es wurden verschiedene Arten von Korrekturmaßnahmen für Software-Produktlinien vorgeschlagen. Zum einen kann zum Beheben eines gefundenen Fehlers das Feature-Modell angepasst werden. Dies geschieht durch das Hinzufügen von Constraints. Zum anderen kann der Quelltext des Feature-Moduls verändert werden. Beispiele für solche Quelltextanpassungen sind das Ändern eines genutzten Typs, oder das Umbenennen einer Methode. Zudem wurden Problemmeldungen vorgestellt, die es dem Anwender deutlich machen sollen, wo das Problem liegt und es ihm erleichtern, das Problem zu lösen.

Als Hauptaugenmerk dieser Arbeit wurde ein Konzept für einen Typchecker für Feature-orientierte Software-Produktlinien vorgestellt. Dieser Typchecker arbeitet in drei Phasen. In der ersten Phase werden die Feature-Module einer Software-Produktlinie in einen abstrakten Syntaxbaum überführt und es werden Hilfsstrukturen erzeugt. Diese Hilfsstrukturen erlauben im weiteren Verlauf einen effizienten Zugriff auf die Informationen über die Feature-Module. Die zweite Phase besteht aus der eigentlichen Typprüfung der Software-Produktlinie. In dieser Phase festgestellte Fehler werden als Probleme gesammelt. Ein Problem ist dabei ein Fehler, mit zusätzlichen Informationen zum Zustandekommen des Fehlers. Teil des Problems kann zum Beispiel die Stelle im Quelltext sein, an der der Fehler auftritt, oder welcher Typcheck den Fehler meldet. Diese Informationen können bei der Problemlösung in

Phase drei des Typcheckers helfen. In der dritten Phase werden die gesammelten Probleme ausgewertet und Korrekturmaßnahmen ermittelt. Es wurde eine Methode präsentiert, wie Korrekturmaßnahmen nach Relevanz sortiert werden können um zum Beispiel bei einer großen Anzahl an Korrekturmaßnahmen den Überblick zu bewahren.

Das Konzept wurde in einer prototypischen Implementierung umgesetzt. Die einzelnen Typchecks wurden im Prototyp als Plugins implementiert. Dieses Vorgehen macht es einfacher, zusätzliche Typchecks zu implementieren und einzubinden. Es erlaubt außerdem, die vorhandenen Check-Plugins variabler einzusetzen. So ist es möglich bestimmte Check-Plugins, zum Beispiel aus Gründen der Performance, für einen Durchgang nicht mit zuladen. Auch die Problem-Handler sind als Plugins realisiert. Dies erlaubt es, den gleichen Typchecker in verschiedenen Situationen, nur durch das Laden unterschiedlicher Problem-Handler, zu nutzen. So wird die FeatureIDE-Integration geladen, wenn der Typchecker mit FeatureIDE genutzt wird. Dies ist allerdings unnötig, wenn der Typchecker von der Kommandozeile gestartet wird.

Die Evaluierung hat wie erwartet gezeigt, dass der Prototyp des Typcheckers wesentlich performanter als der naive, Produkt-basierte Ansatz ist. So ist der Typchecker für die GPL mehr als 2000-mal schneller als der naive Ansatz. Dem Geschwindigkeitsvorteil steht der große Nachteil gegenüber, dass der Typchecker nur einen kleinen Teil aller möglichen Typfehler finden kann. Mit jedem Check-Plugin, das dem Typchecker hinzugefügt wird, wird die Typprüfung etwas langsamer. Mit den richtigen Typchecks lassen sich allerdings die am häufigsten vorkommenden Typfehler finden. Diese Fehler können im Bruchteil der Zeit gefunden werden, die das Komponieren und Kompilieren jeder Variante benötigen würde.

Die Evaluierung hat ebenfalls gezeigt, dass die Nutzung von inkrementellem Parsen, das heißt, das nur die Feature-Module geparst werden, die sich seit dem letzten Durchgang verändert haben, die Zeit für einen Durchgang des Typcheckers deutlich reduzieren kann. Dies eröffnet die Möglichkeit, bei der Einbindung in FeatureIDE und nach einer erster kompletten Typprüfung, bei jeder Änderung am Quelltext eines Feature-Moduls, eine inkrementelle Typprüfung im Hintergrund durchzuführen.

Der Kompromiss aus Zeitersparnis und gefundenen Fehlern, den der Typchecker bietet, ist in den meisten Fällen ausreichend, wenn eine Software-Produktlinie auf Typsicherheit geprüft werden soll. Ein Großteil der vorkommenden Fehler kann gefunden werden und zusätzliche Typchecks lassen sich, ohne die Ausführungszeit des Typcheckers deutlich zu verlängern, implementieren. Besteht trotzdem die Notwendigkeit, in regelmäßigen Abständen *alle* Fehler zu finden, kann dies durch das Komponieren und Kompilieren jeder Variante geschehen.

Ausblick

Die Implementierung des Typcheckers ist nur der Einstiegspunkt in der Entwicklung eines Systems zur Typprüfung von Feature-orientierten Software-Produktlinien. Es gibt noch viele Möglichkeiten diesen in zukünftigen Arbeiten zu verbessern. Möglichkeiten finden sich zum einen in dem Bereich der Check-Plugins und zum anderen in dem Framework des Typchecker an sich.

Typchecks

Der wichtigste, nächste Schritt ist das Implementieren von weiteren Typchecks. Sinnvoll sind Erreichbarkeitschecks, die prüfen, ob bei einem Feld- oder Methodenzugriff, das Feld beziehungsweise die Methode immer erreichbar ist, wenn das aufrufende Feature gewählt ist.

Ein Vorgehen um weitere Typchecks zu finden, ist das Erzeugen und Kompilieren aller Varianten vorhandener Software-Produktlinien. Auftretende Fehler können analysiert und zur Entwicklung neuer Typchecks genutzt werden. Betrachtet man die Notepad-SPL und die durch den Compiler gefundenen Fehler (siehe Abschnitt 5.3), kann ein Typcheck abgeleitet werden, der die Konstruktoren von Klassen prüft. In der Komposition von Konstruktoren darf jede Variable nur einmal deklariert sein. Auch das Schlüsselwort `super()` darf nur einmal und als erster Aufruf des Konstruktors genutzt werden.

Die vorhandenen Check-Plugins sind in der aktuellen Version nicht korrekt. Der Typreferenz-Check meldet auch ein Problem, wenn auf eine Klasse zugegriffen wird, die aus einem Import stammt, in der aufrufenden Rolle einer Klasse aber nicht importiert wird. Dieser Fehler muss in der Komposition von Feature-Modulen allerdings nicht zu einem Fehler in der Variante führen, da die Importe aus den anderen Rollen in die Komposition übernommen werden. Dieses Problem kann mit einem Check gegen die Importe anderer Rollen einer Klasse behoben werden. Das Finden fehlender Importe sorgt aber dafür, dass der Typreferenz-Check in der aktuellen Version nicht korrekt arbeitet. Zusätzlich kommt es zu Problemen wenn eigene Annotationen genutzt werden, wie es zum Beispiel bei der BerkeleyDB-Software-Produktlinie (liegt FeatureIDE als Beispiel bei) der Fall ist.

Typchecker

Einige der in dieser Arbeit vorgestellten Konzepte wurden noch nicht in dem Prototyp realisiert. Dies betrifft vor allem die Korrekturmaßnahmen. Die vorbereitende Implementierung zum Ermitteln der Korrekturmaßnahmen ist in den Check-Plugins bereits vorhanden, muss aber noch um die eigentliche Funktionalität erweitert werden. Da noch keine Korrekturmaßnahmen ermittelt werden, ist auch die Möglichkeit die Korrekturmaßnahmen nach Relevanz zu sortieren noch zu implementieren. Die globale Liste der Korrekturmaßnahmen ist auch noch umzusetzen. Um von den nach Relevanz sortierten Korrekturmaßnahmen zu profitieren, ist es notwendig, dem Anwender die globale Liste zu präsentieren. Dies kann in FeatureIDE umgesetzt werden.

Der Prototypen ist noch nicht auf Performance optimiert. So werden die abstrakten Syntaxbäume zum Beispiel die ganze Zeit im Speicher gehalten. Eine Möglichkeit diesen Umstand zu umgehen, könnte es sein, alle notwendigen Daten in die Hilfsstrukturen zu kopieren. Die gesammelten Baumknoten können dann nach der Initialisierungsphase der Check-Plugins freigegeben werden.

Der Typchecker unterstützt die inkrementelle Typprüfung in dem Prototypen bisher nur während dem Parsen. Es wäre auch möglich, die Initialisierungsphase des Check-Plugins, also das Erstellen der Hilfsstrukturen, inkrementell zu gestalten.

In der Check-Phase des Check-Plugins ist besonders das Aufrufen des SAT-Solvers aufwendig. Da sich in einem Durchlauf des Typcheckers Anfragen an den SAT-Solver wiederholen können, kann es sinnvoll sein, die Ergebnisse einer Anfrage zwischenzuspeichern. So zeigen Kästner et al. für verschiedene annotierte Software-Produktlinien, dass der Anteil an einzigartigen Anfragen an den SAT-Solver sehr gering ist im Vergleich zu der Gesamtanzahl an Anfragen an den SAT-Solver [KATS11].

In der aktuellen Version des Prototypen können die Check-Plugins und Problem-Handler dem Typchecker nur als Instanzen an den Konstruktor übergeben werden. Um die Flexibilität zu erhöhen könnten die Plugins zum Beispiel als eigene FeatureIDE-Plugins implementiert werden. Eine weitere Möglichkeit ist das Nutzen von Java-Reflections. Die Plugins können dann in einen Ordner kopiert werden und werden von dem Plugin-Manager automatisch geladen, ohne das der Quelltext des Typcheckers angepasst werden muss.

Literaturverzeichnis

[AH08] Sven Apel and Delesley Hutchins. A calculus for uniform feature composition. *ACM Trans. Program. Lang. Syst.*, 32(5):19:1–19:33, May 2008. (zitiert auf Seite 57)

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, July 2009. Refereed Column. (zitiert auf Seite 1, 3 und 8)

[AKGL10] Sven Apel, Christian Kästner, Größlinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering – An International Journal*, 2010. to appear; submitted August 23, 2009; accepted February 3, 2010. (zitiert auf Seite 2, 19, 20, 21, 22, 23, 24, 25, 35 und 56)

[AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society. (zitiert auf Seite 8 und 9)

[ALSU08] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, techniques, and tools. (Compiler. Prinzipien, Techniken und Werkzeuge. Fachliche Betreuung und Erweiterungen: Michael Leuschel.) 2nd ed.* it informatik. München: Pearson Studium. xxxvi, 1253 p. EUR 69.95 , 2008. (zitiert auf Seite 1, 10 und 26)

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *Software Product Line Conference (SPLC)*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005. (zitiert auf Seite 4, 5 und 6)

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling stepwise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society. (zitiert auf Seite 8)

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. (zitiert auf Seite 1)

[CN01] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, August 2001. (zitiert auf Seite 1)

[DCB09] Benjamin Delaware, William R. Cook, and Don S. Batory. Fitting the pieces together: a machine-checked model of safe composition. In *ESEC/SIGSOFT FSE*, pages 243–252, 2009. (zitiert auf Seite 55 und 56)

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. (zitiert auf Seite 20)

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988. (zitiert auf Seite 8)

[KATS11] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2011. accepted for publication. (zitiert auf Seite 56, 57 und 62)

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (zitiert auf Seite 3 und 5)

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997. (zitiert auf Seite 8)

[KTS+09] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: Tool framework for feature-oriented software development. In *Proceedings of the 31th International Conference on Software Engineering (ICSE)*, pages 611–614. IEEE Computer Society, May 2009. Formal Demonstration paper. (zitiert auf Seite 2)

[Lev65] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Doklady Akademii Nauk SSSR*, 163(4):845–848, 1965. Original in Russian – translation in Soviet Physics Doklady 10(8):707-710, 1966. (zitiert auf Seite 19)

[Man02] Mike Mannion. Using first-order logic for product line model validation. In *Proceedings of the Second International Conference on Software Product Lines*, SPLC 2, pages 176–187, London, UK, UK, 2002. Springer-Verlag. (zitiert auf Seite 6)

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. (zitiert auf Seite 1 und 3)

[Pie02] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002. (zitiert auf Seite 1 und 10)

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. pages 419–443. Springer, 1997. (zitiert auf Seite 1 und 8)

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, pages 77–91, 2010. (zitiert auf Seite 8)

[SBD11] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. Compositional type-checking for delta-oriented programming. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 43–56, New York, NY, USA, 2011. ACM. (zitiert auf Seite 56)

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. (zitiert auf Seite 8)

[TAK+12] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. Analysis strategies for software product lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, Germany, April 2012. (zitiert auf Seite 1 und 2)

[TBKC07] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE '07, pages 95–104, New York, NY, USA, 2007. ACM. (zitiert auf Seite 55)

[TKES11] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract features in feature modeling. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 191–200. IEEE Computer Society, August 2011. (zitiert auf Seite 6 und 7)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 02. Juli, 2012